Unlocking Blocked Communicating Processes

Adrian Francalanza
CS, ICT, University of Malta

Marco Giunti
RELEASE, DI, Universidade da Beira Interior
NOVA LINCS, DI-FCT, Universidade NOVA de Lisboa

António Ravara
NOVA LINCS, DI-FCT, Universidade NOVA de Lisboa

We study the problem of disentangling locked processes via code refactoring. We identify and characterise a class of processes that is not lock-free; then we formalise an algorithm that statically detects potential locks and propose refactoring procedures that disentangle detected locks. Our development is cast within a simple setting of a finite linear CCS variant — although it suffices to illustrate the main concepts, we also discuss how our work extends to other language extensions.

1 Introduction

The scenario. Concurrent programming is nowadays pervasive to most computational systems and present in most software development processes. In particular, concurrent programming is prevalent in cloud platforms and web-services, *i.e.,* inherently distributed systems that rely heavily on message-based communication protocols. Unfortunately, this style of programming is notoriously difficult and error-prone: concurrency bugs appear frequently and have a substantial impact, as several recent reports show [7, 2]. Concurrency errors are hard to detect because not every execution interleaving exhibits them, and this is further compounded by the large number of possible execution scenarios. Automatic techniques and tools are thus needed to analyse and ensure correct concurrent code.

One common form of bugs is that of (dead)locks [6]: they arise when a computational entity holds exclusive access to a resource without releasing it, while other entities wait to access that resource. In this work we characterise them in a very simple model of concurrent computation, show how to statically detect them, and in some cases, even show how to automatically solve some of the (dead)locks.

Static analysis to the rescue. Concurrency theory is a well-established discipline, providing mathematical models of concurrent systems at various degrees of expressiveness, (logical) languages to specify properties of such systems, suites of verification techniques of both safety and liveness properties, as well as tools to (automatically) analyse if some property holds for a given specification.

We are interested in models centered around communication primitives and synchronisation mechanisms, as these are the key characteristics of a concurrent system. In particular, we are concerned with the static verification of properties for these models, not only because the approach analyses source code, but also because it is used pre-deployment, in an automatic way. The models are useful to specify and verify communication intensive systems and protocol implementations; the static analysis is a light verification technique that demands less from the user, as (s)he does not have to be an expert in logic.

Concretely, herein we use the Calculus of Communicating Systems (CCS) [12] and define a static analysis and refactoring algorithm that is not only fully automatic, but also working on "pure" source code, without further annotations or even types.

M.H. ter Beek and A. Lluch Lafuente (Eds.): 11th International Workshop on Automated Specification and Verification of Web Systems (WWV'15).
EPTCS 188, 2015, pp. 23–32, doi:10.4204/EPTCS.188.4

Behavioural types. This field of study has gained momentum recently by providing statically more than the usual safety properties: (dead)lock-freedom or even progress can be statically established by relying on a broad spectrum of analysis techniques [3, 4, 5, 1, 8, 9, 11, 13, 14]. Despite their utility, such static detection techniques inevitably approximate the solution (conservatively) — since they are "deciding undecidable properties" — and reject lock-free programs (give false positives). More importantly, however, these techniques simply reject programs, without providing help as to where the problem might be, or providing insights on how to solve the problem detected.

Methodology. Following the approach of Giunti and Ravara [9], in this paper we propose that such techniques go a step further, and provide suggestions on how to fix a detected bug, showing possible patches. In particular, the work in [9] focussed on resolving self-holding (dead)locks, *i.e.,* when a thread holds the resources it wants to use itself. In order to detect such errors, a local analysis within one thread of computation, such as those discussed in [9], sufficed.

By contrast, in this paper we investigate methods for resolving circular-wait (dead)locks, *i.e.,* more general instances where concurrent entities block one another by holding access to a subset of the commonly requested resources. Detecting such (dead)locks requires analyses that spread across parallel threads of computation, and one of the main challenges is to devise static techniques that are compositional (thus scalable) *wrt.* the independent computing entities. For this expository paper, we do not consider the full language of [9]. Instead we distill minimal features of the language — namely synchronisation, prefixes, and parallel composition — that permit us to focus the core challenges to be addressed. However, the ultimate aim of the work is still to address circular-wait (dead)locks in the full language of [9]; in the conclusion, we outline some of the additional issues that arise in the full language setting.

Contributions. § 2 briefly introduces a concise, yet sufficiently expressive, process language to rigorously define (dead)locks. In § 3 we formalise the class of non lock-free processes targeted by our work and give an alternative characterisation for this class. We present an algorithm for statically detecting processes in this class in § 4, and in § 5 we describe disentangling procedures for the detected processes. § 6 concludes. We note that whereas § 3 presents formal results, § 4 and § 5 deal with ongoing work. In particular, they present our general approach by formalising potential algorithms for static analysis and resolution, and outlining the properties that these algorithms are expected to satisfy.

2 Language

We consider a very basic language. Assume a countable set NAMES of *names*, ranged over by $a, b, \ldots$, and a disjoint countable set $\overline{\text{NAMES}}$ of *co-names*, such that for every $a \in \text{NAMES}$ there is a $\overline{a} \in \overline{\text{NAMES}}$; the co-action operation is idempotent *i.e.,* $\overline{\overline{a}} = a$, and let $\alpha, \beta \in \left(\text{NAMES} \cup \overline{\text{NAMES}}\right)$ denote actions.

The grammar in Fig. 1 defines the syntax of the language, a process algebra containing only prefixing and parallel composition, together with action synchronisations akin to CCS [12]. Let $\mathscr{C}[Q]$ (*resp.* $\mathscr{E}[Q]$) be the process obtained by substituting the hole $[-]$ occurring in the context $\mathscr{C}$ (*resp.* $\mathscr{E}$) with Q.

The semantics is standard, relying on a structural equivalence relation $\equiv$ (the smallest congruence including the relation inductively generated by the rules below the grammar) and on a reduction relation $\longrightarrow$, inductively generated by the rules of Fig. 1. Let $\longrightarrow^*$ denote the reflexive and transitive closure of $\rightarrow$.

$$P,Q,R \in \text{PROC} ::= \quad \mathbf{0} \quad (\text{inert}) \mid \alpha.P \quad (\text{prefix}) \quad \mid P \| Q \quad (\text{composition})$$

$$\begin{array}{ll} \mathscr{E} ::= [-] \mid P \| \mathscr{E} \mid \mathscr{E} \| P & \text{(Evaluation Contexts)} \\ \mathscr{C} ::= [-] \mid P \| \mathscr{C} \mid \mathscr{C} \| P \mid \alpha.\mathscr{C} & \text{(Process Contexts)} \end{array}$$

$$\text{sNIL} \quad P \| \mathbf{0} \equiv P \qquad \text{sCOM} \quad P \| Q \equiv Q \| P \qquad \text{sASS} \quad P \| (Q \| R) \equiv (P \| Q) \| R$$

$$\text{COM} \frac{}{a.P \| \overline{a}.Q \longrightarrow P \| Q} \qquad \text{CTX} \frac{P \longrightarrow P'}{\mathscr{E}[P] \longrightarrow \mathscr{E}[P']} \qquad \text{STR} \frac{P \equiv P' \longrightarrow Q' \equiv Q}{P \longrightarrow Q}$$

Figure 1: The language (finite CCS): syntax and operational semantics

Finally, assume henceforth a type system enforcing a linear use of names, $\Gamma \vdash P$, along the lines of the work of Kobayashi [10]. In well-typed processes, no name appears more than once with a given capability (input or output), *i.e.,* a name occurs at most twice in a process, or none at all. The set LPROC is the subset of PROC induced by the typing system $\vdash$.

3 Lock Freedom

Our point of departure is *lock-freedom*, as defined and studied by Kobayashi and by Padovani [10, 13].

Definition 3.1 (Synchronisation predicates [13])**.**

$$\begin{array}{ll} \mathit{in}(a,P) \stackrel{\mathit{def}}{=} \exists P', P'' \cdot P \equiv P' \parallel a.P'' & \mathit{out}(a,P) \stackrel{\mathit{def}}{=} \exists P', P'' \cdot P \equiv P' \parallel \overline{a}.P'' \\ \mathit{sync}(a,P) \stackrel{\mathit{def}}{=} \mathit{in}(a,P) \ \mathit{and} \ \mathit{out}(a,P) & \mathit{wait}(a,P) \stackrel{\mathit{def}}{=} \mathit{in}(a,P) \ \mathit{exor} \ \mathit{out}(a,P) \end{array}$$

Definition 3.2 (Lock-Free [13])**.** *We define* $\mathit{LF} \stackrel{\mathit{def}}{=} \{P \in \text{PROC} \mid \mathit{lfree}(P)\}$ *where:*

$$\mathit{lfree}(P) \stackrel{\mathit{def}}{=} P \longrightarrow^* Q \ \mathit{and} \ \mathit{wait}(a,Q) \quad \mathit{implies} \quad \exists R \cdot Q \longrightarrow^* R \ \mathit{and} \ \mathit{sync}(a,R)$$

Following Def. 3.2, locked processes, $\text{LPROC} \setminus \mathsf{LF}$, are those that *never* provide the *resp.* co-action for some waiting action. In the setting of § 2, this could be due to either of two cases: (i) the co-action is not present in the process; (ii) the co-action is present, but stuck underneath a blocked prefix. Whereas in the case of (i), the context may unlock (*i.e.,* catalyse [4]) the process by providing the necessary co-action, in the case of (ii) *no* context can do so *without violating the linear discipline* of the process. Our work targets the unblocking of this second class of locked processes, specifically by *refactoring* the prefixing of the existing process. To this aim, we introduce the notion of a *complete* process.

Definition 3.3 (Complete Processes)**.** $\text{LPROC} \supseteq \mathit{CMP} \stackrel{\mathit{def}}{=} \{P \mid \mathit{cmp}(P)\}$ *where:*

$$\begin{array}{ll} \mathit{cmp}(P) \stackrel{\mathit{def}}{=} \forall a \cdot \big(\mathit{cin}(a,P) \ \mathit{iff} \ \mathit{cout}(a,P)\big) & \mathit{and} \\ \mathit{cin}(a,P) \stackrel{\mathit{def}}{=} \exists \mathscr{C}[-], Q \cdot P \equiv \mathscr{C}[Q] \ \mathit{and} \ \mathit{in}(a,Q) & \mathit{cout}(a,P) \stackrel{\mathit{def}}{=} \exists \mathscr{C}[-], Q \cdot P \equiv \mathscr{C}[Q] \ \mathit{and} \ \mathit{out}(a,Q) \end{array}$$

Remark 3.4. *In contrast to $in(a,P)$ and $out(a,P)$ of Def. 3.1, the predicates $cin(a,P)$ and $cout(a,P)$ of Def. 3.3 consider actions* under contexts *as well.*

Example 3.5. *The process $P = a.b.\mathbf{0} \parallel \overline{b}.\overline{c}.\mathbf{0}$ is* not complete *since, e.g., $cin(a,P)$ but* not *$cout(a,P)$. The process is also locked, but can be unlocked by the catalyser $[-] \parallel \overline{a}.c.\mathbf{0}$ without violating channel linearity i.e., $[P] \parallel \overline{a}.c.\mathbf{0} \longrightarrow^* \mathbf{0}$. The inert process, $\mathbf{0}$ is clearly lock-free and complete.*

$$P_1 = a.b.\mathbf{0} \parallel \overline{b}.\overline{c}.\mathbf{0} \parallel c.\overline{a}.\mathbf{0} \qquad P_2 = d.(a.b.\mathbf{0} \parallel \overline{b}.\overline{c}.\mathbf{0}) \parallel \overline{d}.c.\overline{a}.\mathbf{0}$$
$$P_3 = a.\overline{a}.\mathbf{0} \qquad P_4 = a.(b.\overline{a}.\mathbf{0} \parallel \overline{b}.\mathbf{0})$$

By contrast, processes P_1, P_2, P_3 and P_4 (above) are both complete but not *lock-free. Note that we rule out complete processes such as $a.\overline{a}.a.\mathbf{0}$ since they violate linearity and are thus not typeable (see § 2).* ■

Our work targets the process class $\mathsf{CMP} \setminus \mathsf{LF}$. In what follows we provide a characterisation for this class that is easier to work with.

Definition 3.6 (Deadlock)**.** $dlock(P) \stackrel{def}{=} \left(\nexists Q \cdot P \longrightarrow Q\right) \quad and \quad P \not\equiv \mathbf{0}$

Definition 3.7 (Top-Complete)**.**

$$tcmp(P) \stackrel{def}{=} (in(a,P) \text{ implies } cout(a,P)) \text{ and } (out(a,P) \text{ implies } cin(a,P))$$

Definition 3.8 (Potentially Self-Locking)**.** $PSL \stackrel{def}{=} \{P \in CMP \mid psl(P)\}$ *where:*

$$sl(P) \stackrel{def}{=} dlock(P) \text{ and } tcmp(P)$$
$$psl(P) \stackrel{def}{=} \exists \mathscr{E}[-], Q \cdot \left(P \longrightarrow^* \mathscr{E}[Q] \text{ and } sl(Q)\right)$$

A self-deadlocked processes, $\mathsf{sl}(P)$, denotes a deadlocked process that cannot be unlocked by a context without violating the linearity discipline, since the *resp.* actions are already present in the process, *i.e.,* $\mathsf{tcmp}(P)$. This, together with $\mathsf{dlock}(P)$, also guarantees that these *resp.* actions will *never* be released. A potentially self-locking process, $\mathsf{psl}(P)$, contains an execution that leads to a top-level subprocess, *i.e., Q* in $\mathscr{E}[Q]$, that is self-deadlocked, $\mathsf{sl}(Q)$: $\mathsf{tcmp}(Q)$ then guarantees Q cannot interact with any of the future reductions of $\mathscr{E}[-]$.

Example 3.9. *Recall the processes in Ex. 3.5. Process P_1 is self-locking, $sl(P_1)$, and thus potentially self-locking as well, $psl(P_1)$. Although P_2 is* not *self-locking, $\neg sl(P_2)$ — it is not deadlocked and can reduce by interacting on name d — it is* potentially *self-locking, $psl(P_2)$, since $P_2 \longrightarrow P_1$.*

Both P_3 and P_4 are self-locking as well, but constitute instances of the self-holding deadlocked processes studied in [9]: in both cases, the locked resource (port a) is blocked by the co-action under the prefix of the same process. Such locks may be detected by a local analysis of the process prefixed by the blocked action. By contrast, in order to determine $psl(P_1)$ and $psl(P_2)$, the analysis needs to spread across parallel processes. ■

The main result of the section is that PSL characterises $\mathsf{CMP} \setminus \mathsf{LF}$.

Theorem 3.10. $PSL = CMP \setminus LF$

Proof. See § A. □

Environment Operations

$$\frac{}{\Gamma+\emptyset=\Gamma} \qquad \frac{\Gamma_1+\Gamma_2=\Gamma_3 \quad a\notin \mathbf{dom}(\Gamma_2)}{(\Gamma_1,a:\rho)+\Gamma_2=\Gamma_3,a:\rho} \qquad \frac{\Gamma_1+\Gamma_2=\Gamma_3}{\Gamma_1,a:\rho+\Gamma_2,a:\mu=\Gamma_3,a:\rho+\mu}$$

Layered Environments and Verdicts, and Operations

$$\Delta\in \textsc{LayeredEnv} ::= \varepsilon \mid \Gamma;\Delta \qquad\qquad \phi\in \textsc{Verdict} ::= \Delta \mid \checkmark$$

$$\frac{}{\Delta+\varepsilon=\Delta} \qquad \frac{\Delta_1+\Delta_2=\Delta_3}{\Gamma_1;\Delta_1+\Gamma_2;\Delta_2=(\Gamma_1+\Gamma_2);\Delta_3} \qquad \frac{}{|\varepsilon|=\emptyset} \qquad \frac{|\Delta|=\Gamma'}{|\Gamma;\Delta|=\Gamma+\Gamma'}$$

$$\Gamma :: \phi \stackrel{\text{def}}{=} \begin{cases} \checkmark & \text{if } \phi=\checkmark \text{ or } \big(\phi=\Delta \text{ and } \mathsf{dlock}(\Gamma) \text{ and } \overline{\Gamma}\subseteq|\Delta|\big) \\ \Gamma;\phi & \text{otherwise} \end{cases}$$

$$\phi_1\oplus\phi_2 \stackrel{\text{def}}{=} \begin{cases} \checkmark & \text{if } \phi_1=\checkmark \text{ or } \phi_2=\checkmark \\ \checkmark & \text{if } \phi_1=\Gamma_1;\Delta_1, \phi_2=\Gamma_2;\Delta_2, \mathsf{dlock}(\Gamma_1+\Gamma_2) \text{ and } \overline{\Gamma_1+\Gamma_2}\subseteq|\Delta_1+\Delta_2| \\ \Delta_1\oplus\Delta_2 & \text{if } \phi_1=\Gamma_1;\Delta_1, \phi_2=\Gamma_2;\Delta_2 \text{ and } \mathsf{cmp}(\Gamma_1+\Gamma_2) \\ \phi_1+\phi_2 & \text{otherwise (since } \phi_1=\Delta_1, \phi_2=\Delta_2) \end{cases}$$

Compositional Static Analysis Rules

$$\textsc{dNil}\ \frac{}{\mathbf{0}\triangleright\emptyset} \qquad \textsc{dIn}\ \frac{P\triangleright\phi}{a.P\triangleright(a:\downarrow::\phi)} \qquad \textsc{dOut}\ \frac{P\triangleright\phi}{\overline{a}.P\triangleright(a:\uparrow::\phi)} \qquad \textsc{dPar}\ \frac{P_1\triangleright\phi_1 \qquad P_2\triangleright\phi_2}{P_1\,\|\,P_2\triangleright\phi_1\oplus\phi_2}$$

Figure 2: Static Analysis for Potential Self-Deadlock

4 Static Detection for Potentially Self-Locking Processes

We devise an algorithm for detecting potentially self-locking processes. To be scalable, the algorithm is *compositional*. The intuition behind is that of constructing *layers* of permission environments $\Gamma_1;\ldots;\Gamma_n$, *approximating* the prefixing found in the process being analysed, and then checking whether this structure satisfies the two conditions defining self-deadlock (see $\mathsf{sl}(-)$ in Def. 3.8), namely that the top environment Γ_1 represents a *deadlock* and that the layered structure is, in some sense, *top-complete*.

Example 4.1. *We determine that the process P_1 from Ex. 3.5 is (potentially) self-locking by constructing the list of layered environments*

$$\underbrace{(a:\downarrow, b:\uparrow, c:\downarrow)}_{\Gamma_1} ; \underbrace{(a:\uparrow, b:\downarrow, c:\uparrow)}_{\Gamma_2} ; \varepsilon$$

and checking that

- *the top environment Γ_1 does not contain any matching permissions, i.e., $\updownarrow\notin \mathbf{cod}(\Gamma_1)$ — this implies that the (composite) process is deadlocked;*
- *that* all *the resp. dual permissions are in Γ_2 — this implies that the (composite) process is blocking itself and cannot be unblocked by an external process composed in parallel with it.*

The main challenge of our compositional analysis is to detect eventual self-deadlocks in cases when the constituent are dispersed across a number of parallel processes. In the case of P_2 of Ex. 3.5, we need to analyse the parallel (sub) processes $d.(a.b.\mathbf{0} \,\|\, \overline{b}.\overline{c}.\mathbf{0})$ and $\overline{d}.c.\overline{a}.\mathbf{0}$ in isolation, *and then determine the eventual deadlock once we merge the sub-analyses; recall, from Ex. 3.9, that P_2 reduces to P_1 from the respective continuations prefixed by d and $\overline{d}$.* ■

Formally, permissions, $\rho, \mu \in \{\downarrow, \uparrow, \updownarrow\}$, denote *resp.* input, output and input-output capabilities. The merge, $\rho + \mu$, and complement, $\overline{\rho}$, (partial) operations are defined as:

$$\downarrow + \uparrow \stackrel{\text{def}}{=} \updownarrow \qquad\qquad \overline{\downarrow} \stackrel{\text{def}}{=} \uparrow \qquad \overline{\uparrow} \stackrel{\text{def}}{=} \downarrow \qquad \overline{\updownarrow} \stackrel{\text{def}}{=} \updownarrow$$

Environments, Γ, are partial maps from names to permissions. We assume the following overloaded notation: complementation, $\overline{\Gamma}$, inverts the respective permissions in $\mathbf{cod}(\Gamma)$ whereas deadlock and complete predicates are defined as:

$$\mathsf{dlock}(\Gamma) \stackrel{\text{def}}{=} \mathbf{cod}(\Gamma) = \{\downarrow, \uparrow\} \qquad\qquad \mathsf{cmp}(\Gamma) \stackrel{\text{def}}{=} \mathbf{cod}(\Gamma) = \{\updownarrow\}$$

The rules in Fig. 2 define the merge operation over environments, $\Gamma_1 + \Gamma_2$ (we elide symmetric rules). Layered environments, Δ, are lists of environments. Our static analysis sequents take the form $P \triangleright \phi$ where ϕ is a *verdict*: it can either be a layered environment or $\checkmark$, denoting a detection. Layered environments may be merged, $\Delta_1 + \Delta_2$, or flattened into a single environment, $|\Delta|$; see Fig. 2.

The static analysis rules are given in Fig. 2, and rely on two verdict operations. Prefixing, $\Gamma :: \phi$, collapses to the definite verdict $\checkmark$ if ϕ was definite or else Γ is deadlocked, $\mathsf{dlock}(\Gamma)$, and top-complete, $\overline{\Gamma} \subseteq |\Delta|$, but creates an *extended* layered environment otherwise. Verdict merging, $\phi_1 \oplus \phi_2$, collapses to $\checkmark$ if either subverdict is a definite detection, or the combined top environments, $\Gamma_1 + \Gamma_2$, satisfy environment deadlock and top-completeness; if $\Gamma_1 + \Gamma_2$ is complete, then it is safe to discard it and check for self-deadlock in the sub-layers (see Rem. 4.3) otherwise the verdicts (which must both be layered environments) are merged.

Example 4.2. *Recall process P_2 from Ex. 4.1. We can derive the sequents:*

$$d.(a.b.\mathbf{0} \,\|\, \overline{b}.\overline{c}.\mathbf{0}) \;\triangleright\; (d:\downarrow);(a:\downarrow,b:\uparrow);(b:\downarrow,c:\downarrow);\varepsilon \tag{1}$$

$$\overline{d}.c.\overline{a}.\mathbf{0} \;\triangleright\; (d:\uparrow);(c:\downarrow);(a:\uparrow);\varepsilon \tag{2}$$

For instance, in the case of (1)*, we first derive the judgements $a.b.\mathbf{0} \triangleright a:\downarrow;b:\downarrow;\varepsilon$ and $\overline{b}.\overline{c}.\mathbf{0} \triangleright b:\uparrow;c:\uparrow;\varepsilon$ using rules* DNIL, DIN *and* DOUT*. Applying* DPAR *on these two judgements requires us to calculate*

$$(a:\downarrow;b:\downarrow;\varepsilon) \oplus (b:\uparrow;c:\uparrow;\varepsilon) \;=\; (a:\downarrow;b:\downarrow;\varepsilon) + (b:\uparrow;c:\uparrow;\varepsilon) \;=\; (a:\downarrow,b:\uparrow);(b:\downarrow,c:\downarrow);\varepsilon$$

using the definition of $\phi_1 \oplus \phi_2$ from Fig. 2. We thus obtain (1) *by applying* DIN *on the resultant judgement.*

Importantly, when we use rule DPAR *again, this time to merge judgements* (1) *and* (2)*, the definition of $\phi_1 \oplus \phi_2$ allows us to reexamine the environments in the sub-layers, since the merged top-layer is complete, $\mathsf{cmp}(d:\downarrow + d:\uparrow)$, from which we infer that the top actions guarding the merged parallel processes will safely interact and release the processes in the sub-layers. Stated otherwise, we obtain:*

$$\big(d:\downarrow;(a:\downarrow,b:\uparrow);(b:\downarrow,c:\downarrow);\varepsilon\big) \oplus \big(d:\uparrow;c:\downarrow;a:\uparrow;\varepsilon\big) \;=\; \big((a:\downarrow,b:\uparrow);(b:\downarrow,c:\downarrow);\varepsilon\big) \oplus \big(c:\downarrow;a:\uparrow;\varepsilon\big) \;=\; \checkmark$$

since $\mathsf{dlock}((a:\downarrow,b:\uparrow)+(c:\downarrow))$ and $\overline{a:\downarrow,b:\uparrow,c:\downarrow} \subseteq |\big((b:\downarrow,c:\downarrow);\varepsilon\big) + \big(a:\uparrow;\varepsilon\big)| = (b:\downarrow,c:\downarrow,a:\uparrow)$. ■

$$\Gamma \propto_1 \mathbf{0} \stackrel{\text{def}}{=} \mathbf{0} \qquad \Gamma \propto_1 P \,\|\, Q \stackrel{\text{def}}{=} (\Gamma \propto_1 P) \,\|\, (\Gamma \propto_1 Q)$$

$$\Gamma \propto_1 \overline{a}.P \stackrel{\text{def}}{=} \begin{cases} \overline{a}.\mathbf{0} \,\|\, P & \Gamma(a) = \uparrow \\ \overline{a}.(\Gamma \propto_1 P) & \text{otherwise} \end{cases} \qquad \Gamma \propto_1 a.P \stackrel{\text{def}}{=} \begin{cases} a.\mathbf{0} \,\|\, P & \Gamma(a) = \downarrow \\ a.(\Gamma \propto_1 P) & \text{otherwise} \end{cases}$$

$$\Gamma \propto_2 \mathbf{0} \stackrel{\text{def}}{=} \mathbf{0} \qquad \Gamma \propto_2 P \,\|\, Q \stackrel{\text{def}}{=} (\Gamma \propto_2 P) \,\|\, (\Gamma \propto_2 Q)$$

$$\Gamma \propto_2 \overline{a}.P \stackrel{\text{def}}{=} \begin{cases} \overline{a}.\mathbf{0} \,\|\, (\Gamma \propto_2 P) & \Gamma(a) = \uparrow \\ (\Gamma \propto_2 P) & \Gamma(a) = \downarrow \\ \overline{a}.(\Gamma \propto_2 P) & \text{otherwise} \end{cases} \qquad \Gamma \propto_2 a.P \stackrel{\text{def}}{=} \begin{cases} a.(\Gamma \propto_2 P) \,\|\, \overline{a}.\mathbf{0} & \Gamma(a) = \downarrow \\ a.(\Gamma \propto_2 P) & \text{otherwise} \end{cases}$$

Figure 3: Disentangling for Potential Self-Deadlock

Remark 4.3. *When merging verdicts, it is unsafe to ignore individual complete mappings, e.g., $a:\updownarrow$, even though this makes the analysis imprecise. It is only safe to ignore them (and check for potential deadlocks in lower layers) when the* entire *top environment is complete, i.e., $cmp(\Gamma)$. As a counter-example justifying this, consider the lock-free process $(a.\overline{b}.\mathbf{0} \,\|\, b.\mathbf{0}) \,\|\, \overline{a}.\mathbf{0}$. We currently deduce*

$$(a.\overline{b}.\mathbf{0} \,\|\, b.\mathbf{0}) \triangleright (a:\downarrow, b:\downarrow); b:\uparrow; \varepsilon \quad \textit{and} \quad \overline{a}.\mathbf{0} \triangleright a:\uparrow; \varepsilon \quad \textit{where} \quad \overline{a:\downarrow, b:\downarrow + a:\uparrow} = a:\updownarrow, b:\uparrow \not\supseteq b:\uparrow$$

However, eliding $a:\updownarrow$ from the analysis, i.e., assuming that $(a:\downarrow, b:\downarrow) + a:\uparrow = b:\uparrow$, yields an unsound detection. Precisely, when merging the sub-verdicts for rule DPAR, *$\big((a:\downarrow, b:\downarrow); b:\uparrow; \varepsilon\big) \oplus \big(a:\uparrow; \varepsilon\big)$, we would first obtain $dlock((a:\downarrow, b:\downarrow) + a:\uparrow)$ and moreover that $\overline{(a:\downarrow, b:\downarrow) + a:\uparrow} = b:\uparrow$ is a subset of $|(b:\uparrow; \varepsilon) + \varepsilon| = b:\uparrow$, which yields ✓ according to Fig. 2.* ■

We expect the judgement $P \triangleright \checkmark$ to imply $\mathsf{psl}(P)$, which would in turn imply $\neg\mathsf{lfree}(P)$ by Thm. 3.10. We leave the proof of the first implication for future work.

5 Disentangling Potentially Self-Locking Processes

To illustrate the ultimate aim of our study, we outline possible disentangling functions that refactor a potentially self-deadlocked process into a corresponding lock-free process. These disentangling functions are meant to be used in conjunction with the detection algorithm of § 4 as a static analysis tool for automating the disentangling of processes. There are a number of requirements that a disentangling algorithm should satisfy. For instance, it should not violate any safety property that is already satisfied by the entangled process (*e.g.,* if an entangled process P type-checked according to some typing discipline, *i.e.,* $\Gamma \vdash P$, the resulting disentangled processes, say Q, should still typecheck *wrt.* the same type discipline/environment, *i.e.,* $\Gamma \vdash Q$). Additionally, one would also expect the resultant disentangled process to be lock-free, as expressed in Def. 3.2, or at the very least to resolve a subset of the locks detected. But there are also a number of additional possibilities for what constitutes a valid process disentangling. Within the simple language of § 2, we can already identify at least two (potentially conflicting) criteria:

1. the order of name usage respects that dictated by the innermost prefixing of the entangled process. Stated otherwise, any locks are assumed to be caused by prefixing at the top-level of the process.
2. the order of input prefixes in the entangled process should be preserved.

We envisage a straightforward extension to the system $P \triangleright \phi$ of § 4, with extended detection reports, $\langle \checkmark, \Gamma \rangle$. The tuple $\langle \checkmark, \Gamma \rangle$, in some sense, *explains* the source of the problem detected by including the offending top-layer environment of a self-deadlock, Γ; this information is then used by the disentangling procedure to refactor the detected process.

Fig. 3 defines two disentangling functions that take this (top-layer) environment and the *resp.* detected process as input, and return a refactored process as output. The first function, $\Gamma \propto_1 P$, translates problematic prefixing (as dictated by Γ) into parallel compositions. The second function $\Gamma \propto_2 P$ operates asymmetrically on input and output prefixes: whereas problematic outputs are treated as before, blocked inputs are *not* parallelised; instead the *resp.* output is *pulled out* at input level.

Example 5.1. *The algorithm of § 4 could detect P_5 (below) as $P_5 \triangleright \langle \checkmark, (a:\downarrow, c:\uparrow) \rangle$, where $\Gamma = (a:\downarrow, c:\uparrow)$*

$$P_5 = a.\overline{b}.c.\mathbf{0} \parallel \overline{c}.b.\overline{a}.\mathbf{0}$$

Using the offending top-layer environment Γ, we can apply the two disentangling algorithms of Fig. 3 and obtain the following:

$$\Gamma \propto_1 P_5 = (a.\mathbf{0} \parallel \overline{b}.c.\mathbf{0}) \parallel (\overline{c}.\mathbf{0} \parallel b.\overline{a}.\mathbf{0}) \tag{3}$$

$$\Gamma \propto_2 P_5 = (\overline{a}.\mathbf{0} \parallel a.\overline{b}.c.\mathbf{0}) \parallel (\overline{c}.\mathbf{0} \parallel b.\mathbf{0}) \tag{4}$$

While both refactored processes are lock-free, it turns out that the first disentangling function observes the first criteria: in the refactored process, (3)*, interactions on a and c happen* after *interactions on b, since these names are (both) prefixed by b (and $\overline{b}$) at the innermost level of P_5. Conversely, the second disentangling function observes the second criteria discussed above: in the refactored process,* (4)*, the input prefixing that orders a before c in P_5 is preserved (this was not the case in* (3)*). Note that both refactorings preserve channel linearity (a safety criteria) while returning lock-free processes.* ■

6 Conclusion

We have outlined our strategy for automating correct disentangling of locked processes, generalising preliminary results previously presented [9]. Although we limited our discussion to a very simple language — the variant of the finite CCS without recursion, choice or name scoping — this was expressive enough to focus on the usefulness of the concepts and techniques we propose, *i.e.,* resolving circular locks across parallel compositions. We define precisely the class of (dead)locked processes within this setting, and provide a faithful characterisation of them in terms of a novel notion: potentially self-locking processes.We also devised a compositional algorithm to statically detect these processes and unlock them, improving previous results (cf. [9]). In particular, Giunti and Ravara [9] used a different technique (based on balanced session types) and could only disentangle self-holding deadlocks such as those in processes P_3 and P_4 of Ex. 3.5. The technique does not support reasoning about (and disentangle) locks across parallel compositions, such as those shown for processes P_1 and P_2 of Ex. 3.5 and P_5 of Ex. 5.1.

We expect the concepts and techniques developed to carry over to more expressive languages. We are considering language extensions such as process recursion, unrestricted channel names (to allow non-determinism), and value-passing. For instance, disentangling the value-passing program (an extension of the process P_5 in Ex. 5.1) $P_6 = a(x).\overline{b}\langle x+1 \rangle.c(y).\mathbf{0} \parallel \overline{c}\langle 5 \rangle.b(z).\overline{a}\langle 7 \rangle.\mathbf{0}$ may not be possible for certain disentangling functions (and criteria) *e.g.,* $\Gamma \propto_1 P$, whereas others may require auxiliary machinery, *e.g.,* the **findVal**$(-)$ function used by [9] for pulling out the *resp.* output values in $\Gamma \propto_2 P$; in (complete) linear settings, there is a unique output for any particular channel, which can be obtained through a linear scan

of the process. The input binding structure may also make certain processes impossible to disentangle. *E.g.,* consider a modification in P_6 above where the value 7 is changed to the bound value z. This would create a circular binding dependency: one between the input on channel a and the output on b through variable x, but also another one between the input on b and the output on a through variable z. These issues will all be considered in future work.

Acknowledgements

Adrian Francalanza was supported by the grants ECOST-STSM-IC1201-280114-038254 and ECOST-STSM-IC1201-250115-054509. Marco Giunti was supported by the grant ECOST-STSM-IC1201-220713-032903 and by the Software Testing Center, Centro de Negócios e Serviços Partilhados do Fundão. António Ravara was supported by the grant ECOST-STSM-IC1201-210713-033367. Marco Giunti and António Ravara were also supported by the grant FCT/MEC NOVA LINCS PEst UID/CEC/04516/2013.

References

[1] M. Coppo et al. (2013): *Inference of Global Progress Properties for Dynamically Interleaved Multiparty Sessions*. In: *COORDINATION, LNCS* 7890, pp. 45–59, doi:10.1007/978-3-642-38493-6_4.

[2] Shan Lu et al (2008): *Learning from mistakes: a comprehensive study on real world concurrency bug characteristics*. In: *ASPLOS'08*, pp. 329–339, doi:10.1145/1346281.1346323.

[3] Luís Caires & Hugo Torres Vieira (2010): *Conversation types*. *Theoretical Computer Science* 411(51-52), pp. 4399–4440, doi:10.1016/j.tcs.2010.09.010.

[4] Marco Carbone, Ornela Dardha & Fabrizio Montesi (2014): *Progress as Compositional Lock-Freedom*. In: *COORDINATION, LNCS* 8459, pp. 49–64, doi:10.1007/978-3-662-43376-8_4.

[5] Marco Carbone & Fabrizio Montesi (2013): *Deadlock-freedom-by-design: multiparty asynchronous global programming*. In: *POPL*, pp. 263–274, doi:10.1145/2429069.2429101.

[6] Edward G. Coffman, Melanie Elphick & Arie Shoshani (1971): *System deadlocks*. *ACM Computing Surveys* 3(2), pp. 67–78, doi:10.1145/356586.356588.

[7] Pedro Fonseca, Cheng Li, Vishal Singhal & Rodrigo Rodrigues (2010): *A study of the internal and external effects of concurrency bugs*. In: *DSN*, pp. 221–230, doi:10.1109/DSN.2010.5544315.

[8] Elena Giachino, Naoki Kobayashi & Cosimo Laneve (2014): *Deadlock Analysis of Unbounded Process Networks*. In: *CONCUR, LNCS* 8704, pp. 63–77, doi:10.1007/978-3-662-44584-6_6.

[9] Marco Giunti & António Ravara (2014): *Towards Static Deadlock Resolution in the Pi-Calculus*. In: *TGC'13, LNCS* 8358, pp. 136–155, doi:10.1007/978-3-319-05119-2_9.

[10] Naoki Kobayashi (2000): *Type Systems for Concurrent Processes: From Deadlock-Freedom to Livelock-Freedom, Time-Boundedness*. *IFIP TCS* 1872, pp. 365–389, doi:10.1007/3-540-44929-9_27.

[11] Naoki Kobayashi (2006): *A New Type System for Deadlock-Free Processes*. In: *CONCUR, LNCS* 4137, pp. 233–247, doi:10.1007/11817949_16.

[12] Robin Milner (1980): *A Calculus of Communicating Systems*. *LNCS* 92, doi:10.1007/3-540-10235-3.

[13] Luca Padovani (2014): *Deadlock and lock freedom in the linear pi-calculus*. In: *LICS*, doi:10.1145/2603088.2603116.

[14] Hugo Torres Vieira & Vasco Thudichum Vasconcelos (2013): *Typing progress in communication-centred systems*. In: *COORDINATION, LNCS* 7890, pp. 236–250, doi:10.1007/978-3-642-38493-6_17.

A Proofs

This section is devoted to the proof of Theorem 3.10. We start with some auxiliary definitions and lemmas.

Given a process P of Figure 1, we indicate with $\text{NAMES}(P)$ the subset of NAMES induced by the rule $\text{NAMES}(a.P) = \{a\} \cup \text{NAMES}(P)$: the remaining cases are homomorphic. We use $\sqcup$ for disjoint union of sets.

We remember that we assume that the processes P of our interest are linear, that is they never contain two or more inputs or outputs on the same channel, and deploy the following results.

Lemma A.1. *If $P \to P'$ then there is port, a, such that $\text{NAMES}(P') \sqcup \{a\} = \text{NAMES}(P)$ and $\mathsf{sync}(a,P)$.*

Proof. By induction on the rules of Figure 1; straightforward. □

Corollary A.2. *If $P \to^* Q$ then the following holds:*

1. $\text{NAMES}(Q) \subseteq \text{NAMES}(P)$
2. *if $\text{NAMES}(P) \backslash \text{NAMES}(Q) = \{a, \dots\}$ then there exists a P_a and a P'_a such that $P \to^* P_a \to P'_a \to^* Q$ with $\text{NAMES}(P'_a) \sqcup \{a\} = \text{NAMES}(P_a)$ and $\mathsf{sync}(a,P_a)$.*

Lemma A.3. *If $\mathsf{cin}(a,P)$ ($\mathsf{cout}(a,P)$) and $P \to^* Q$ then exactly one of the following cases holds:*

1. *$\mathsf{cin}(a,Q)$ ($\mathsf{cout}(a,Q)$)*
2. *$a \notin \text{NAMES}(Q)$ and there exists an R_a and an R'_a such that $P \to^* R_a \to R'_a \to^* Q$ and $\text{NAMES}(R'_a) \sqcup \{a\} = \text{NAMES}(R_a)$ and $\mathsf{sync}(a,R_a)$.*

Proof. We have two cases corresponding to (i) $a \in \text{NAMES}(Q)$ or (ii) $a \notin \text{NAMES}(Q)$. In case (i), assume $P \to P_1 \to_1 \cdots \to_n Q_n \to Q$. We proceed by induction on n. From Lemma A.1 we know that there exists b such that $\text{NAMES}(P_1) \sqcup \{b\} = \text{NAMES}(P)$ and $\mathsf{sync}(b,P)$. Since $\text{NAMES}(Q) \subset \text{NAMES}(P_1)$, we infer $a \in \text{NAMES}(P_1)$ and in turn $a \neq b$. From this and $\mathsf{cin}(a,P)$ we deduce that $\mathsf{cin}(a,P')$. Now assume that $\mathsf{cin}(a,Q_n)$. From Lemma 1 we deduce $\text{NAMES}(Q) \sqcup \{c\} = \text{NAMES}(Q_n)$ for some $c \neq a$: thus $\mathsf{cin}(a,Q)$. The case $\mathsf{cout}(a,P)$ is analogous. Case (ii) is a direct consequence of Corollary A.2.

□

Lemma A.4. *If $P \in \mathit{CMP}$ and $P \to^* Q$ then $Q \in \mathit{CMP}$.*

Lemma A.5. *For any P there exists Q such that $P \to^* Q$ and $Q \not\to$.*

Proof of Theorem 3.10. To show the right to the left direction, assume $P \in \mathsf{CMP} \setminus \mathsf{LF}$. By definition:

$$\mathsf{CMP} \setminus \mathit{LF} \stackrel{\text{def}}{=} \{P \in \mathsf{CMP} \mid \exists (Q,a)\ .\ P \to^* Q \ \wedge\ \mathsf{wait}(a,Q) \Rightarrow \forall R\ .\ Q \to^* R \Rightarrow \neg\mathsf{sync}(a,R)\}$$

Let Q_a be a distinctive redex of P: thus $\mathsf{in}(a,Q_a)$ exor $\mathsf{out}(a,Q_a)$. Assume $\mathsf{in}(a,Q_a)$ and consider $R_{\mathbf{stop}}$ such that $Q_a \to^* R_{\mathbf{stop}} \not\to$, which does exists by Lemma A.5. By Lemma A.3 we know that $\mathsf{in}(a,R_{\mathbf{stop}})$: from $\neg\mathsf{sync}(a,R_{\mathbf{stop}})$ we infer $\neg\mathsf{out}(a,R_{\mathbf{stop}})$. From Lemma A.4 we infer $R_{\mathbf{stop}} \in \mathsf{CMP}$: thus $\mathsf{cout}(a,R_{\mathbf{stop}})$. Therefore $\mathsf{dlock}(R_{\mathbf{stop}})$ and $\mathsf{tcmp}(R_{\mathbf{stop}})$, as required. The case $\mathsf{out}(a,Q_a)$ is analogous.

To see the left to the right direction, assume that $P \in \mathsf{CMP}$ and that $P \longrightarrow^* \mathscr{E}[Q]$ with $\mathsf{dlock}(Q)$ and $\mathsf{tcmp}(Q)$. Note that this excludes the case $Q \equiv \mathbf{0}$: therefore $\text{NAMES}(Q) \neq \emptyset$, and in turn $\text{NAMES}(P) \neq \emptyset$, because of Corollary A.2. From $\text{NAMES}(Q) \neq \emptyset$ and the rules of structural congruence we infer that there is $a \in \text{NAMES}(Q)$ such that (i) $Q \equiv a.Q' \parallel Q''$ or (ii) $Q \equiv \overline{a}.Q' \parallel Q''$. In case (i) we infer $\mathsf{in}(a,Q)$; from $Q \not\to$ we deduce $\neg\mathsf{out}(a,Q)$; in case (ii) we infer $\mathsf{out}(a,Q)$; from $Q \not\to$ we deduce $\neg\mathsf{in}(a,Q)$. In both cases we infer $\mathsf{wait}(a,Q)$, and in turn $\neg\mathsf{sync}(a,Q)$ which completes the proof since Q has no redexes: that is, $P \in \mathsf{CMP} \setminus \mathit{LF}$. □