# Minimal Translations from Synchronous Communication to Synchronizing Locks

Manfred Schmidt-Schauß
Goethe-University, Frankfurt am Main, Germany
schauss@ki.cs.uni-frankfurt.de

David Sabel
LMU, Munich, Germany
david.sabel@ifi.lmu.de

In order to understand the relative expressive power of larger concurrent programming languages, we analyze translations of small process calculi which model the communication and synchronization of concurrent processes. The source language SYNCSIMPLE is a minimalistic model for message passing concurrency while the target language LOCKSIMPLE is a minimalistic model for shared memory concurrency. The former is a calculus with synchronous communication of processes, while the latter has synchronizing mutable locations – called locks – that behave similarly to binary semaphores. The criteria for correctness of translations is that they preserve and reflect may-termination and must-termination of the processes. We show that there is no correct compositional translation from SYNCSIMPLE to LOCKSIMPLE that uses one or two locks, independent from the initialisation of the locks. We also show that there is a correct translation that uses three locks. Also variants of the locks are taken into account with different blocking behavior.

## 1 Introduction

Different models of concurrency are studied and used in theory and in practice of computer science. One main approach are *message passing models* where the concurrently running threads (or processes) communicate by sending and receiving messages. A prominent example for a message passing model is the $\pi$-calculus [6, 16]. There exist approaches with asynchronous and with synchronous message passing. In asynchronous message passing, a sender sends its message and proceeds *without* waiting that a receiver collects the message (thus the message is kept in some medium until the receiver collects it from that medium). In synchronous message passing, the message is exchanged in one step and thus sender and receiver wait until the communication has happened. Thus, synchronous message passing can be used for synchronization of processes.

Another main approach for concurrency are program calculi with *shared memory* where concurrent processes communicate by using shared memory primitives. For instance, $\lambda(\mathbf{fut})$ [7] is a program calculus that models the core of the strict concurrent functional language Alice ML, and it has concurrent threads, handled futures, and memory cells with an atomic exchange-operation. Also other shared memory synchronization primitives like concurrent buffers and their encodability into $\lambda(\mathbf{fut})$ are analyzed [24]. Other examples are the calculi CH [19] and CHF [14, 15, 21]. The latter is a program calculus that models the core of Concurrent Haskell [10]: it extends the functional programming language Haskell by concurrent threads and so-called MVars, which are synchronizing mutable memory locations. Thus, depending on the model (or the concurrent programming languages) there exist different primitives. The simplest approach is some kind of locking primitive to block a process until some event happens. To exchange a message, for instance, atomic read-write registers can be used. More sophisticated primitives are for example semaphores, monitors, or Concurrent Haskell's MVars. All these approaches have in common that processes can be blocked until an event occurs, which is performed by another process.

V. Castiglioni and O. Dardha (Eds.): Combined Workshop on Expressiveness in Concurrency and Structural Operational Semantics (EXPRESS/SOS 2021)
EPTCS 339, 2021, pp. 59–75, doi:10.4204/EPTCS.339.7

Expressivity of (concurrent) programming languages is an important topic, since the corresponding results allow us to classify the languages and their programming constructs, and to understand their differences. Investigating the expressivity to clarify the relation between message passing models and shared memory concurrency can in principle be done by constructing correct translations from one model to the other. Our research considers the question whether and how synchronous message passing can be implemented by models that support shared memory and some of these synchronization primitives.

In previous work [19], we analyzed translations from the synchronous $\pi$-calculus into a core language of Concurrent Haskell. In particular, we looked for compositional translations that preserve and reflect the convergence behavior of processes (in all program-contexts) w.r.t. may- and should-convergence. This means, processes can successfully terminate or not, where may-convergence observes whether there is a possible execution path to a successful process and should-convergence means that the ability to become successful holds for all execution paths. We found correct translations and proved them to be correct with respect to this correctness notion. Looking for small translations has several advantages: The resource usage of the translated programs is lower, they are easier to understand than larger ones, and the corresponding correctness proofs often are easier than for large ones. Hence, we also tried to find smallest translations, but in the end we could not answer the following question: what is the minimal number of MVars that are necessary to correctly encode the message passing synchronization using MVars? This leads us to the general question how synchronous communication can be encoded by synchronizing primitives and what is the minimal number of primitives that is required. This question is addressed in this paper. We choose to work with models that are as simple as possible and also as complex as needed, but nevertheless are also relevant for full programming languages (we discuss the transportion of the results to full languages in Section 2.3). Thus we consider translations from a small message passing source language into a small target language with shared memory concurrency and synchronizing primitives.

For the source language SYNCSIMPLE, we use a minimalistic model for concurrent processes that synchronize by communication. The language has constructs for sending (denoted by "!") and for receiving (denoted by "?"). A communication step atomically processes one ! from one process together with one ? from another process. For simplicity, there is no message that is sent and there are no channel names (i.e. the language can be seen as a variant of the synchronous $\pi$-calculus (without replication and sums) where only one global channel name exists).

For the target language LOCKSIMPLE we choose a similar calculus where the communication is removed and replaced by synchronizing shared memory locations. These locations are called *locks*. A lock can be empty or full. There are operations to fill an empty lock (put) or to empty a full lock (take). The main variant that we consider is the one where the put-operation blocks on a full lock, but the take-operation is not blocking on an empty lock. Thus these locks are like binary semaphores where put is the wait-operation and take is the signal-operation (where signal on an unlocked semaphore is allowed but has no effect). We also consider the language with several locks with different initializations (empty or full). Based on this setting, the question addressed by the paper is:

> *What is the minimal number of locks that is required to correctly translate the source calculus into the target calculus?*

The notion of correctness of a translation requires comparing the semantics of both calculi. We adopt the approach of observational correctness [17, 22] and thus we use correctness w.r.t. a contextual equivalence which considers the may- and the must-convergence in both calculi. May-convergence means that the process can be evaluated to a successful process (in both calculi we add a constant to signal success). Due to the nondeterminism, observing may-convergence is too weak since for instance,

it equates processes that must become successful with processes that either diverge or become successful. Hence we also observe must-convergence, which holds if any evaluation of the process ends with a successful process. Considering must-convergence only is also too weak since it equates processes that always fail with processes that either fail or become successful. Thus we use the combination of both convergencies as program semantics. In turn, a correct translation must preserve and reflect the may- and must-convergence of any program.

This can also be seen as a minimalistic requirement on a correct translation since for instance, requiring equivalence of strong or weak bisimulation (see e.g. [16]) would be a much stronger requirement.

*Results.* We show that there does not exist a correct compositional translation from SYNCSIMPLE into LOCKSIMPLE that uses one (Theorem 3.2) or two locks (Theorem 5.17), while there is a correct compositional translation that uses three locks (Theorem 2.9).

The non-existence is proved for any initial state of the lock variables and also for different kinds of blocking behaviour of the lock (i.e. whether put or whether take blocks).

*Related Work.* Validity of translations between process calculi is discussed in [4, 3] where five criteria for valid translations resp. encodings are proposed: compositionality, name invariance, operational correspondence, divergence reflection, and success sensitiveness. Compositionality and name invariance restrict the syntactic form of the translated processes; operational correspondence means that the transitive closure of the reduction relation is transported by the translation, modulo the syntactic equivalence; and divergence reflection and success sensitiveness are conditions on the semantics.

We adopt the first condition for our non-encodability results since we will require that the translation is compositional. The name invariance is irrelevant since our simple calculi do not have names. We do not use the third condition in the proposed form, since it has a flavour of showing equivalence of bisimulations, instead, we require equivalence of may- and must-convergence which is a bit weaker. Thus, for our non-encodability result the property could be included (still showing non-encodability), but for the correct translation in Theorem 2.9, we did not check the property. Convergence equivalence for may- and must-convergence is our replacement of Gorla's divergence reflection and success sensitiveness.

Translations from synchronous to asynchronous communication are investigated in the $\pi$-calculus [5, 1, 9, 8]. Encodability results are obtained for the $\pi$-calculus without sums [5, 1], while Palamidessi analyzed synchronous and asynchronous communication in the $\pi$-calculus with *mixed sums* and non-encodability is the main result [8, 9].

A high-level encoding of synchronous communication into shared memory concurrency is an encoding of CML-events in Concurrent Haskell using MVars [12, 2], however a formal correctness proof for the translation is not provided.

*Outline.* In Section 2 we introduce the process language SYNCSIMPLE with synchronous communication and the process language LOCKSIMPLE with asynchronous locks. After defining the correctness conditions on translations, we show that three locks (with a specific initialization) are sufficient for a correct translation and we discuss variants of the target language. In particular, we show that changing blocking variants is equivalent to a modification of the initial store. In Section 3 it is shown that one lock in LOCKSIMPLE is insufficient for a correct translation and Section 4 exhibits certain general properties of correct translations which use two or more locks. Section 5 contains the structuring into different blocking types of translations, and proofs that there are no correct translations for two locks and any initial store. Section 6 concludes the paper. For space reasons some proofs are omitted, but they can be found in the extended version of this paper [23].

# 2 Languages for Concurrent Processes

We define abstract and simple models for concurrent processes with synchronous communication and for concurrency with synchronizing shared memory. The former model is a simplified variant of the $\pi$-calculus with a single global channel name and without replication or recursion, the latter can be seen as a variant where interprocess communication is replaced by binary semaphores. Thereafter we define correct translations, prove correctness of a specific translation and consider variants of the target language.

## 2.1 The Calculus SYNCSIMPLE

**Definition 2.1.** *The syntax of processes and subprocesses of the calculus* SYNCSIMPLE *is defined by the following grammar, where $i \in \{1,\ldots,k\}$:*

$$\begin{array}{ll} \textit{Subprocesses} & \mathscr{U} ::= \checkmark \mid 0 \mid !\mathscr{U} \mid ?\mathscr{U} \\ \textit{Processes} & \mathscr{P} ::= \mathscr{U} \mid \mathscr{U} \mid \mathscr{P} \end{array}$$

We informally describe the meaning of the symbols. The symbol 0 means the silent subprocess; the symbol ✓ means success, The operation ! means an output (or send-command), and ? means an input (or receive-command), and | is parallel composition. For example, the expression ?!!✓ | !?0 is a process, and so are also ???!!!?✓ and ?!!✓ | !?0 | ✓ | !!!!!!?!✓. We assume that | is commutative and associative and that 0 is an identity element w.r.t. |, i.e. $0 \mid P = P$ for all $P$. Thus a process can be seen as a multiset of subprocesses.

**Definition 2.2.** *The* operational semantics *of* SYNCSIMPLE *is a (non-deterministic) small-step operational semantics. A single step $\xrightarrow{SYS}$ is defined as*

$$!\mathscr{U}_1 \mid ?\mathscr{U}_2 \mid \mathscr{P} \xrightarrow{SYS} \mathscr{U}_1 \mid \mathscr{U}_2 \mid \mathscr{P}$$

*where $\mathscr{U}_1, U_2$ are arbitrary subprocesses and $\mathscr{P}$ is an arbitrary process.*

*The reflexive-transitive closure of $\xrightarrow{SYS}$ is denoted as $\xrightarrow{SYS,*}$.*

*If a process is of the form ✓ | $\mathscr{P}$, then the process is* successful. *A sequence of $\xrightarrow{SYS}$-steps starting with $\mathscr{P}$ is called* an execution of $\mathscr{P}$.

Note that there may be several executions of processes, but every execution terminates.

**Example 2.3.** *Two examples for the execution of $P =$ ?!0 | !!✓ | ?0 are:*

- $P =$ ?!0 | !!✓ | ?0 $\xrightarrow{SYS}$ !0 | !✓ | ?0 $\xrightarrow{SYS}$ !0 | ✓ | 0*, where the final process is successful.*
- $P =$ ?!0 | !!✓ | ?0 $\xrightarrow{SYS}$ !0 | !✓ | ?0 $\xrightarrow{SYS}$ 0 | !✓ | 0 *where the final process is terminated, but not successful.*

*This means there may be executions leading to a successful process, and at the same time executions leading to a fail.*

We often omit the suffix 0 for a subprocess, i.e. whenever a subprocess ends with symbol ! or ? we mean the same subprocess extended by 0.

**Definition 2.4.** *A process $\mathscr{P}$ is called*

- may-convergent *if there is some successful process $\mathscr{P}'$ with $\mathscr{P} \xrightarrow{SYS,*} \mathscr{P}'$.*

- must-convergent *if for all processes $\mathscr{P}'$ with $\mathscr{P} \xrightarrow{SYS,*} \mathscr{P}'$, the process $\mathscr{P}'$ is may-convergent.*
- must-divergent *or a* fail*, if there is no execution leading to a successful process.*
- may-divergent*, if there exists an execution $\mathscr{P} \xrightarrow{SYS,*} \mathscr{P}'$, where $\mathscr{P}'$ is a fail.*

Our definition of must-convergence is the same as so-called should-convergence (see e.g. [13, 18, 14]). However, since there are no infinite reduction sequences, the notions of should- and must-convergence coincide (see e.g. [11, 18] for more discussion on the different notions). Thus, an alternative but equivalent definition of must-convergence is: a process $P$ is must-convergent, if all maximal reductions starting from $P$ end with a successful process.

## 2.2 The Calculus LOCKSIMPLE

We now define the calculus LOCKSIMPLE which can be seen as a modification of SYNCSIMPLE where ? and ! are removed, and operations $P_i$ and $T_i$, which mean put and take, are added where $i = 1,\ldots,k$ and $k$ is the number of locks (i.e. storage cells). Locks can be empty (written as $\square$) or full (written as $\blacksquare$). For $k$ locks, the *initial store* is a $k$-tuple $(C_1,\ldots,C_k)$ where $C_i \in \{\square,\blacksquare\}$. We make this explicit by writing $\text{LOCKSIMPLE}_{k,IS}$ for the language with $k$ locks and initial store $IS$. Subprocesses in $\text{LOCKSIMPLE}_{k,IS}$ for a fixed value $1 \leq k \in \mathbb{N}$ are built from $\checkmark, 0$, the symbols $P_i, T_i$ and concatenation. Processes are a multiset of subprocesses: they are composed by parallel composition $\mid$ which is assumed to be associative and commutative.

**Definition 2.5.** *The syntax of processes and subprocesses of the calculus* $\text{LOCKSIMPLE}_{k,IS}$ *is defined by the following grammar:*

$$\begin{array}{lllll} \textit{subprocess:} & \mathscr{U} & ::= & 0 \mid \checkmark \mid P_i\mathscr{U} \mid T_i\mathscr{U} \\ \textit{process:} & \mathscr{P} & ::= & \mathscr{U} \mid \mathscr{U} \mid \mathscr{P} \end{array}$$

We first describe the operational semantics of processes of $\text{LOCKSIMPLE}_{k,IS}$ and then give the formal definition. The operational semantics is a non-deterministic small-step reduction $\xrightarrow{LS}$ which operates on $k$ locks $C_i$ (which are full (i.e. $\blacksquare$) or empty (written as $\square$)). The execution of the operations $P_i$ or $T_i$ is as follows:

$P_i$: (put) changes $C_i$ from $\square \to \blacksquare$, or waits, if $C_i$ is $\blacksquare$.
$T_i$: (take) changes $C_i$ from $\blacksquare \to \square$, or goes on (no change), if $C_i$ is $\square$

Note that locks together with $P_i$ and $T_i$ behave like binary semaphores, where $(P_i, T_i)$ means (wait,signal) (or (down,up), resp.). The semaphore is set to 1 if the lock is empty, and set to 0 if the lock is full. Note that locks specify a particular behavior for the case of a signal operation and the semaphore set to 1: the signal has no effect (since $T_i$ on an empty lock does not have an effect). Now we formally define the operational semantics:

**Definition 2.6.** *The relation $\xrightarrow{LS}$ operates on a pair $(\mathscr{P},(C_1,\ldots,C_k))$, where $\mathscr{P}$ is a* $\text{LOCKSIMPLE}_{k,IS}$*-process, $C_1,\ldots,C_k$ are the storage cells. For a* $\text{LOCKSIMPLE}_{k,IS}$*-process $\mathscr{P}$ the reduction starts with initial store $(\mathscr{P},IS)$.*

*We write the state as $\mathscr{C}$, and with $\mathscr{C}[C_i = \square]$ we denote that the specific cell $C_i$ has value $\square$. The notation $\mathscr{C}[C_i \mapsto \square]$ means that in $\mathscr{C}$ the value in storage cell $C_i$ is replaced by $\square$. The same for $\blacksquare$ instead of $\square$. The relation $\xrightarrow{LS}$ is defined by the following two rules:*

$$(P_i\mathscr{U} \mid \mathscr{P},\mathscr{C}[C_i = \square]) \xrightarrow{LS} (\mathscr{U} \mid \mathscr{P},\mathscr{C}[C_i \mapsto \blacksquare]) \quad \textit{and} \quad (T_i\mathscr{U} \mid \mathscr{P},\mathscr{C}) \xrightarrow{LS} (\mathscr{U} \mid \mathscr{P},\mathscr{C}[C_i \mapsto \square])$$

*The reflexive-transitive closure of $\xrightarrow{LS}$ is denoted as $\xrightarrow{LS,*}$. A sequence $(\mathscr{P},\mathscr{C}) \xrightarrow{LS,*} (\mathscr{P}',\mathscr{C}')$ is called an* execution of $(\mathscr{P},\mathscr{C})$*, and if $\mathscr{C} = IS$ then it is also called an* execution of $\mathscr{P}$*.*

To simplify notation, we write LOCKSIMPLE$_k$ for the language with $k$ locks where all locks are empty at the beginning, i.e. it is LOCKSIMPLE$_{k,IS}$ with $IS = (\square, \ldots, \square)$).

Note that the blocking behavior of the put-operation is modelled by the operational semantics as follows: for $(P_i\mathscr{U} \mid \mathscr{P},\ \mathscr{C}[C_i = \blacksquare])$ there is no step (for subprocess $P_i\mathscr{U}$) defined and thus $P_i\mathscr{U}$ has to wait until another subprocess changes the value of $C_i$.

**Definition 2.7.** *A process $\mathscr{P}$ of* LOCKSIMPLE$_{k,IS}$ *is called* successful*, if there is a subprocess $\checkmark$ of $\mathscr{P}$, i.e. $\mathscr{P} = \checkmark \mid \mathscr{P}'$ for some $\mathscr{P}'$. A state $(\mathscr{P},\mathscr{C})$ is called*

- successful*, if $\mathscr{P}$ is successful.*
- may-convergent*, if there is some successful $(\mathscr{P}',\mathscr{C}')$ with $(\mathscr{P},\mathscr{C}) \xrightarrow{LS,*} (\mathscr{P}',\mathscr{C}')$.*
- must-convergent*, if for all states $(\mathscr{P}',\mathscr{C}')$ with $(\mathscr{P},\mathscr{C}) \xrightarrow{LS,*} (\mathscr{P}',\mathscr{C}')$, the state $(\mathscr{P}',\mathscr{C}')$ is may-convergent.*
- must-divergent *or a* fail*, if there is no execution leading to a successful state.*
- may-divergent*, if for some state $(\mathscr{P}',\mathscr{C}')$: $(\mathscr{P},\mathscr{C}) \xrightarrow{LS,*} (\mathscr{P}',\mathscr{C}')$, where $(\mathscr{P}',\mathscr{C}')$ is a fail.*

*A process $\mathscr{P}$ is called* may-convergent*,* must-convergent*,* must-divergent*, or* may-divergent*, resp. iff the state $(\mathscr{P},IS)$ is* may-convergent*,* must-convergent*,* must-divergent*, or* may-divergent*, resp.*

An example for a reduction sequence for $k = 2$ is:

$$(P_20 \mid T_2\checkmark, (\square,\square)) \xrightarrow{LS} (0 \mid T_2\checkmark, (\square,\blacksquare)) \xrightarrow{LS} (0 \mid \checkmark, (\square,\square)) \quad \text{(successful)}$$

The process $P_20 \mid T_2\checkmark$ is even must-convergent.

In the following, we often leave the state implicit and in abuse of notation, we "reduce" processes without explicitly mentioning the state.

As in SYNCSIMPLE we often omit the suffix, 0, for a subprocess, i.e. whenever a subprocess ends with symbol $P_i$ or $T_i$ we mean the same subprocess extended by 0.

## 2.3 Correct Translations

We are interested in translations from one full concurrent programming language with synchronous semantics into another full imperative concurrent language with locks, where the issues are expressive power and the comparison between the languages. In order to focus considerations, we investigate this issue by considering translations from a core concurrent language (SYNCSIMPLE) with synchronous semantics into a core of an imperative concurrent language (LOCKSIMPLE).

However, even in our simple languages there are interesting questions, for example, whether there exists a correct translation and how many locks are necessary for such a translation.

Since our analysis started top-down, we are sure that the non-encodability results can be transferred back to larger calculi. For discussing this, let us call the full languages SYNCFULL and LOCKFULL, resp. The language SYNCFULL may be the $\pi$-calculus and thus, it extends SYNCSIMPLE by names, named channels, name restriction, sending and receiving names and replication or recursion. The language LOCKFULL may be a variant of the core language of Concurrent Haskell, where locks are extended to synchronising memory cells which have addresses (or names) and content (for instance, numbers). The main argument why non-encodability in the small languages implies non-encodability in the larger languages is the following: Suppose we have non-encodability between the small languages for 2 locks, and there exists a correct (compositional) translation $\phi$ : SYNCFULL $\to$ LOCKFULL that uses

only one synchronising memory cell in LOCKFULL. Then the idea is to embed every SYNCSIMPLE-program $\mathscr{P}$ into a SYNCFULL-program $\mathscr{P}'$ by using only one channel, and then using the translation $\phi$ to derive a LOCKFULL-program $\phi(\mathscr{P}')$. Using this construction, we also get a translation of ! and ? into LOCKFULL, where every ! translates into a send-prefix, and every ? into a receive-prefix. The parallel-operator remains as it is. Then the correctness of $\phi$ tells us that the LOCKFULL-program $\phi(\mathscr{P}')$ has the same may- and must-convergencies. Compositionality gives us a LOCKSIMPLE-program that uses at most 2 locks, and it has the same parallel-structure as $\mathscr{P}$, and the !,?, are translated always in the same way. The result can be reduced to a LOCKSIMPLE-program with at most 2 locks, (perhaps after restricting $\phi$ w.r.t. contents of messages and recursion), which contradicts the result on small languages, since the reasoning holds for all $\mathscr{P}$.

**Definition 2.8.** *A mapping $\tau$ from the processes of* SYNCSIMPLE *into processes of* LOCKSIMPLE$_{k,IS}$ *is called a* translation*.*

- *$\tau$ is called* compositional *iff $\tau(0) = 0$, $\tau(\checkmark) = \checkmark$, $\tau(\mathscr{P}_1 \mid \mathscr{P}_2) = \tau(\mathscr{P}_1) \mid \tau(\mathscr{P}_2)$; $\tau(\mathscr{U})$ does not contain the parallel operator $\mid$ for every subprocess $\mathscr{U}$; and $\tau(!\mathscr{U}) = \tau(!)\tau(\mathscr{U})$ and $\tau(?\mathscr{U}) = \tau(?)\tau(\mathscr{U})$ for every subprocess $\mathscr{U}$*
- *$\tau$ is called* correct *iff for all* SYNCSIMPLE*-processes P, P is may-convergent iff $\tau(P)$ is may-convergent, and P is must-convergent iff $\tau(P)$ is must-convergent,*

*Compositional translations $\tau$ in our languages can be identified with the pair $(\tau(!), \tau(?))$ of strings, and we say that $\tau$ has* length *n, if $|\tau(!)| + |\tau(?)| = n$.*

For example, a correct translation cannot map $\tau(0) = \checkmark$ since then 0 is must-divergent, but $\tau(0)$ is must-convergent. Hence $\tau(0) = 0$ and $\tau(\checkmark) = \checkmark$ make sense for correct translations.

We show that three locks are sufficient for a correct compositional translation.

**Theorem 2.9.** *For $k = 3$, the translation $\tau$ with $\tau(!) = P_1T_3P_2T_1$ and $\tau(?) = P_3T_2$ is correct for initial store* $(\square, \blacksquare, \blacksquare)$.

*Proof.* We give a sketch (the full proof can be found in [23]): A communication starts with executing $P_1$ of $\tau(!) = P_1T_3P_2T_1$, leaving the storage $(\blacksquare, \blacksquare, \blacksquare)$. Then no other sequence $\tau(!), \tau(?)$ in parallel processes can be executed. Then $T_3$ is executed, leaving the storage $(\blacksquare, \blacksquare, \square)$. The next step is that one process with $\tau(?) = P_3T_2$ may start, and $P_3$ is executed, leaving the storage $(\blacksquare, \blacksquare, \blacksquare)$. Now $T_2$ is executed, and this is the only possibility. the storage is then $(\blacksquare, \square, \blacksquare)$. Again, the only possibility is now $P_2$ from $\tau(!)$ and the storage $(\blacksquare, \blacksquare, \blacksquare)$. The last step is executing $T_1$, which restores the initial storage $(\square, \blacksquare, \blacksquare)$.

This is the only execution possibility of $\tau(!)$ and $\tau(?)$, hence it can be retranslated into an interaction communication of a single ! and a single ?. □

There are also other correct compositional translations for $k = 3$: An example is a compositional correct translation $\tau$ of length 8, detected by an automated search, with $\tau(!) = P_2P_1T_3P_1T_1T_2$ and $\tau(?) = P_3T_1$ and with initial store $(\square, \square, \blacksquare)$.

The observation is that the communication is completely protected by using $P_2$ as a mutex, which is similar to the translation of length 6 (see Theorem 2.9)

## 2.4 Blocking Variants of LOCKSIMPLE

We choose for our locks, that $P_i$ blocks, but $T_i$ never blocks. However, also other choices are possible. Variants of LOCKSIMPLE where for every $i$ either $P_i$ blocks on a full lock, but $T_i$ is non-blocking, or $T_i$ blocks on an empty lock, but $P_i$ is non-blocking, do not lead to really new problems: In [23] we show that

all those variants are equivalent to the previously defined language where for all $i$: $P_i$ is blocking, but $T_i$ is non-blocking. This is possible since we take into account any initial store and thus the main argument of the equivalence is that we can change the initial store for every $i$ by switching the role of $P_i, T_i$ and at the same time switching the initial store for $i$ from ■ to □ and vice versa. Thus this extension does not increase the number of (really) different languages for a fixed $k$. However, the variant where $P_i$ blocks for a full lock *and* $T_i$ blocks for an empty lock for all $i$ (which is related to an implementation using the MVars in Concurrent Haskell) appears to be different from our LOCKSIMPLE languages. There are results on possibility and impossibility of correct translations from SYNCSIMPLE into a further restricted variant of LOCKSIMPLE [20]. A deeper investigation in these languages is future work.

## 3 One Lock is Insufficient for any Initialization

We show that there is no correct (compositional) translation into $\text{LOCKSIMPLE}_{1,IS}$, the language with one lock, for any initial storage, i.e. for initial storage ■ and initial storage □.

**Lemma 3.1.** *Let $\tau$ be a correct translation* $\text{SYNCSIMPLE} \to \text{LOCKSIMPLE}_{1,IS}$. *Then $\tau(!)$ as well as $\tau(?)$ either start with $P_1$ or have a subsequence $P_1P_1$.*

*Proof.* Consider the processes $!\checkmark$ and $?\checkmark$ which are both must-divergent. If $\tau(!)$ does not satisfy the condition, then the process $\tau(!\checkmark)$ can be executed without any wait and is successful. The same for $\tau(?\checkmark)$. However, this is a contradiction to correctness. □

**Theorem 3.2.** *There is no correct translation* $\text{SYNCSIMPLE} \to \text{LOCKSIMPLE}_{1,IS}$.

*Proof.* Let $\tau$ be a correct translation. We first consider the case that the initial storage is □. Then from Lemma 3.1 we derive that $\tau(!)$ as well as $\tau(?)$ have a subsequence $P_1P_1$ or start with $P_1$. since $P_1$ as a prefix is executable (and similar as in the proof of Lemma 3.1, the processes $!\checkmark$ and $?\checkmark$ can be used as examples to refute the correctness of $\tau$). Consider the process $\tau(!\checkmark \mid ?\checkmark)$, which is must-convergent. First, reduce $\tau(!\checkmark)$ until exactly before the first occurrence of $P_1P_1$. Then reduce $\tau(?\checkmark)$. Since the reduction starts with $C_1 = \square$, it will block after executing the first $P_1$ of the leftmost subsequence $P_1P_1$ (or earlier). Then $C_1 = \blacksquare$, and we have a deadlock. This is a contradiction to correctness of $\tau$.

Now we consider the case that the initial store is ■. Then Lemma 3.1 shows that $\tau(!)$ and $\tau(?)$ contain a subsequence $P_1P_1$ or start with $P_1$. We again use the must-convergent process $\tau(!\checkmark \mid ?\checkmark)$. If both $\tau(!)$ and $\tau(?)$ start with $P_1$, then there is an initial deadlock. Suppose that neither $\tau(!)$ nor $\tau(?)$ do start with a $P_1$, then they both start with a $T_1$, and have a subsequence $P_1P_1$. Let us consider the leftmost such subsequence for $\tau(!)$ as well as for $\tau(?)$. Construct the following execution for $\tau(!\checkmark \mid ?\checkmark)$: First $\tau(!)$ until it blocks at the second $P_1$ of the sequence $P_1P_1$, then the execution of $\tau(?)$ until the second $P_1$ of the sequence $P_1P_1$. Then we have a deadlock, which is impossible.

If $\tau(!)$ starts with a $P_1$, but not $\tau(?)$, then there is a leftmost sequence $P_1P_1$ of $\tau(?)$. Execute $\tau(?)$ until it is blocked at $P_1$. Then we reach a deadlock. This is a contradiction. □

## 4 General Properties for at Least Two Locks

In this section, we consider compositional translations $\text{SYNCSIMPLE} \to \text{LOCKSIMPLE}_{k,IS}$ with $k \geq 2$ and prove several properties of correct compositional translations that will help us later to show that $k = 2$ is impossible. We also introduce the notion of a blocking type for a translation. The idea of this

notion is recording how $\tau$ establishes that executing $\tau(!)$ in the process $\tau(!\checkmark)$ blocks and why executing $\tau(?)$ in the process $\tau(?\checkmark)$ blocks. Both processes must block if $\tau$ is correct, since the the processes $!\checkmark$ and $?\checkmark$ are both blocking (and not successful) in SYNCSIMPLE.

Below this notion helps to structure the arguments for different cases.

**Lemma 4.1.** *Let $\tau$ be a correct translation from* SYNCSIMPLE $\to$ LOCKSIMPLE$_{k,IS}$ *for $k \geq 1$. Then there is a reduction sequence of $\tau(!) \mid \tau(?)$ that executes every symbol in $\tau(!) \mid \tau(?)$.*

*Proof.* First, consider $\tau(!\checkmark) \mid \tau(?0)$, which is must-convergent (since $!\checkmark \mid ?0$ is must-convergent), and hence there is a reduction sequence of $\tau(!) \mid \tau(?)$ consuming at least all symbols in $\tau(!)$. The same sequence can be used as a partial reduction sequence of $\tau(!0) \mid \tau(?\checkmark)$, and since this process is must-convergent (since $!0 \mid ?\checkmark$ is must-convergent), the sequence will also consume all symbols of $\tau(?\checkmark)$. □

The notation $\#(S,r)$ means the number of occurrences of the symbol $S$ in the string $r$.

**Proposition 4.2.** *Let $\tau$ :* SYNCSIMPLE $\to$ LOCKSIMPLE$_{k,IS}$ *for $k \geq 2$ be a correct translation. Then for every $1 \leq i \leq k$: $\#(P_i,\tau(!)) + \#(P_i,\tau(?)) \leq \#(T_i,\tau(!)) + \#(T_i,\tau(?))$.*

*Proof.* The processes $!!\checkmark \mid ??\checkmark$, $!!0 \mid ??\checkmark$ and $!!\checkmark \mid ??0$ are must-convergent, hence also their images under $\tau$. Now suppose the claim is false. Then for some index, say 1, $\#(P_1,\tau(!)) + \#(P_1,\tau(?)) > \#(T_1,\tau(!)) + \#(T_1,\tau(?))$. We apply Lemma 4.1 to $\tau(!!\checkmark \mid ??\checkmark)$ and obtain a reduction sequence $R_1$ that exactly consumes the top parts $\tau(!)$ and $\tau(?)$ of $\tau(!!\checkmark \mid ??\checkmark)$.

Replacing $\checkmark$ by 0, the reduction sequence $R_1$ can be also used for $\tau(!!\checkmark \mid ??0)$. Since $\tau(!!\checkmark \mid ??0)$ is must-convergent, $R_1$ can be continued to $R_1R_2$ ending in a success of the form $\checkmark \mid Q0$ where $Q$ is a suffix of $\tau(?)$, since $!!\checkmark \mid ??0$ is must-convergent.

The reduction sequence $R_1R_2$ can also be used for $\tau(!!0 \mid ??\checkmark)$ (by interchanging 0 and $\checkmark$), ending in $0 \mid Q\checkmark$. Since $!!0 \mid ??\checkmark$ is must-convergent, the reduction sequence $R_1R_2$ can be extended to $R_1R_2R_3$ resulting in $0 \mid \checkmark$.

After $R_1$, we have $C_1 = \blacksquare$ and that the initial store for index 1 is $\square$, due to the assumption, and since the symbols in $\tau(!),\tau(?)$ are completely consumed. Hence $R_2R_3$ must execute a $T_1$ before every other $P_1$. But since the number of $T_1$-symbols is strictly smaller than the number of $P_1$-symbols, there must be a deadlock situation at least for one of the symbols $P_1$.

This is a contradiction, hence the proposition holds. □

**Definition 4.3.** *For a correct translation $\tau$ into* LOCKSIMPLE$_{k,IS}$*, a* blocking prefix *of a sequence $S$ of symbols in* LOCKSIMPLE$_{k,IS}$ *is a prefix of $S$ of one of the two forms:*

1. *$R_1P_iR_2P_i$, where $R_1,R_2$ are sequences, and $R_2$ does not contain $P_i,T_i$, and the execution of $S$ that starts with store IS deadlocks exactly before the last symbol, which is $P_i$.*
2. *$R_1P_i$, where $R_1$ does not contain $P_i,T_i$, and the execution of $S$ that starts with store IS deadlocks exactly before the last symbol, which is $P_i$.*

*We may also speak of $R_1P_i$ or $P_iR_2P_i$, respectively, as a* blocking subsequence *of $S$.*
*In the case that $S$ has a blocking sequence, we say that the* blocking type *of $S$ is $P_iP_i$ if the blocking sequence is $R_1P_iR_2P_i$, and the blocking type is $P_i$, if the blocking sequence is $R_1P_i$.*

*We say a translation $\tau$ has* blocking type $(W_1,W_2)$*, if $W_1$ is the blocking type of $\tau(!)$, and $W_2$ is the blocking type of $\tau(?)$.*

**Lemma 4.4.** *Let $\tau$ :* SYNCSIMPLE $\to$ LOCKSIMPLE$_{k,IS}$ *be a correct translation where $k \geq 2$. Then there is some $i$, such that $\tau(!)$ has a blocking subsequence of the form $RP_i$, or $P_iRP_i$, where $R$ does not contain $P_i,T_i$. The same holds for $\tau(?)$.*

*Proof.* The reduction of $\tau(!)$ cannot be completely executed, since $\tau(!\checkmark)$ is a fail. Hence the execution stops at a symbol $P_i$, and it is either the first occurrence of $P_i$, or a later occurrence. Hence the sequence before is of the form $R$, or $P_iR$, where $R$ does not contain $P_i, T_i$. The same arguments hold for $\tau(?)$. □

**Lemma 4.5.** *Let $\tau$ : SYNCSIMPLE $\to$ LOCKSIMPLE$_{k,IS}$ be a correct translation where $k \geq 2$. If $\tau(!)$ is of blocking type $P_i$ then $IS_i = \blacksquare$, and if $\tau(!)$ is of blocking type $P_iP_i$ then the first $i$-symbol is $T_i$, or $IS_i = \square$; The same holds for $\tau(?)$.*

*Proof.* The blocking type $P_i$ is only possible if in $R$ of the prefix $RP_i$ there is no $T_i$, hence the initial store $IS_i = \blacksquare$. If the blocking type is $P_iP_i$ and $IS_i = \blacksquare$, then the first $i$-symbol must be a $T_i$. The other case is that $IS_i$ is $\square$. □

# 5 Non-Existence of a Correct Translation for Two Locks

In this section, we will show that there is no correct compositional translation from SYNCSIMPLE to LOCKSIMPLE$_{2,IS}$ (for any initial storage $IS$). We distinguish several cases by considering different blocking types according to Definition 4.3. When reasoning on translations, we use an extended notation of translations as pairs of strings (i.e. $(\tau(!), \tau(?))$): We describe sets of translations using set-concatenation (writing singletons without curly braces) and the Kleene-star. For instance, we write $(\{P_1, T_1\}^* T_2, \{P_2\}^* T_1)$ to denote the set of all translations where $\tau(!)$ starts with arbitrary many $P_1$- and $T_1$-steps ending with $T_2$, and $\tau(?)$ starting with an arbitrary number of $P_2$-steps followed by a single $T_1$-step.

An automated search for compositional translations for $k = 2$ and length $\leq 10$ has refuted the correctness of all these translations for all initializations of the initial storage. This is consistent with our general arguments in this section.

## 5.1 Refuting the Blocking Type $(P_iP_i, P_jP_j)$

**Proposition 5.1.** *Let $\tau$ : SYNCSIMPLE $\to$ LOCKSIMPLE$_{2,IS}$ be a correct translation of blocking type $(P_iP_i, P_jP_j)$. Then $i \neq j$.*

*Proof.* W.l.o.g. assume that the blocking type is $(P_1P_1, P_1P_1)$. Then the blocking prefixes of $\tau(!)$ and $\tau(?)$ are $M_1P_1RP_1$ and $M_2P_1R'P_1$, respectively. Now we reduce the must-convergent process $\tau(!0) \mid \tau(?\checkmark)$ by selecting the following reduction sequence: first, reduce $\tau(!)$ until $M_1P_1R$ is completely executed, and then reduce $\tau(?)$ as far as possible. Let $Q$ be the prefix of $\tau(?)$ of the form $\{P_2, T_2\}^*\{P_1, T_1\}$. If $P_1$ is the symbol from $\{P_1, T_1\}$ of $\tau(?)$, then a deadlock would occur, which is not possible, since $!0 \mid ?\checkmark$ is must-convergent. Hence $Q$ as a prefix of $\tau(?)$ must be of the form $\{P_2, T_2\}^* T_1$. There are two cases:

1. After executing $M_1P_1R$ it holds $C_1 = \blacksquare$ and $C_2 = \square$.
   (a) $IS_2 = \square$. Now, since reducing $\tau(?)$ starts with $C_2 = \square$, and the final $T_1$ of $Q$ resets $C_1$, the reduction sequence starting with $\tau(!0)$ and then executing $\tau(?)$ is possible until the end of $M_2P_1R'$. Since now $C_1 = \blacksquare$ and in both pending subprocesses a $P_1$ is to be executed, we have a deadlock, which is impossible due to must-convergence of $!0 \mid ?\checkmark$.
   (b) $IS_2 = \blacksquare$. Then the first $\{P_2, T_2\}$-symbol of $\tau(?)$ cannot be $P_2$, since it would block. Hence the first $\{P_2, T_2\}$-symbol of $\tau(?)$ is $T_2$. Then the further reduction of $\tau(?)$ is independent of the initial values and it is the same as in the previous case.

2. After executing $M_1P_1R$ it holds $C_1 = \blacksquare$ and $C_2 = \blacksquare$. Then the first symbol of $\tau(?)$ cannot be $P_2$, since this would be a deadlock. Also, the first symbol of $\tau(?)$ cannot be $T_2$, since then the reduction of $\tau(?)$ alone is the same as started with the initialization $C_1 = C_2 = \square$, and the reduction proceeds until the end of the blocking sequence, which leads to a deadlock. Hence $\tau(?)$ starts with $T_1$. The prefix of $\tau(?)$ cannot be $T_1\{T_1,P_1\}^*P_2$, since this either blocks within $T_1\{T_1,P_1\}^*$ or at $P_2$. Hence the prefix is $T_1\{T_1,P_1\}^*T_2$. This implies that $\tau(?)$ is executable until the blocking $P_1$, and thus leads to a deadlock. Hence this case is also not possible.

We have checked all cases, hence $i = j$ is not possible and the lemma is proved. □

We consider the blocking type $(P_1P_1, P_2P_2)$ in the rest of this subsection, which suffices due to symmetry and Proposition 5.1.

**Lemma 5.2.** *Let $\tau$ : SYNCSIMPLE $\to$ LOCKSIMPLE$_{2,IS}$ be a correct translation of blocking type $(P_1P_1, P_2P_2)$. Then the following holds:*

1. *The blocking prefix of $\tau(!)$ is $R_1P_1\{P_2,T_2\}^*T_2P_1$ and the blocking prefix of $\tau(?)$ is $R_3P_2\{P_1,T_1\}^*T_1P_2$.*
2. *$\{T_1,P_1\}^*T_2$ is a prefix of $\tau(!)$, and $\{T_2,P_2\}^*T_1$ is a prefix of $\tau(?)$.*

*Proof.* Let the blocking prefix of $\tau(!)$ be $R_1P_1P_1$ and the blocking prefix of $\tau(?)$ be $R_3P_2\{T_1,P_1\}^*P_2$. Then first execute $R_1$, and then $R_3P_2\{T_1,P_1\}^*$ until it blocks. If it blocks at a $P_1$, then it is a deadlock. If it blocks at a $P_2$, then $P_1P_1$ cannot be both executed, hence a deadlock. Hence $\tau(!)$ has a blocking prefix $R_1P_1R_2P_1$ where $R_2 \neq \emptyset$. By symmetry, we obtain that the blocking prefix of $\tau(?)$ is $R_3P_2R_4P_2$ where $R_4 \neq \emptyset$. Now let the blocking prefix of $\tau(!)$ be $R_1P_1\{T_2,P_2\}^*P_2P_1$. Execute $\tau(!)$ until $P_2P_1$ is left, and then execute $\tau(?)$. Clearly, $\tau(?)$ must block, independent of the previous executions. If $\tau(?)$ blocks at $P_1$, then we have a deadlock, and if it blocks at $P_2$, then we also have a deadlock. Hence the blocking prefix of $\tau(!)$ is of the form $R_1P_1\{T_2,P_2\}^*T_2P_1$.

By symmetry, we obtain that the blocking prefix of $\tau(?)$ is of the form $R_3P_2\{T_1,P_1\}^*T_1P_2$. Now we prove restrictions on the prefix of $\tau(!)$ and $\tau(?)$. Assume that the prefix of $\tau(?)$ is $\{T_2,P_2\}^*P_1$. Then first reduce $\tau(!)$ until it blocks before $P_1$, then reduce $\tau(?)$, until it blocks within $\{T_2,P_2\}^*$ or at the (first) $P_1$ in $\tau(?)$. Both cases lead to a deadlock, hence this case is impossible. Thus $\tau(?)$ has prefix $\{T_2,P_2\}^*T_1$. □

For the rest of this subsection, we assume blocking type $(P_1P_1, P_2P_2)$, and that only correct translations are of interest.

**Lemma 5.3.** *Let $\tau$ be a correct translation. Then for any initial storage the prefix of $\tau(!)$ cannot be $T_1^+T_2$ nor $T_2^+T_1$.*

*Proof.* In each case the must-divergent process $\tau(!\checkmark \mid \ldots \mid !\checkmark)$ with sufficiently many subprocesses can be reduced such that it leads to a success, which contradicts the correctness of $\tau$: Fix the first subprocess and reduce it until the end using the prefixes of the other subprocesses to proceed in case of a blocking. This leads to success, which is a contradiction. □

Lemma 5.2 implies:

**Lemma 5.4.** *The prefix of $\tau(!)$ cannot be $T_1^*P_2$.*

**Lemma 5.5.** *Let $\tau$ be a correct translation. Then the prefix of $\tau(!)$ cannot be $T_2^+P_2$.*

*Proof.* Consider the must-convergent process $\tau(!\checkmark \mid \ldots \mid !\checkmark \mid ?0)$. First reduce all the prefixes $T_2^+$ in all $\tau(!\checkmark)$ until $P_2$ is the first symbol. Since $\{T_2,P_2\}^*T_1$ is a prefix of $\tau(?)$, and due to the assumption of the blocking type, reduction cannot block at a $P_2$ in $\{T_2,P_2\}^*$. Hence $T_1$ is executed, which means that reduction is now independent of the initial store. We reduce $\tau(?)$ until it stops before the second $P_2$ of the blocking subsequence. Then it is a deadlock, which contradicts correctness of $\tau$. □

**Lemma 5.6.** *The prefix of $\tau(!)$ cannot be $P_1$.*

*Proof.* Assume the prefix of $\tau(!)$ is $P_1$. Then $IS_1 = \square$ due to the assumption that the blocking type is $(P_1P_1,P_2P_2)$. Consider the must-convergent process $!\checkmark \mid \ldots \mid !\checkmark \mid ?0$, where we fix the number of $!\checkmark$-subprocesses later if this is necessary. We will use the structure of the subprocesses $\tau(!)$ and $\tau(?)$ proved in Lemma 5.2 whenever necessary.

1. Reduce $\tau(?0)$ before it stops at the second $P_2$ of the blocking subsequence. After this we have $C_1 = \square, C_2 = \blacksquare$.
2. Reduce one subprocess $\tau(!\checkmark)$ until it blocks. Since $C_1 = IS_1 = \square$ at the start and $\{P_1,T_1\}^*T_2$ is a prefix of $\tau(!)$, the reduction is the same as started with $IS$, hence it stops at the second $P_1$ of the blocking subsequence and so $C_1 = \blacksquare, C_2 = \square$ at the end.
3. We go on with the reduction of $\tau(?)$ until it blocks. It cannot block at a $P_1$, since this would be a deadlock. If the reduction consumes all of $\tau(?)$, then we reduce the next $\tau(!)$: The prefix $\{T_1,P_1\}^*T_2$ shows that it cannot block at $P_1$ of $\{T_1,P_1\}^*$, since this would be a deadlock, hence $T_2$ is executed, Now it cannot block at a $P_2$ before the end of the blocking sequence. Thus reduction will lead to a deadlock at the end of the blocking sequence, since all remaining subprocesses start with a $P_1$.

   The last case is that the further reduction of $\tau(?0)$ blocks at a $P_2$. Then again we reduce the next subprocess $\tau(!\checkmark)$. It cannot block at $P_1$ of the prefix $\{T_1,P_1\}^*T_2$, since this would be a deadlock, hence it executes a $T_2$, and thus again it blocks at a $P_1$ at the end of a blocking sequence. This is the final deadlock. □

**Lemma 5.7.** *Let $\tau$ be a correct translation. Then the prefix of $\tau(!)$ cannot be $T_2^+P_1$.*

*Proof.* Consider the must-convergent process $\tau(!\checkmark \mid \ldots \mid !\checkmark \mid ?0)$. First, reduce all $T_2^+$-prefixes away, thhen use the same arguments as in Lemma 5.6, which is possible, since it is the same process. □

Since $P_1$ as prefix of $\tau(!)$ is already excluded, we show the following.

**Lemma 5.8.** *Let $\tau$ be a correct translation. Then the prefix of $\tau(!)$ cannot be $T_1^+P_1$.*

*Proof.* Let us assume that the prefix of $\tau(!)$ is $T_1^+P_1$. We know that it is also $\{P_1,T_1\}^*T_2$. Consider the must-convergent process $\tau(!\checkmark \mid \ldots \mid !\checkmark \mid ?0)$. Reduce $\tau(?)$ until it stops before the second $P_2$ of the blocking subsequence with $C_1 = \square, C_2 = \blacksquare$. There are two cases:

1. $\tau(!\checkmark)$ can be reduced until it blocks at a $P_2$. Then we assume that the process is $\tau(!\checkmark \mid ?0)$ Hence we have a deadlock.
2. $\tau(!\checkmark)$ can be reduced until it blocks at a $P_1$. This position must be the second position in a blocking subsequence, since reduction starts with $C_1 = \square$, and the prefix $\{P_1,T_1\}^*T_2$ enforces that a $T_2$ is executed before any $P_2$ in $\tau(!)$. Due to the form of the blocking sequence the last step before blocking was a $T_2$. We continue now the reduction of $\tau(?)$. This can block at a $P_2$, and we will

again use a $\tau(!)$-subprocess for unblocking. Or it stops at a $P_1$, then we use the $T_1^+$ at the start of a fresh $\tau(!)$ to unblock. Finally, $\tau(?)$ is worked-off. The already used $\tau(!)$ now remain with a prefix $P_1$. We execute the remaining $\tau(!)$ until the blocking $P_1$.

All cases lead to a deadlock, which is a contradiction to correctness of $\tau$. □

**Proposition 5.9.** *Blocking type $(P_iP_i,P_jP_j)$ is impossible for a correct translation for $k = 2$.*

*Proof.* Proposition 5.1 excludes the case $i = j$. For the case $i \neq j$, it is sufficient to consider $i = 1$, $j = 2$ (due to symmetry). Assume that $\tau$ is a correct translation of blocking type $(P_1P_1,P_2P_2)$. Lemma 5.2 shows that $\{T_1,P_1\}^*T_2$ and $\{T_1,T_2,P_1,P_2\}^*P_1\{P_2,T_2\}^*T_2P_1$ must be prefixes of $\tau(!)$. Thus $\tau(!)$ must start with $T_1,P_1$ or $T_2$ and the length of $\tau(!)$ is at least 3. Lemma 5.6 shows that $\tau(!)$ cannot start with $P_1$. Lemmas 5.3, 5.4 and 5.8 show that the prefix of $\tau(!)$ cannot be $T_1^+T_2$, $T_1^+P_1$, nor $T_1^*P_2$. Thus $\tau(!)$ cannot start with $T_1$. Lemmas 5.3, 5.5 and 5.7 show that the prefix of $\tau(!)$ cannot be $T_2^+P_2,T_2^+T_1$, nor $T_2^+P_1$. Thus $\tau(!)$ cannot start with $T_2$. Hence, we have a contradiction, and $\tau$ cannot be correct. □

## 5.2 Refuting Blocking Types $(P_iP_i,P_i)$, $(P_i,P_iP_i)$, $(P_i,P_i)$, $(P_iP_i,P_j)$

**Proposition 5.10.** *Let $\tau$ be a correct translation. For $k = 2$ the blocking types $(P_1P_1,P_1)$, $(P_1,P_1P_1)$, and $(P_1,P_1)$ are not possible.*

*Proof.* First, we assume $(P_1P_1,P_1)$. Consider the process $\tau(!\checkmark) \mid \tau(?\checkmark)$ which must be must-convergent for a correct translation $\tau$. The blocking prefix of $\tau(?)$ is of the form $\{P_2,T_2\}^*P_1$, and $IS_1 = \blacksquare$. Then construct the following reduction: first, reduce $\tau(!\checkmark)$ until the blocking $P_1$ (now $C_1 = \blacksquare$ still holds), and then the prefix $\{P_2,T_2\}^*P_1$ of $\tau(?\checkmark)$. If it blocks at some $P_2$, then it is a deadlock, and if it blocks at the $P_1$, it is also a deadlock. The symmetric type $(P_1,P_1P_1)$ is also impossible (by the symmetric reduction). Now assume the type is $(P_1,P_1)$. Then the blocking prefixes of $\tau(!)$ and $\tau(?)$ are both of the form $\{P_2,T_2\}^*P_1$. Reducing $\tau(!)$ blocks at $P_1$. Afterwards reducing $\tau(?)$ either stops at a $P_2$, which is a deadlock, or at $P_1$, which is also a deadlock. Thus for the must-convergent process $(! \mid ?\checkmark)$ we can construct a reduction sequence for $\tau(! \mid ?\checkmark)$ that ends in a deadlock. □

In the following, we only have to think about the blocking types $(P_1P_1,P_2)$, and $(P_1,P_2)$, since $(P_2,P_1P_1)$ is a symmetric case of the first one.

**Lemma 5.11.** *Blocking type $(P_1P_1,P_2)$ is not possible for a correct translation and $k = 2$.*

*Proof.* Assume that the blocking type of $\tau$ is $(P_1P_1,P_2)$. Lemma 4.5 shows that $IS_2 = \blacksquare$, and the prefix of $\tau(?)$ is $\{P_1,T_1\}^*P_2$. This holds, since if the first symbol in $\tau(?)$ which is in $\{P_2,T_2\}^*$ is $T_2$, then the blocking type would be different for $\tau(?)$.

Since the blocking type of $\tau(!)$ is $P_1P_1$, Lemma 4.5 shows that either $IS_1 = \square$ or the first 1-symbol in the blocking-sequence (which is of the form $R_1P_1\{T_2,P_2\}^*P_1$) is $T_1$.

The blocking prefix of $\tau(!)$ cannot be $\{P_1,T_1\}^*$: This would imply that it stops with $P_1P_1$. Then the process $\tau(!\checkmark \mid ?)$ permits a failing reduction: First, reduce $\tau(?)$ until it blocks with $P_2$, and then reduce $\tau(!)$, which blocks at $P_1$ without changing $C_2$, hence it is a deadlock.

A prefix of $\tau(!)$ is of the form $\{P_1,T_1\}^*T_2$: Suppose the prefix is $\{P_1,T_1\}^*P_2$. Reducing $\tau(!\checkmark \mid ?)$ as follows: First $\tau(!)$, which cannot block within the prefix $\{P_1,T_1\}^*$, hence it blocks at $P_2$. Subsequent reduction of $\tau(?)$ leads to a deadlock since it blocks at $P_2$.

For the final contradiction, we show that the process $\tau(!\checkmark \mid ?)$ permits a failing reduction: First, reduce $\tau(?)$ until it blocks with $P_2$, and then reduce $\tau(!)$, which blocks at $P_1$. If $C_2 = \blacksquare$ after the reduction,

then it is a deadlock. Hence $C_2 = \square$ after the reduction. This holds for every reduction of $\tau(!)$ until blocking. Now we restart with the process $\tau(!\checkmark \mid \ldots \mid !\checkmark \mid ?)$, where we will fix the number of $!\checkmark$-subprocesses later. First, reduce $\tau(!)$ until the blocking $P_1$ and get $C_2 = \square$. Then we reduce $\tau(?)$ as far as possible. There are cases:

1. $\tau(?)$ can be completely reduced. Then we reduce the second $\tau(!)$ until a blocking, which will occur at $P_1$. Then $C_1 = \blacksquare$, and hence both $\tau(!)$ are blocked forever.
2. $\tau(?)$ blocks at a $P_1$, then we have a deadlock.
3. $\tau(?)$ blocks at a later $P_2$. Then again we use the next subprocess $\tau(!)$ and reduce it to the blocking $P_1$, with $C_2 = \square$, and can proceed with $\tau(?)$. This can be repeated until $\tau(?)$ is completely reduced, where we assume sufficiently many subprocesses $\tau(!\checkmark)$. Finally we get a deadlock by reducing the last $\tau(!)$ to the blocking, and then we have a deadlock. □

## 5.3 Refuting the Blocking Type $(P_1, P_2)$

The treatment of blocking type $(P_1, P_2)$ requires more arguments. We first show a lemma on the suffix of $\tau(!)$ and $\tau(?)$, that permit to reuse results for other initial stores than $(\blacksquare, \blacksquare)$.

**Lemma 5.12.** *For $k = 2$ and a correct translation $\tau$ of blocking type $(P_1, P_2)$, the initial store can only be $(\blacksquare, \blacksquare)$ and the prefixes of $\tau(!)$ and $\tau(?)$ are $\{P_2, T_2\}^* P_1$, and $\{P_1, T_1\}^* P_2$.*

Due to space constraints the proof of the following proposition is given in [23]:

**Proposition 5.13.** *Let $\tau$ be a translation for $k = 2$ of blocking type $(P_1, P_2)$. Let $\tau(!)$ consist of a sequence of building blocks which follow the pattern $\{T_1, T_2\}^* P_1$ or $\{T_1, T_2\}^* P_2$, where in addition a suffix $\{T_1, T_2\}^*$ is appended. Let $\tau(?)$ consist of a sequence of building blocks which follow the pattern $\{T_1, T_2\}^* P_1$ or $\{T_1, T_2\}^* P_2$. Then $\tau$ is not correct.*

**Corollary 5.14.** *Let $\tau$ be a correct translation for $k = 2$ of blocking type $(P_1, P_2)$. Then $\tau(!)$ and $\tau(?)$ have a nontrivial suffix in $\{T_1, T_2\}^+$.*

Extending a must-convergent process by $! \mid ?$ may destroy the must-convergence. An example is $!?0 \mid ?\checkmark$, where $! \mid ? \mid !?0 \mid ?\checkmark$ becomes may-divergent. However, for flat processes, the extension preserves must-convergence, where a SYNCSIMPLE-process is *flat* if it is of the form $A_1 \mid \ldots \mid A_n$, where $A_i$ is $!0, ?0, !\checkmark$, or $?\checkmark$.

**Lemma 5.15.** *Let $Q$ be a flat* SYNCSIMPLE*-process that is must-convergent. Then the process $! \mid ? \mid Q$ is also must-convergent.*

**Proposition 5.16.** *Blocking type $(P_1, P_2)$ is impossible for correct translations for $k = 2$.*

*Proof.* Assume that $\tau$ is correct for initial state $(\blacksquare,\blacksquare)$. Then Corollary 5.14 shows that $\tau(!)$ and $\tau(?)$ must end with $\{T_1, T_2\}^+$. Since $\tau(! \mid ?)$ must be completely executable (see Lemma 4.1), reducing $\tau((! \mid ? \mid Q), (\blacksquare, \blacksquare)) \xrightarrow{LS,*} (\tau(Q), (k_1, k_2))$ must lead to a state $(k_1, k_2) \neq (\blacksquare, \blacksquare)$ for every $Q$. We consider the blocking behavior of $\tau$ for $(k_1, k_2) \neq (\blacksquare, \blacksquare)$.

- If $\tau(?)$ is non-blocking for $(k_1, k_2)$, then consider the must-divergent process $!?\checkmark \mid ?$. Then $(\tau(!?\checkmark \mid ?), (\blacksquare, \blacksquare)) \xrightarrow{LS,*} (\tau(?)\checkmark, (k_1, k_2)) \xrightarrow{LS,*} (\checkmark, (l_1, l_2))$. Thus $\tau$ is not correct.
- If $\tau(!)$ is non-blocking for $(k_1, k_2)$, then consider the must-divergent process $?!\checkmark \mid !$. Then $(\tau(?!\checkmark \mid !), (\blacksquare, \blacksquare)) \xrightarrow{LS,*} (\tau(!)\checkmark, (k_1, k_2)) \xrightarrow{LS,*} (\checkmark, (l_1, l_2))$. Thus $\tau$ is not correct.

- We know that the prefix of $\tau(!)$ cannot be $T_1^+ T_2$ nor $T_2^+ T_1$ (see Lemma 5.3).
- The blocking type of $\tau$ for $(k_1,k_2)$ is $(P_iP_i, P_jP_j)$. Then the proof of Proposition 5.1 can be adapted to first show that $i \neq j$: It uses flat must-convergent processes and constructs failing reductions. Let $Q$ be such a counter-example process Lemma 5.15 shows that $! \mid ? \mid Q$ is also must-convergent, and thus $\tau(! \mid ? \mid Q, (\blacksquare,\blacksquare)) \xrightarrow{LS,*} (\tau(Q),(k_1,k_2))$ and thus $(\tau(Q),(k_1,k_2))$ also must be must-convergent. But the constructed failing reductions of Proposition 5.1 refute this. For the case $i \neq j$, we can reason as in the lemmas before Proposition 5.9 and also as in Proposition 5.9 itself, since they all use flat must-convergent SYNCSIMPLE-processes and show that there are failing reductions after translating them. Again if $Q$ is such a process, Lemma 5.15 shows that $! \mid ? \mid Q$ is also must-convergent, and thus $\tau(! \mid ? \mid Q, (\blacksquare,\blacksquare)) \xrightarrow{LS,*} (\tau(Q),(k_1,k_2))$. Thus $(\tau(Q),(k_1,k_2))$ must be must-convergent. But the constructed failing reductions in the proofs in the lemmas before Proposition 5.9, or in the proof of Proposition 5.9, respectively, refute the must-convergence. Thus the proved properties also hold if $\tau$ is of blocking type $(P_iP_i, P_jP_j)$ for $(k_1,k_2)$ (where Lemma 5.3 can be used directly, since it holds for any initial state). This shows $(P_iP_i, P_jP_j)$ is impossible as blocking type of $\tau$ for $(k_1,k_2)$.
- The blocking type of $\tau$ for $(k_1,k_2)$ is $(P_1P_1, P_1)$ or $(P_1, P_1P_1)$ or $(P_1,P_1)$. Then the must-convergent SYNCSIMPLE-processes in the proof of Proposition 5.10 can be used, since they are flat. Let $Q$ be such a process. By Lemma 5.15 $! \mid ? \mid Q$ is must-convergent. Since $\tau$ is correct $\tau(! \mid ? \mid Q)$ is must-convergent and thus $(\tau(Q),(k_1,k_2))$ is must-convergent. The proof of Proposition 5.10 shows that $(\tau(Q),(k_1,k_2))$ may-diverges, a contradiction.
- $\tau$ is of blocking type $(P_1P_1, P_2)$ for $(k_1,k_2)$. Then the reasoning is analogous to the previous case using the must-convergent flat counterexample processes of Lemma 5.11.
- The blocking type $(P_1,P_2)$ is not possible, since we have a store $(k_1,k_2) \neq (\blacksquare,\blacksquare)$. □

We now prove the main result:

**Theorem 5.17.** *Let IS be an initial store with two elements, and* $\tau$ : SYNCSIMPLE $\rightarrow$ LOCKSIMPLE$_{2,IS}$ *be a compositional translation. Then* $\tau$ *is not correct.*

*Proof.* The proof is structured along the blocking types (Definition 4.3) of translations. For $k = 2$ there are 4 blocking types of subprocesses, and 16 potentially possible blocking types of translations. Proposition 5.1 shows that type $(P_iP_i, P_iP_i)$ is impossible, and Proposition 5.9 that $(P_iP_i, P_jP_j)$ for $i \neq j$ is impossible. Proposition 5.10 shows that blocking types $(P_1P_1, P_1)$, $(P_1, P_1P_1)$, and $(P_1,P_1)$ are impossible, and also the same for $P_2$, since this is analogous. Lemma 5.11 shows that blocking types $(P_1P_1, P_2)$ (and also $(P_2P_2, P_1)$, $(P_1, P_2P_2), (P_2.P_1P_1)$ are impossible. The harder case $(P_1,P_2)$ (and the symmetric case $(P_2,P_1)$) is shown in a series of lemmas and finally proved in Proposition 5.16. □

# 6 Conclusion

We proved that for locks where exactly one of the operations (put or take) blocks if the store is not as expected, a correct translation from SYNCSIMPLE into LOCKSIMPLE requires at least three locks, and also exhibited a correct translation for three locks. It remains open whether for all the considered blocking variants and initial storage values there are correct translations for $k \geq 3$. Future work is to provide more arguments that our results can be transferred to full concurrent programming languages. Future work is also to investigate the same questions for locks where both, put and take are blocking, if the store is not as expected (like MVars in Concurrent Haskell).

## References


[1] Gérard Boudol (1992): *Asynchrony and the Pi-calculus*. Technical Report Research Report RR-1702,inria-00076939, INRIA, France. Available at `https://hal.inria.fr/inria-00076939`.

[2] Avik Chaudhuri (2009): *A concurrent ML library in concurrent Haskell*. In: *ICFP 2009*, ACM, pp. 269–280, doi:10.1145/1596550.1596589.

[3] Rob van Glabbeek, Ursula Goltz, Christopher Lippert & Stephan Mennicke (2019): *Stronger Validity Criteria for Encoding Synchrony*. In: *The Art of Modelling Computational Systems: A Journey from Logic and Concurrency to Security and Privacy - Essays Dedicated to Catuscia Palamidessi on the Occasion of Her 60th Birthday*, *LNCS* 11760, Springer, pp. 182–205, doi:10.1007/978-3-030-31175-9_11.

[4] Daniele Gorla (2010): *Towards a unified approach to encodability and separation results for process calculi*. *Inf. Comput.* 208(9), pp. 1031–1053, doi:10.1016/j.ic.2010.05.002.

[5] Kohei Honda & Mario Tokoro (1991): *An Object Calculus for Asynchronous Communication*. In: *Proceedings of the European Conference on Object-Oriented Programming*, ECOOP '91, Springer-Verlag, pp. 133–147, doi:10.1007/BFb0057019.

[6] Robin Milner, Joachim Parrow & David Walker (1992): *A calculus of mobile processes, I*. *Information and computation* 100(1), pp. 1–40, doi:10.1016/0890-5401(92)90008-4.

[7] Joachim Niehren, Jan Schwinghammer & Gert Smolka (2006): *A Concurrent Lambda Calculus with Futures*. *Theoretical Computer Science* 364(3), pp. 338–356, doi:10.1016/j.tcs.2006.08.016.

[8] Catuscia Palamidessi (1997): *Comparing the Expressive Power of the Synchronous and the Asynchronous pi-calculus*. In: *POPL 1997*, ACM Press, pp. 256–265, doi:10.1145/263699.263731.

[9] Catuscia Palamidessi (2003): *Comparing The Expressive Power Of The Synchronous And Asynchronous Pi-Calculi*. *Math. Structures Comput. Sci.* 13(5), pp. 685–719, doi:10.1017/S0960129503004043.

[10] Simon L. Peyton Jones, Andrew Gordon & Sigbjorn Finne (1996): *Concurrent Haskell*. In: *POPL 1996*, ACM, pp. 295–308, doi:10.1145/237721.237794.

[11] Arend Rensink & Walter Vogler (2007): *Fair testing*. *Inform. and Comput.* 205(2), pp. 125–198, doi:10.1016/j.ic.2006.06.002.

[12] George Russell (2001): *Events in Haskell, and How to Implement Them*. In: *ICFP 2001*, ACM, pp. 157–168, doi:10.1145/507635.507655.

[13] David Sabel & Manfred Schmidt-Schauß (2008): *A Call-by-Need Lambda-Calculus with Locally Bottom-Avoiding Choice: Context Lemma and Correctness of Transformations*. *Math. Structures Comput. Sci.* 18(03), pp. 501–553, doi:10.1017/S0960129508006774.

[14] David Sabel & Manfred Schmidt-Schauß (2011): *A contextual semantics for Concurrent Haskell with futures*. In: *PPDP 2011*, ACM, pp. 101–112, doi:10.1145/2003476.2003492.

[15] David Sabel & Manfred Schmidt-Schauß (2012): *Conservative Concurrency in Haskell*. In: *LICS 2012*, IEEE, pp. 561–570, doi:10.1109/LICS.2012.66.

[16] Davide Sangiorgi & David Walker (2001): *The $\pi$-calculus: a theory of mobile processes*. Cambridge university press.

[17] Manfred Schmidt-Schauß, Joachim Niehren, Jan Schwinghammer & David Sabel (2008): *Adequacy of Compositional Translations for Observational Semantics*. In: *IFIP TCS 2008*, *IFIP* 273, Springer, pp. 521–535, doi:10.1007/978-0-387-09680-3_35.

[18] Manfred Schmidt-Schauß & David Sabel (2010): *Closures of may-, should- and must-convergences for contextual equivalence*. *Inform. and Comput.* 110(6), pp. 232 – 235, doi:10.1016/j.ipl.2010.01.001.

[19] Manfred Schmidt-Schauß & David Sabel (2020): *Correctly Implementing Synchronous Message Passing in the Pi-Calculus By Concurrent Haskell's MVars*. In: *EXPRESS/SOS 2020*, *Electronic Proceedings in Theoretical Computer Science* 322, Open Publishing Association, pp. 88–105, doi:10.4204/EPTCS.322.8.

[20] Manfred Schmidt-Schauß & David Sabel (2020): *On Impossibility of Simple Translations of Concurrent Calculi*. Presented at WPTE 2020, pre-proceedings available via http://maude.ucm.es/wpte20/.

[21] Manfred Schmidt-Schauß, David Sabel & Nils Dallmeyer (2018): *Sequential and Parallel Improvements in a Concurrent Functional Programming Language*. In: *PPDP 2018*, ACM, pp. 20:1–20:13, doi:10.1145/3236950.3236952.

[22] Manfred Schmidt-Schauß, David Sabel, Joachim Niehren & Jan Schwinghammer (2015): *Observational program calculi and the correctness of translations*. *Theor. Comput. Sci.* 577, pp. 98–124, doi:10.1016/j.tcs.2015.02.027.

[23] Manfred Schmidt-Schauß & David Sabel (2021): *Minimal Translations from Synchronous Communication to Synchronizing Locks (Extended Version)*. *CoRR* abs/2107.14651. Available at `https://arxiv.org/abs/2107.14651`.

[24] Jan Schwinghammer, David Sabel, Manfred Schmidt-Schauß & Joachim Niehren (2009): *Correctly translating concurrency primitives*. In: *ML 2009*, ACM, pp. 27–38, doi:10.1145/1596627.1596633.