# On Impossibility of Simple Modular Translations of Concurrent Calculi

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 previous research we showed that there are correct translations from the pi-calculus with Stop into CH, a core language of Concurrent Haskell. Correctness means that processes before and after the translation behave the same. Technically, we use contextual semantics, and require that at least so-called may- and should-convergence of processes are retained by the translation. Translations are restricted to be modular in the syntactical structure. In this paper we solve a problem that was left open in our previous research: correct translations must use at least two extra one-place buffers per channel in CH to correctly encode the synchronous communication of the pi-calculus with the asynchronous concurrency primitives provided by CH. The necessity of at least two of these buffers was conjectured, but a proof was missing. In this paper we prove the conjecture by transferring the problem into a problem of the two simple process-languages PISIMPLE and CHSIMPLE without channels and one-place buffers with unit content, where we prove the impossibility of modular simple translations. Thus we also show that the reason for requiring at least two extra one-place buffers for a correct translation from the pi-calculus with Stop into CH is only its weak synchronisation power.

## 1 Introduction

The issue of expressivity of concurrent programming languages, in particular the relation of the synchronous pi-calculus to other concurrent programming languages and their relations can in principle be answered by investigating correct translations from one language into another. For instance, translations between synchronous and asynchronous variants of the $\pi$-calculus and reasoning on their correctness can be found in [3, 1, 6]. However, in our research paper [14] we investigated modular translations from the synchronous pi-calculus [5, 12, 4], in particular the synchronous pi-calculus with `Stop` [11] into a core-language of Concurrent Haskell [2, 7, 9, 10]. The latter is a functional programming language with concurrent threads and so-called MVars as synchronization primitives. MVars are one-place buffers that are either filled or empty. As semantics we use observational semantics where two convergence properties are observed: may- and should-convergence. Both are defined in terms of a reduction semantics and a notion of success, where may-convergence means that a process can be reduced to a successful result and should-convergence means that all reduction possibilities lead to success. We were able to prove that at least one translation preserves the properties, which qualifies it for a comparison of expressivity. This translation uses the idea to construct so-called private MVars (i.e. MVars that are known to exactly one sender and one receiver) to establish safe and correct communication. This approach is comparable to using private names for translating the synchronous pi-calculus into the asynchronous pi-calculus [3, 1]. A disadvantage of this translation is that a new (private) MVar must be created (and garbage collected) for every communication action between two processes. That is why we also started an investigation for a search for classes of further correct translations with similar properties but an a priori fixed number of MVars per channel, i.e. translations that do not use (an unbounded number of) private MVars.

[*]This research is supported by the Deutsche Forschungsgemeinschaft (DFG) under grant SA2908/3-1

Our search delivers candidates of correct translations that use *two or more* so-called *check-MVars* for synchronization where a check-MVar is an MVar that is either filled by the unit value () or it is empty (check-MVars are quite similar to binary semaphores). Since no candidate translations are detected that only use *one* check-MVar, even by extensive automated searches by our implemented tool, this leads to the formulation of the conjecture that such translations are impossible. The answer to this question justifies the unavoidable use of at least 2 MVars for such translations. Our proof method by using the two simplified languages also shows that that this holds also in the case of only one channel, and that a clever use of channel-names in MVars in the pi-calculus to CH translation does not help. Informally speaking: "synchronous communication can be simulated by asynchronous communication but requires two locks". To the best of our knowledge this is a novel result.

This paper is an analysis into translations using only one check-MVar, with the final result that the conjecture holds. We transfer the reasoning into simpler languages: PISIMPLE is derived from the pi-calculus by removing several constructs and using only one channel: the simplified language only permits parallel processes consisting of !, ?, which mean output and input, respectively, and the final constants 0 (the silent process) and 1 (the successful process). Synchronous communication in PISIMPLE is always of the form $!\mathcal{Q}_1 \mid ?\mathcal{Q}_2 \mid \mathcal{Q} \to \mathcal{Q}_1 \mid \mathcal{Q}_2 \mid \mathcal{Q}$, which means that no data is transferred, only the synchronous communication is visible. The language CHSIMPLE is a simplified concurrent process language with 2 one-place-buffers, behaving like MVars that may store only a potential unit value, and where one is for the usual send/receive-synchronization and the other is for additional synchronization (for instance, the receiver may acknowledge that it received the message by this second MVar). A process consists of parallel subprocesses which are sequential commands for filling/emptying the buffers, and the final constants 0 and 1. The restricted form of MVars (that are either empty or filled, but only with a unit value) behave like locks, where filling an empty MVar corresponds to set the lock, und emptying a filled MVar corresponds to reset the lock. However, the roles of lock and unlock can also be interchanged (using filling for set and emptying for unlock) which is a difference to usual locks.

Our main result is that there is no correct modular translation from PISIMPLE into CHSIMPLE (Theorem 6.9), which can be transferred back to the corresponding question for the synchronous pi-calculus with `Stop` and CH, where it means that there are no correct translations from the synchronous pi-calculus with `Stop` into CH which only use one global check-MVar per channel (Corollary 6.10). This shows that the reason for the impossibility of simple modular translations from the pi-calculus with `Stop` into CH is only the weak synchronization property of one check-MVar together with one send-MVar. We also discuss some variants of the languages and the translations in Section 7. In particular, we consider a simpler case (so-called PT-only translations). These translations also use one check-MVar, but are not allowed to use the additional send-MVar. We show in Theorem 7.1 that there also does not exist a correct PT-only translation from PISIMPLE into CHSIMPLE.

Our proof method for the theorems is an extensive case analysis and an inductive reasoning performed using the languages PISIMPLE and CHSIMPLE, where the induction is on the syntactic length of translations, where the structuring is on a representation of various regular languages.

## 2 The Translation Problem

The paper solves the open problem of (im)possibility of certain forms of simple translations from the pi-calculus into Concurrent Haskell (CH) by showing that indeed there are no such translations. Before we formally define a specialized form of this question, we informally explain the issue for the pi-calculus and CH and then argue for a simplified but equivalent formulation.

The synchronous pi-calculus that we use, permits parallel processes, channel names, input-, output-prefixes, (new) name restriction, replication and a `Stop` constant [5, 12]. The translation is into CH, a core language of Concurrent Haskell that is a monomorphically typed core language of Haskell, extended by concurrent threads using monadic programming, where threads can be started and terminated, and the only synchronization is by one-place buffers (so-called MVars) which can be filled, if they are empty, and read-off and emptied, if they are filled. Trying to fill a filled MVar or to empty an empty MVar leads to blocking of the calling thread. In [14] there is an investigation on the correct translations from the pi-calculus into CH, showing that there are correct translations, where the simplest one uses two MVars per channel. These translations are restricted to modular ones: it is like a homomorphism and there is only one non-local element in the translations: the creation of an MVar representing `Stop`.

## 2.1 The Automated Tools

The first attack to find the most simple translation was to use our tool Refute-Pi[1] [14] for finding correct translations from pi-calculus into CH. It is implemented in Haskell such that one parameter is the search space, or a generator for the search space, and the tool scans all these translations to find a correct one. The possibility of no MVar was already excluded by formal arguments. This attack was not successful in finding a translation that only uses one global MVar, which leads to the conjecture that one global MVar is insufficient for a correct modular translation.

We will follow the idea to look for the same problem for sublanguages of the pi-calculus and CH: There is only one channel name, and one MVar that can hold a unit value and further restrictions. This leads to a simplified formulation of the languages and the translation problem (Section 3), where the proof of impossibility of a modular translation indeed answers the original problem negatively (Section 5). The final proof consists of a series of case distinctions structuring the cases using regular expressions to represent (infinite) sets of translations. We then test them as a translation for a critical, finite test set of (20) simplified pi-processes, which is again supported by the tool Refute-Regex[2]. It processes the regular expression to obtain a finite list of simple regular expressions that do not contain the alternative-operator. Such a simpler regular expression represents a class of translations. The expressions may contain non-nested exponentials and a nonterminal $M$ as the last symbol with the meaning "more…". Then the pattern is checked whether it represents a correct or incorrect translation. The tool is implemented in Haskell and that helps to automatically structure and verify the complex case distinctions.

## 3 Languages for Distributed Processes

We define abstract and simplified models of the pi-calculus and of Concurrent Haskell. We will later explain the connection between the simplification and the original and also explain its consequences.

### 3.1 The Language PISIMPLE

**Definition 3.1.** *We define the simple process language (PISIMPLE) by the following grammar:*

$$\begin{array}{lcll} \textit{Subprocesses} & \mathcal{U} & ::= & 0 \mid 1 \mid {!}\mathcal{U} \mid {?}\mathcal{U} \\ \textit{Processes} & \mathcal{P} & ::= & \mathcal{U} \mid \mathcal{U} \mathbin{\boldsymbol{|}} \mathcal{P} \end{array}$$

[1]available via `https://gitlab.com/davidsabel/refute-pi`

[2]available via `https://gitlab.com/davidsabel/refute-regex` including several case analyses that are performed by the tool.

The informal meaning of the symbols is: 0 means silence; 1 means success, ! means an output, and ? means an input, and $\mid$ is parallel composition. For example, the expression ?!!1 $\mid$ !?0 is a process, also ???!!!?1 and ?!!1 $\mid$ !?0 $\mid$ 1 $\mid$ !!!!!!?!1. We assume that $\mid$ is commutative and associative. Thus a process can be seen as a multiset of subprocesses. We sometimes write ? (or !, respectively) to abbreviate the subprocess ?0 (or !0, respectively).

**Definition 3.2.** *The* operational semantics*, or the (non-deterministic) execution (using* $\xrightarrow{PIS}$*-steps) of the processes is as follows:*

1. *If some subprocess of* $\mathcal{P}$ *is* 1*, then the complete process* $\mathcal{P}$ *is* successful.
2. *A communication step is as follows* $!\mathcal{U}_1 \mid ?\mathcal{U}_2 \mid \mathcal{P} \xrightarrow{PIS} \mathcal{U}_1 \mid \mathcal{U}_2 \mid \mathcal{P}$
3. *These steps can be iterated.*

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

Note that there may be more than one execution of processes, but every execution terminates.

**Example 3.3.** *All possible executions of* $\mathcal{P} = ?!0 \mid !!1 \mid ?0$ *are:*

$$\begin{array}{llr} \mathcal{P} = ?!0 \mid !!1 \mid ?0 & \xrightarrow{PIS} \ !0 \mid !1 \mid ?0 & (\textit{a communication}) \\ & \xrightarrow{PIS} \ !0 \mid 1 \mid 0 & (\textit{a communication; and now it is successful}) \end{array}$$

$$\begin{array}{llr} \mathcal{P} = ?!0 \mid !!1 \mid ?0 & \xrightarrow{PIS} \ !0 \mid !1 \mid ?0 & (\textit{a communication}) \\ & \xrightarrow{PIS} \ 0 \mid !1 \mid 0 & (\textit{a communication; terminated, but not successful}) \end{array}$$

$$\begin{array}{llr} \mathcal{P} = ?!0 \mid !!1 \mid ?0 & \xrightarrow{PIS} \ ?!0 \mid !1 \mid 0 & (\textit{a communication}) \\ & \xrightarrow{PIS} \ !0 \mid 1 \mid 0 & (\textit{a communication; and now it is successful}) \end{array}$$

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

We define the observations of may- and should-convergence, that will be used to test the processes. While may-convergence tests whether a successfully ending reduction sequence exists, should-convergence requires to keep may-convergence on all evaluation possibilities. In the literature there is also a notion of must-convergence that in addition forbids the possibility of an infinite evaluation. In the setting of the simple languages, should- and must-convergence coincide, since there are no infinite evaluations. For more general calculi, like the pi-calculus with replication or Concurrent Haskell, the notions differ. See e.g. [8, 13] for discussions on these notions.

**Definition 3.4.** *A process* $\mathcal{P}$ *is called*

- successful*, if there is a subprocess 1, i.e.* $\mathcal{P} = 1 \mid P'$ *for some* $\mathcal{P}'$*.*
- may-convergent *if there is some successful process* $\mathcal{P}'$ *with* $\mathcal{P} \xrightarrow{PIS,*} \mathcal{P}'$*.*
- should-convergent *if for all processes* $\mathcal{P}'$ *with* $\mathcal{P} \xrightarrow{PIS,*} \mathcal{P}'$*, the process* $\mathcal{P}'$ *is may-convergent.*
- must-divergent *or a* fail*, if there is no execution leading to a successful process.*
- may-divergent *if for some processes* $\mathcal{P}'$*:* $\mathcal{P} \xrightarrow{PIS,*} \mathcal{P}'$*, where* $\mathcal{P}'$ *is a fail.*

Note, that should-convergence is the same as must-convergence in the literature, since execution always terminates in PISIMPLE.

## 3.2 The Language CHSIMPLE

Subprocesses in the language *CHSIMPLE* are built from $0, 1$ and the symbols $S, R, P, T$, which mean: send, receive, put, take, respectively. Processes can be seen as lists of subprocesses: they are composed by parallel composition $\mid$ which is assumed to be associative and commutative.

**Definition 3.5.** *The language CHSIMPLE is defined as follows:*

$$\begin{array}{llll} \textit{Subprocesses:} & \mathcal{U} & ::= & 0 \mid 1 \mid S\mathcal{U} \mid R\mathcal{U} \mid P\mathcal{U} \mid T\mathcal{U} \\ \textit{Processes:} & \mathcal{P} & ::= & \mathcal{U} \mid \mathcal{U} \mathbin{|} \mathcal{P} \end{array}$$

We sometimes write $X$ to abbreviate $X0$ for $X \in \{S, R, P, T\}$.

**Definition 3.6.** *The operational semantics of processes of CHSIMPLE is a non-deterministic reduction relation ($\xrightarrow{CS,*}$) as follows. There are two 1-place buffers (MVars): a send-receive-MVar, and a check-MVar. Both can be full (written as $full$) or empty (written as $\emptyset$), and only an empty one can be filled and a full one can be read-off, where it is empty after the read. The four available commands are $S, R, P, T$ with following operational meaning:*

| | | |
|---|---|---|
| *S:* | *(send)* | *fills the send-receive-MVar,* |
| *R:* | *(receive)* | *empties the send-receive-MVar,* |
| *P:* | *(put)* | *fills the check-MVar,* |
| *T:* | *(take)* | *empties the check-MVar,* |

*The operational semantics operates on triples $(\mathcal{P}, M_1, M_2)$, where $\mathcal{P}$ is a CHSIMPLE-process, $M_1$ is the send-receive MVar, and $M_2$ is the check-MVar, and for some process $\mathcal{P}$ it starts with $(\mathcal{P}, \emptyset, \emptyset)$.*

*The relation $\xrightarrow{CS}$ is defined as follows:*

- $(S\mathcal{U} \mathbin{|} \mathcal{P}, \emptyset, M_2) \xrightarrow{CS} (\mathcal{U} \mathbin{|} \mathcal{P}, full, M_2)$
- $(R\mathcal{U} \mathbin{|} \mathcal{P}, full, M_2) \xrightarrow{CS} (\mathcal{U} \mathbin{|} \mathcal{P}, \emptyset, M_2)$
- $(P\mathcal{U} \mathbin{|} \mathcal{P}, M_1, \emptyset) \xrightarrow{CS} (\mathcal{U} \mathbin{|} \mathcal{P}, M_1, full)$
- $(T\mathcal{U} \mathbin{|} \mathcal{P}, M_1, full) \xrightarrow{CS} (\mathcal{U} \mathbin{|} \mathcal{P}, M_1, \emptyset)$

*and the reduction relation is the reflexive, transitive closure $\xrightarrow{CS,*}$.*

**Definition 3.7.** *A process $\mathcal{P}$ is called*

- successful, *if there is a subprocess 1 of $\mathcal{P}$, i.e. $\mathcal{P} = 1 \mathbin{|} \mathcal{P}'$ for some $\mathcal{P}$'.*
- may-convergent, should-convergent, must-divergent, *or* may-divergent, *resp. iff the triple $(\mathcal{P}, \emptyset, \emptyset)$ is* may-convergent, should-convergent, must-divergent, *or* may-divergent, *resp.*

*A triple $(\mathcal{P}, M_1, M_2)$ is called*

- successful, *if $\mathcal{P}$ is successful.*
- may-convergent, *if there is some successful $(\mathcal{P}', M_1', M_2')$ with $(\mathcal{P}, M_1, M_2) \xrightarrow{CS,*} (\mathcal{P}', M_1', M_2')$.*
- should-convergent, *if for all triples $(\mathcal{P}', M_1', M_2')$ with $(\mathcal{P}, M_1, M_2) \xrightarrow{CS,*} (\mathcal{P}', M_1', M_2')$, the triple $(\mathcal{P}', M_1', M_2')$ is may-convergent.*
- must-divergent *or a* fail, *if there is no execution leading to a successful triple.*
- may-divergent, *if for some triple $(\mathcal{P}', M_1', M_2')$: $(\mathcal{P}, M_1, M_2) \xrightarrow{CS,*} (\mathcal{P}', M_1', M_2')$, where $(\mathcal{P}', M_1', M_2')$ is a fail.*

**Example 3.8.** *An example for a reduction sequence is:*

$$(S0 \mid R1, \emptyset, \emptyset) \xrightarrow{CS} (0 \mid R1, full, \emptyset) \xrightarrow{CS} (0 \mid 1, \emptyset, \emptyset) \quad (successful)$$

*The process $S0 \mid R1$ is even should-convergent. As a second example consider the process $SR1$. It is should-convergent, since*

$$(SR1, \emptyset, \emptyset) \xrightarrow{CS} (R1, full, \emptyset) \xrightarrow{CS} (1, \emptyset, \emptyset)$$

*is the only possible reduction sequence for $SR1$.*

## 4 Translations Between the SIMPLE Languages

We consider translations from PISIMPLE to CHSIMPLE, where we are interested in *correct translations*, i.e. such that may-convergent processes are translated into may-convergent ones and should-convergent processes into should-convergent ones, and vice versa. This means that we search for so-called *convergent-equivalent* translations. See [15] for discussions and notions on the correctness of translations.

**Definition 4.1.** *A (modular) translation $\tau$ : PISIMPLE $\to$ CHSIMPLE is a homomorphism on the languages, and defined by the mappings: $\tau(!) = s_1$; $\tau(?) = s_2$; $\tau(\mid) = \mid$; $\tau(0) = 0$; $\tau(1) = 1$ where $s_1$ is a string over $\{P,T,S\}$, and $s_2$ is a string over $\{P,T,R\}$. If $s_1$ contains exactly one occurrence of S, and $s_2$ contains exactly one occurrence of R, then it is called a send-receive-unique translation,* SRU-translation.

We are mostly interested in modular SRU-translations, so sometimes we will only speak of translation, if it is clear from the context. If $\tau$ is a modular translation then we call $(\tau(!), \tau(?)) = (s_1, s_2)$ the *translation pair* (corresponding to $\tau$), which is cleary sufficient to define the translation.

**Example 4.2.** *Let $\tau_1(!) = S$, $\tau_1(?) = R$. Then $\tau_1(!0 \mid ??1) = S0 \mid RR1$.*

*For the translation $\tau_2$ with $\tau_2(!) = PPST$ and $\tau_2(?) = TPTRPPPP$, a translation example is $\tau_2(!?0 \mid ??) = PPSTTPTRPPPP0 \mid TPTRPPPPTPTRPPPP$.*

**Definition 4.3.** *A translation $\tau$ is called* correct*, if it is* convergence equivalent*, i.e. for all PISIMPLE processes $\mathcal{P}$:*

- *$\mathcal{P}$ is may-convergent iff $\tau(\mathcal{P})$ is may-convergent, and*
- *$\mathcal{P}$ is should-convergent iff $\tau(\mathcal{P})$ is should-convergent.*

**Remark 4.4.** *Contextual equivalence on subprocesses in the calculi PISIMPLE and CHSIMPLE is defined as $\mathcal{U}_1 \sim \mathcal{U}_2$ if for all contexts C (i.e. processes with a hole at subprocess position):*

- *$C[\mathcal{U}_1]$ is may-convergent iff $C[\mathcal{U}_2]$ is may-convergent*
- *$C[\mathcal{U}_1]$ is should-convergence iff $C[\mathcal{U}_2]$ is should-convergent*

*Analogously, contextual equivalence can be defined on processes by testing invariance of may- and should-convergence in all contexts that have the hole at process position.*

*Correctness of a translation implies (see [15]) that the translation is* observationally correct*, i.e. $C[\mathcal{U}]$ is may-convergent iff $\tau(C)[\tau(\mathcal{U})]$ is may-convergent and $C[\mathcal{U}]$ is should-convergent iff $\tau(C)[\tau(\mathcal{U})]$ is should-convergent, which implies* adequacy *w.r.t. $\sim$ of $\tau$, i.e. $\tau(\mathcal{U}_1) \sim \tau(\mathcal{U}_2) \implies \mathcal{U}_1 \sim \mathcal{U}_2$.*

**Example 4.5.** *Translation $\tau_1$ from Example 4.2 is not correct, since for instance the process $!?1$ is must-divergent, but $\tau_1(!?1) = SR1$ is should-convergent (see Example 3.8).*

*The translation $\tau_3(!) = SPP$, $\tau_3(?) = RTT$ can be refuted, however, the counterexample process was hard to find. We found one and verified it using Refute-Pi. We explain it for the SIMPLE languages. One counterexample process is $! \mid ? \mid !?!1$, which is neither may- nor should-convergent, which means it is must-divergent. The translated process $SPP \mid RTT \mid SPPRTTSPP1$ has a successful reduction sequence: execute the commands corresponding to order given by the following numbering:*

$$\begin{array}{cccccccccccccccc} S & P & P & \mid & R & T & T & \mid & S & P & P & R & T & T & S & P & P & 1 \\ & 6 & 9 & & 3 & 4 & 13 & & 1 & 2 & 5 & 7 & 8 & 10 & 11 & 12 & 14 & \end{array}$$

*This shows that the translated process is may-convergent. Hence the translation $\tau_3$ is incorrect.*

**Remark 4.6.** *Note that if we would allow arbitrary translations $\sigma$ : PISIMPLE $\to$ CHSIMPLE, then it would be easy to define a correct one: Let*

$$\sigma(\mathcal{P}) = \begin{cases} 0, & \text{if } \mathcal{P} \text{ is must-divergent} \\ 1, & \text{if } \mathcal{P} \text{ is should-convergent} \\ P0 \mid P1, & \text{otherwise} \end{cases}$$

*Then $\mathcal{P}$ is may-convergent iff $\sigma(\mathcal{P})$ is may-convergent, and $\mathcal{P}$ is should-convergent iff $\sigma(\mathcal{P})$ is should-convergent. Note that $\sigma$ is computable, since may- and should-convergence in PISIMPLE are decidable.*

**Remark 4.7.** *There are (modular SRU-) translations $\tau$ according to Definition 4.1 such that*

- *$\tau$ is may-convergence* preserving*, i.e. for all PISIMPLE-processes $\mathcal{P}$: If $\mathcal{P}$ is may-convergent, then $\tau(\mathcal{P})$ is may-convergent. The simplest such translation is $\tau_1(!) = S$ and $\tau_1(?) = R$, since every communication in $\mathcal{P}$ can be done in $\tau_1(\mathcal{P})$ by two steps.*

- *$\tau$ is may-convergence* reflecting*, i.e. for all PISIMPLE-processes $\mathcal{P}$: if $\tau(\mathcal{P})$ is may-convergent, then $\mathcal{P}$ is may-convergent. The simplest one is $\tau_5(!) = TPS$ and $\tau_5(?) = R$. Then $\tau_5(\mathcal{P})$ cannot execute any translated $!$-operation, since $TPS$ will deadlock without performing any operation. Thus $\tau_5(\mathcal{P})$ can also not perform any encoded $?$-operation, since R-steps are impossible without a preceding S-operation. Thus $\tau_5(\mathcal{P})$ can only be may-convergent, if it contains a 1 on the surface, but then $\mathcal{P}$ also contains a 1 on the surface and thus is may-convergent.*

- *$\tau$ is should-convergence* reflecting*, i.e. for all PISIMPLE-processes $\mathcal{P}$: If $\tau(\mathcal{P})$ is should-convergent, then $\mathcal{P}$ is should-convergent. The simplest one is $\tau_5(!) = TPS$ and $\tau_5(?) = R$, as above. The arguments are the same as for the previous case.*

- *$\tau$ may-convergence* equivalent*, i.e. for all PISIMPLE-processes $\mathcal{P}$: process $\mathcal{P}$ is may-convergent if, and only if $\tau(\mathcal{P})$ is may-convergent. We found such a translation by using our tool. It refutes all translations of size $< 6$, but it is unable to refute the following three translations of size 6: $(\tau_6(!), \tau_6(?)) = (PSPT, RT)$, $(\tau_7(!), \tau_7(?)) = (ST, PPTR)$, and $(\tau_8(!), \tau_8(?)) = (SPP, TTR)$. All three translations preserve may-convergence, which can be easily shown, by observing that for $i = 6,7,8$: $(\tau_i(!)\tau_i(\mathcal{P}_1) \mid \tau_i(?)\tau_i(\mathcal{P}_2) \mid \tau_i(\mathcal{P}_3), \emptyset, \emptyset) \xrightarrow{CS,*} (\tau_i(\mathcal{P}_1) \mid \tau_i(\mathcal{P}_2) \mid \tau_i(\mathcal{P}_3), \emptyset, \emptyset)$*

  *We now show that $\tau_8$ is may-convergence equivalent. It suffices to show that $\tau_8$ reflects may-convergence: this can be done by an inductive argument. Consider a successful reduction sequence in the image (i.e. for $\tau_8(\mathcal{P})$ for some PISIMPLE-process $\mathcal{P}$), where only the prefixes are shown and where the state of the MVars is omitted. If the reduction sequence is of the form $SPP \mid TTR \to PP \mid TTR \to P \mid TTR \to P \mid TR \to TR \to R \to \emptyset$, then this can be simulated in*

*PISIMPLE by one reduction step. A special case is* $SPP1 \mid TTR \mid TTR \to PP1 \mid TTR \mid TTR \to P1 \mid TTR \mid TTR \to P1 \mid TR \mid TTR \to 1 \mid TR \mid TTR$*, where in PISIMPLE a successful state can be reached by one reduction step, which corresponds to* $TR \mid TR$*. The only other case is as follows:* $SPP \mid TTR \mid TTR \to PP \mid TTR \mid TTR \to P \mid TTR \mid TTR \to P \mid TR \mid TTR \to TR \mid TTR \to TR \mid TR$*, which deadlocks, and no further reduction step is possible, hence this reduction sequence is not a witness for may-convergence in the image of* $\tau_8$*. There are no other cases, hence* $\tau_8$ *is may-convergence reflecting.*

*Note that our tool Refute-Pi invalidates should-convergence preservation (and equivalence) in a large search space. Thus an alternative proof of non-existence of modular translations may be trying to prove that no should-convergence preserving translation exists, or no should-convergence equivalent translation. However, this may be harder than our approach.*

## 5 The Pi-Calculus, Concurrent Haskell and their Relation to PISIMPLE and CHSIMPLE

We will argue (in a sketchy way) that the nonexistence of a modular correct SRU-translation from PISIMPLE into CHSIMPLE, also shows the nonexistence of modular correct translation from the pi-calculus [5, 12] with `Stop` [11], into Concurrent Haskell [2, 7], where the translations can only use global names for channels and MVars.

We informally explain the pi-calculus and Concurrent Haskell such that our argument becomes plausible: The main operation in the pi-calculus is communication: $x(y).\mathcal{P} \mid \overline{x}u.\mathcal{P}' \mid \mathcal{P}''$ permits an operation: $x$ is the channel-name where the two subprocesses $x(y).\mathcal{P}$ and $\overline{x}u.\mathcal{P}'$ can exchange data, which can only be channel names. The result is $\mathcal{P}[u/y] \mid \mathcal{P}' \mid \mathcal{P}''$. Thus the channel name $u$ is transported over $x$ from the second subprocess to the first one and the place holder $y$ is replaced by $u$ in $\mathcal{P}$. There are also other operations like replication of a subprocess, and manipulating name restriction using the operator $\nu$. We added the constant process `Stop` that plays the role of a signal for success. Concurrent Haskell (CH) as we use it, is a process calculus where the sequence of actions is done using monadic programming for defining the sequence of actions, and the subprocesses are programmed mainly in usual Haskell.

PISIMPLE corresponds to a subset of the variant of the pi-calculus with `Stop`: `Stop` corresponds to 1; there is only one channel name, say $x$, and ! corresponds to the output-prefix $\overline{x}y$, and ? corresponds to the input-prefix $x(y)$, where $y$ can be omitted, since only one fixed name will be transported over all $y$. Since there is also only one name, we can omit it from the syntax, and thus the output $\overline{x}y$ will be represented by ! and the input $x(y)$ by ?. The translations from the pi-calculus into Concurrent Haskell that we investigated in [14] are only permitted to use one MVar for send and receive (i.e. for delivering the message). In the simpler setting of this paper, only the information filled or empty is available (but no message). Thus, we abbreviate those by the symbols $S$ and $R$. There is also only one global check-MVar, and writing and reading in CHSIMPLE are represented by $P$ and $T$. Note also that may-and should-convergence are coincident by this transfer.

Thus the existence of a modular correct translation (satisfying the global condition) from the pi-calculus with `Stop` into CH would imply a correct modular SRU-translation from PISIMPLE into CHSIMPLE.

## 6 Analyzing and Refuting Translation Patterns

### 6.1 Refuting Correctness of Translations

Now we explain how to proceed to refute a modular simple translation, where we will give only an overview of our arguments which we did completely by a mixture of hand-made proofs and using our tool Refute-Regex for lots of case analyses, and also the tool Refute-Pi for eliminating several short translations.

The following is proved for a correct and modular translation $\tau$:

- The number of $P$-s is the same as the number of $T$-s in the multiset-union $\tau(!) \cup \tau(?)$.
- $\tau(!) \mid \tau(?)$ can be executed without any deadlock until the process is empty. Moreover, any partial execution sequence of $\tau(!) \mid \tau(?)$ can be completed, such that the process is empty after the reduction sequence.
- There are no correct translations $\tau$ with $|\tau(!)| + |\tau(?)| \leq 10$.
  This is done by the tool Refute-Pi for checking translations from the synchronous pi-calculus with `Stop` into CH see the remarks in Section 5.

### 6.2 Analyzing the Structure of Potentially Correct Translations

In many cases the so-called flat PISIMPLE processes are easy test examples. These are defined as consisting only of subprocesses $!k, ?k$ for $k \in \{0,1\}$. Small flat processes are $\mathcal{Q}_1 = \ !1 \mid ?$, $\mathcal{Q}_2 = \ !0 \mid ?1$, $\mathcal{Q}_3 = \ !1 \mid !1 \mid ?$ and $\mathcal{Q}_4 = \ ! \mid ?1 \mid ?1$, which are should-convergent. The processes $\mathcal{Q}_{3,n} = !1 \mid \ldots \mid !1 \mid ?$ with $n$ copies of $!1$, and $\mathcal{Q}_{4,n} = \ ?1 \mid \ldots \mid ?1 \mid !$, with $n$ copies of $?1$, are also should-convergent, and are used as test-example processes in proofs.

In the following we fix the notation for $s_1, s_2, r_1, r_2$ for a given translation $\tau$, which is always characterized by $\tau(!) = s_1 S s_2$ and $\tau(?) = r_1 R r_2$.

Now we explain how to proceed in the cases of translation patterns until all patterns are refuted.

A tedious case analysis shows the first result on the structure of the translation strings

**Proposition 6.1.** *Let $\tau$ be a correct SRU-translation for flat processes. Then $s_1$ and $r_1$ do not contain the pattern $PP$ nor $TT$.*

*Proof (Sketch).* Assuming a translation with pattern $PP$ in $s_1$ can often be refuted from being correct by trying the test-processes $\mathcal{Q}_{3,n}$, which for sufficiently large $n$ can force a deadlock. Similar for the pattern $TT$ and $\mathcal{Q}_{4,n}$.

Note that the cases for $s_1$ and $r_1$ are mostly symmetric but not completely, since the execution of $R$ is only possible after at least one execution of some $S$. □

A consequence of this proposition is that the prefixes $s_1$ and $r_1$ are sequences of the form $\ldots PTPT \ldots$ and can be written as one of the following four patterns: $(PT)^n$, $(PT)^n P$, $(TP)^n$, $(TP)^n T$ for some $n$.

### 6.3 Further Restrictions for Prefixes

We will use regular expressions for representing infinitely many translation strings in CHSIMPLE using the following ideas. We in general indicate whether a translation string ends, or may be continued. In the latter case we use $M$ for indicating that there may be "more". We also use $\lambda$ for the empty string in regular language patterns, repetitions using flat exponentiation and also | for alternatives.

Case analyses supported by our tool Refute-Regex permit to show the following properties:

**Lemma 6.2.** *There are no correct modular SRU-translations with* $s_1 \in \{(TP)^nT, (PT)^nP\}$.

*Proof (sketchy explanation).* This proof is done making case distinctions on the other parts where restricted regular expressions are used to catch all cases. The regular expressions have repetitions, but not nested ones. These repetitions permit implicit induction. □

**Lemma 6.3.** *There are no correct modular SRU-translations with* $r_1 \in \{(TP)^nT, (PT)^nP\}$.

*Proof (sketchy explanation).* This proof is very similar to the proof of the previous lemma. However, note that there is an asymmetry between $s_1, r_1$, hence there are some cases that are different. □

**Example 6.4.** *The general idea of the tool is to look for deadlocks in the prefix, where exponents are treated incompletely by an expansion, and if the M has to be expanded, then this is interpreted as unknown. If the pattern failed completely, then it is signalled as unsolved. In a subcase, our simulator Refute-Regex has as input* $(S(PT)^*(\lambda|T|TTM|P|PPM), (PT)^*PR(\lambda|PM|TM))$, *which will be expanded first to* 12 *translation pairs. The subexpression* $S(PT)^*$ *is expanded into* $(S \mid SPT)$ *and* $(PT)^*$ *into* $\lambda \mid PT$, *and then into 48 pairs. All cases are refuted but* $(STTM, PRPM)$. *This is done using further detailing and calls.*

The result is that only the prefixes $(TP)^n$ and $(PT)^m$ are possible for $s_1, r_1$ of correct modular SRU-translations. Hence there are four combinations of the prefixes. We will show in the following that also these restricted translations can be refuted by further detailing $n, m, s_2, r_2$.

First we show two lemmas that cannot be proved directly using our simulator. We will exploit them in submitting specialized patterns into the simulator Refute-Regex.

**Lemma 6.5.** *Let* $\tau$ *be a modular SRU-translation. Let* $\tau(!) = (PT)^nSP^ks_3$ *and* $\tau(?) = RT^hr_3$, *where* $n \geq 0$, $h,k \geq 2$, $h+k \geq 5$, $s_3$ *does not start with P, and* $r_3$ *does not start with* $T$. *Then the translation is not correct.*

**Lemma 6.6.** *Let* $(\tau(!), \tau(?))$ *be a translation pair of the form* $((PT)^nST^kM, RP^hM)$, *where* $n \geq 0$, $h,k \geq 2$. *Then the translation is not correct.*

Using the two previous lemmas in combination with our tool, we are able to prove the following two lemmas, where the first one refutes the first combination of the remaining patterns for $s_1, r_1$, and the second one refutes the other combinations.

**Lemma 6.7.** *Let* $\tau$ *be a correct modular SRU-translation. Then* $s_1, r_1$ *of this translation are not of the form* $(PT)^n$. *More details according to the structure of the arguments are:*

1. *In the case* $\tau(!) = (PT)^nSs_2$ *and* $\tau(?) = (PT)^mRr_2$, *the string* $r_2$ *does not start with* $T$.
2. *In the case* $\tau(!) = (PT)^nSs_2$ *and* $\tau(?) = (PT)^mRr_2$, *the string* $r_2$ *is nonempty and does not start with P.*

**Lemma 6.8.** *For a modular SRU-translation* $\tau$, *the translation patterns for the three cases*

1. $\tau(!) = (PT)^nSs_2$ *and* $\tau(?) = (TP)^mRr_2$,
2. $\tau(!) = (TP)^nSs_2$ *and* $\tau(?) = (PT)^mRr_2$,
3. $\tau(!) = (TP)^nSs_2$ *and* $\tau(?) = (TP)^mRr_2$,

*do not contain any correct translations.*

*Proof (Sketch).* This is proved by several case distinctions using regular expressions for the strings $s_1, s_2, r_1, r_2$ and then using Refute-Regex. □

Summarizing, Proposition 6.1 and Lemmas 6.2, 6.3, 6.5, 6.7, and 6.8 imply the following theorem.

**Theorem 6.9.** *There are no modular correct SRU-translations from PISIMPLE into CHSIMPLE.*

Transferring this theorem to the issue of simple translations from the pi-calculus into CH, according to the arguments in section 5, we obtain:

**Corollary 6.10.** *There are no modular correct translations from the pi-calculus into CH, where the translations uses only one check-MVar per channel.*

## 7 Translations into Variants of CHSIMPLE

As a further question we consider very simple modular translations, i.e. we consider the case that the translations are not permitted to use send- nor receive-actions. We call those translations *PT-only translations*, and hence it is only permitted to use $P,T$ for the actions, and $0,1$ for the last symbol of subprocesses. We argue that there are no correct very simple translations from PISIMPLE into CHSIMPLE, where the case analysis is also supported by our tool, to verify the cases and the case distinctions.

**Theorem 7.1.** *There are no correct PT-only translations.*

*Proof (Sketch).* This can be proved by a case analysis on the patterns for translations $\tau(!)$ and $\tau(?)$. First it can be shown using a case analysis that for a correct PT-only translation, the strings $\tau(!)$ and $\tau(?)$ do not contain an adjacent occurrence of $P$ or $T$, i.e. $PP$ or $TT$ cannot occur. This implies that $\tau(!)$ and $\tau(?)$ have alternating occurrences of $P,T$, hence are of the form $(PT)^n$, $(PT)^nP$, $(TP)^n$, or $(TP)^nT$ for $n \geq 0$. This however is not possible:

- The empty string is not possible, since then $!1$ (or $?1$), which are must-divergent, would be translated into 1, which is should-convergent.
- The same holds for $(PT)^n$ and $(PT)^nP$, since these strings alone can be completely executed.
- The remaining strings are $(TP)^n$, and $(TP)^nT$. PT-only translations consisting only of these strings for ? and ! cannot be correct, since the translated processes are deadlocked where the co-image is should-convergent. □

**Remark 7.2.** *There is a correct modular SRU-translation from PISIMPLE $\to$ CHSIMPLE$_2$, where CHSIMPLE$_2$ is like CHSIMPLE, but with an extra copy of $P,T$; i.e. with $P_1,T_1,P_2,T_2$, and $P_2,T_2$ also has an extra full/empty marker in the operational semantics. The correct modular translation is $\tau_4(!) = P_1ST_2T_1$ and $\tau_4(?) = RP_2$. Since this is a simplified form of a correct SRU-translation of the pi-calculus into CH, a correctness proof can be derived from the paper [14] and Section 5. An informal argument is that there can only be one translated $!$-operation run at the same time, since the operation is protected by $P_1$ at the beginning and $T_1$ at the end, which acts like a mutex to protect a critical section. For the translation of $?$, i.e. $RP_2$, there is no such protection, but if a translated sender has performed S (in the sequence $P_1ST_2T_1$ and a translated receiver performs R, then no other sender or receiver can perform any step, since the MVar corresponding to $P_1$ is full and the MVar corresponding to $P_2$ is empty). Thus the only possible step in the whole system is to perform $P_2$ and thus the receiver completes its operation.*

**Remark 7.3.** *There is a correct modular PT-only translation from PISIMPLE $\to$ CHSIMPLE$_3$, where CHSIMPLE$_3$ is defined analogously to CHSIMPLE$_2$ in Remark 7.2 but with 3 copies of $P,T$. The correct modular PT-only translation is $\tau(!) = P_1P_3T_2T_1$ and $\tau(?) = T_3P_2$. Correctness can be derived from the correctness result in Remark 7.2, since it replaces the send-MVar and $S,R$ by the third check-MVar and $P_3,T_3$.*

**Remark 7.4.** *One gap that remains to be analysed is whether there is a correct PT-only translation from PISIMPLE $\rightarrow$ CHSIMPLE$_2$. This is the same question as to whether there are correct translations from PISIMPLE $\rightarrow$ CHSIMPLE where $R,S$ may be used multiple times.*

## 8 Conclusion

In this paper we solve an open question on the existence/nonexistence of correct modular translations from the pi-calculus into *CH*, where the special question was on the number of MVars (synchronized one-place buffers) that are required for every channel in the pi-calculus. The answer is that two are sufficient, and that one is insufficient. It also establishes a sharp boundary between synchronous and asynchronous communication in concurrent calculi.

## References

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

[2] Haskell-Community (2020): *Haskell Main Website*. `www.haskell.org`.

[3] Kohei Honda & Mario Tokoro (1991): *An Object Calculus for Asynchronous Communication*. In: *ECOOP 1991*, Springer-Verlag, pp. 133–147, doi:10.1007/BFb0057019.

[4] Robin Milner (1999): *Communicating and mobile systems - the Pi-calculus*. Cambridge University Press.

[5] 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.

[6] 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.

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

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

[9] 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.

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

[11] David Sabel & Manfred Schmidt-Schauß (2015): *Observing Success in the Pi-Calculus*. In: *WPTE 2015*, *OASICS* 46, pp. 31–46, doi:10.4230/OASIcs.WPTE.2015.31.

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

[13] 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.

[14] Manfred Schmidt-Schauß & David Sabel (2020): *Embedding the Pi-Calculus into a Concurrent Functional Programming Language*. Frank report 60, Institut für Informatik. Fachbereich Informatik und Mathematik. J. W. Goethe-Universität Frankfurt am Main. Available at `http://www.ki.informatik.uni-frankfurt.de/papers/frank/frank-60v4.pdf`. Submitted for publication.

[15] 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.