Nils Berg, Thomas Göthel, Armin Danziger, Sabine Glesner

Preserving Liveness Guarantees from Synchronous Communication to Asynchronous Unstructured Low-Level Languages

The final authenticated publication is available online at https://doi.org/10.1007/978-3-030-02450-5_18

Berg, N., Göthel, T., Danziger, A., & Glesner, S. (2018). Preserving Liveness Guarantees from Synchronous Communication to Asynchronous Unstructured Low-Level Languages. In Formal Methods and Software Engineering (LNCS, volume 11232). pp. 303–319. Springer International Publishing. https://doi.org/10.1007/978-3-030-02450-5_18

Preserving Liveness Guarantees from Synchronous Communication to Asynchronous Unstructured Low-Level Languages

Nils Berg, Thomas Göthel, Armin Danziger, and Sabine Glesner

Technische Universität Berlin
n.berg@tu-berlin.de

Abstract. In the implementation of abstract synchronous communication in asynchronous unstructured low-level languages, e. g. using shared variables, the preservation of safety and especially liveness properties is a hitherto open problem due to inherently different abstraction levels. Our approach to overcome this problem is threefold: First, we present our notion of *handshake refinement* with which we formally prove the correctness of the implementation relation of a handshake protocol. Second, we verify the soundness of our *handshake refinement*, i. e., all safety and liveness properties are preserved to the lower level. Third, we apply our *handshake refinement* to show the correctness of *all* implementations that realize the abstract synchronous communication with the handshake protocol. To this end, we employ an exemplary language with asynchronous shared variable communication. Our approach is scalable and closes the verification gap between different abstraction levels of communication.

Keywords: unstructured code, liveness properties, handshake protocol, formal verification, refinement

1 Introduction

In the rigorous model-driven design of low-level implementations, formal specifications are iteratively refined until an implementation model is reached. In the subsequent transition to executable code, correctness is mostly subject to informal reasoning due to the different abstraction levels. In this paper, we consider unstructured low-level languages that are required to preserve safety and liveness properties from the formal specification. The formal verification of the relation between specification and implementation of communicating low-level code can be split in two parts: *1)* State transformations and control flow, and *2)* communication. While we have presented an approach for *1)* in [7,8], in this paper we focus on the low-level implementation of communication. In particular, we do not consider the general question whether it is possible to implement synchronous communication with asynchronous means, as this was shown in e. g. [3]. In contrast, we propose a methodology to verify that a specification

using abstract synchronous communication and a concrete implementation in a low-level language using asynchronous shared variable communication have the same safety and liveness properties based on a simple handshake protocol.

In synchronous communication, sender and receiver are determined at the same time, whereas they are determined at different points in time in asynchronous communication. Thus, asynchronous communication has more decision points and a different branching behavior. The major problem is to prove preservation of liveness properties for systems with different branching behavior. To overcome this problem, we define the *handshake refinement* that enables the construction and formal verification of implementation relations for the abstract communication instruction. We show that this relation preserves safety and liveness properties. Finally, we use our notion of *handshake refinement* to show the correctness of the implementation of abstract synchronous communication with a handshake protocol in our generalized low-level language using shared variable communication. Our theorem shows that *all* implementations with this protocol are correct. This once-and-for-all approach is highly scalable and allows for compositional reasoning over shared variable communication. While the *handshake refinement* is designed for this specific handshake protocol, its concepts can be adapted to other protocols, which is left for future work.

2 Related work

In [4], Broy and Olderog investigate the relationship between synchronous and asynchronous communication, where asynchronous communication is *buffered*, e. g. via an additional buffer process. However we do not consider high-level constructs such as buffers in our implementation language. Apart from the different abstraction level used by Broy and Olderog, their transformation from synchronous to asynchronous systems is to introduce buffers for all (previously synchronous) communication. In doing so, they lose synchronicity and the "refusal structure" of the synchronous specification, i. e. the transformation does not preserve liveness properties.

Peeters [9] models hardware, where low-level communication is synchronous (a wire from sender to recipient). They still use synchronization primitives for the implementation, and thus, does not apply to the problem we consider.

Basu et al. [1] define synchronizability of asynchronous systems. We show a similar relation, namely that it is appropriate to consider a synchronized version of the asynchronous system. The use of modeled queues is too abstract for our problem, as we aim at verifying the abstract communication construct (e.g. queue in this case) itself.

The `CSP++` framework from Gardner [6] constructs a communication backbone from a CSP (Communicating Sequential Processes) process, which can then be enriched with `C++` code. Although the idea of this framework is akin to correct by construction design, the verification of this framework itself is not addressed.

In summary, all these approaches consider either a rather high level of abstraction, or do not consider formal verification of safety and liveness properties.

Vertical Bisimulation by Rensink and Gorrieri [10] provides a congruence relation for action refinements whose implementations can interleave. They do not consider the refinement of the synchronization mechanism itself: Both source language and target language use the same CSP-like synchronization. Their definition is different from the standard bisimulation in that it keeps track of started executions of the implementations. This idea inspired our definition of the *handshake refinement*.

Frutos Escrig et al [5] propose *global bisimulations* to achieve associativity of nondeterministic choice. It has some similarity to our problem with split up decisions (i. e. comparing two decisions at once to two consecutive decisions), however our choices are deterministic. Moreover, it is specifically intended to be a *symmetric* relation, and our problem is asymmetric, as we consider different levels of abstraction. Therefore, the vertical bisimulation is a fitter candidate to be adapted to our problem.

3 Background

In the following, we briefly introduce CSP and the low-level language CUC (Communicating Unstructured Code) with synchronous communication which we formerly presented. We base our notion of safety and liveness on CSP and obtain compositionality for CUC by using CSP communication.

3.1 Communicating Sequential Processes (CSP)

For our specification language CUC, we consider CSP-like abstract synchronous communication (without broadcast) throughout this paper. The advantage is that we can perform proofs compositionally, which is inherited from CSP. In CSP [11], a refinement ($Spec \sqsubseteq Impl$) describes a subset relation of the behavior. The *trace* semantics ($\mathcal{T}$) records the traces and the trace refinement ensures the preservation of safety properties. The *stable failures* semantics ($\mathcal{SF}$) additionally records the refused (and by negation the possible) events at each stable state, thus allowing the stable failures refinement to ensure the preservation of liveness properties. In CSP, the notion of refinement is compositional w.r.t. contexts ($\mathcal{C}$), i. e., when only a part of the system is refined, the whole system is also in a refinement relation: $A \sqsubseteq B \Longrightarrow \mathcal{C}(A) \sqsubseteq \mathcal{C}(B)$. As parallel composition can also be part of the context $\mathcal{C}$, this allows for modular verification of concurrent systems.

3.2 Communicating Unstructured Code (CUC)

We aim at verifying safety and liveness properties of the shared variable implementation of abstract synchronous communication. To focus on the difference between abstract synchronous and low-level asynchronous communication, we choose two languages which only differ in this aspect. As the implementation language should be a low-level language with shared variable communication, we choose a low-level language with abstract synchronous communication as

a specification language. To this end, we employ the language CUC. It is a generic low-level language with an abstract communication instruction, using CSP's multi-way synchronization. We introduced its operational and trace semantics in [7] and its stable failures semantics together with a Hoare calculus in [8]. The latter provides a framework to verify the stable failures refinement relation between a CSP process and a CUC program, ensuring that the CUC program preserves all safety and liveness properties of the CSP process.

We give a brief overview over CUC here, for details see [8]. The operational semantics is depicted in Figure 1. The state σ is split into its program counter σ_{pc} and its register store σ_{rs}, *code* is a fixed set of labeled instructions. CUC has three instructions: *1)* A nondeterministic multiple assignment (DO), which can be instantiated to actual low-level instruction, e. g. arithmetic operations. *2)* A conditional branch (CBR) and *3)* the communication primitive. It communicates an event nondeterministically chosen from the result of f_{ev} and then changes the state according to f_{reg}. The `comm` instruction modifies the register store to record input data. The implementation of `comm` f_{ev} f_{reg} is the subject of this paper. The communication of CUC is the same as communication in CSP: All programs offer events (in their alphabets α_i), and if multiple offer the same events, they non-deterministically choose one of them and make a synchronous step (SYNC). Non-synchronized events and τ are performed interleavingly (INTERLEAVING).

$$\frac{(\sigma_{pc}, \texttt{do}\ f) \in \mathit{code} \qquad \sigma'_{rs} \in f(\sigma_{rs}) \qquad \sigma'_{pc} = \sigma_{pc} + 1}{\sigma \xrightarrow{\tau}_{\mathit{code}} \sigma'} \text{ DO}$$

$$\frac{(\sigma_{pc}, \texttt{cbr}\ b\ m\ n) \in \mathit{code} \qquad \sigma'_{rs} = \sigma_{rs} \qquad b\ \sigma \wedge \sigma'_{pc} = m \vee \neg b\ \sigma \wedge \sigma'_{pc} = n}{\sigma \xrightarrow{\tau}_{\mathit{code}} \sigma'} \text{ CBR}$$

$$\frac{(\sigma_{pc}, \texttt{comm}\ f_{ev}\ f_{reg}) \in \mathit{code} \quad \mathit{ev} \in f_{ev}(\sigma_{rs}) \quad \sigma'_{rs} = f_{reg}(\sigma_{rs}, \mathit{ev}) \quad \sigma'_{pc} = \sigma_{pc} + 1}{\sigma \xrightarrow{\mathit{ev}}_{\mathit{code}} \sigma'} \text{ COMM}$$

$$\frac{\sigma_1 \xrightarrow{a}_{c_1} \sigma'_1 \qquad \sigma_2 \xrightarrow{a}_{c_2} \sigma'_2 \qquad a \in \alpha_1 \cap \alpha_2}{\sigma_1 \parallel \sigma_2 \xrightarrow{a}_{\left(c_1 \alpha_1 \parallel_{\alpha_2} c_2\right)} \sigma'_1 \parallel \sigma'_2} \text{ SYNC}$$

INTERLEAVING-LEFT

$$\frac{\sigma_1 \xrightarrow{a}_{c_1} \sigma'_1 \qquad a \in (\alpha_1 \cup \{\tau\}) \setminus \alpha_2}{\sigma_1 \parallel \sigma_2 \xrightarrow{a}_{\left(c_1 \alpha_1 \parallel_{\alpha_2} c_2\right)} \sigma'_1 \parallel \sigma_2}$$

INTERLEAVING-RIGHT

$$\frac{\sigma_2 \xrightarrow{a}_{c_2} \sigma'_2 \qquad a \in (\alpha_2 \cup \{\tau\}) \setminus \alpha_1}{\sigma_1 \parallel \sigma_2 \xrightarrow{a}_{\left(c_1 \alpha_1 \parallel_{\alpha_2} c_2\right)} \sigma_1 \parallel \sigma'_2}$$

Fig. 1: Operational Semantics for CUC

In the transition from synchronous to asynchronous communication, we perform a refinement based on low-level communication protocols. In this paper, we focus on a handshake protocol over shared variables and restrict the use of CUC constructs accordingly, i. e., to use only a sender and a receiver version of `comm` and exclude communication with the environment. Additionally, we illustrate our approach with a simple protocol here, and therefore prohibit the use of external choice within a component. To restrict the communication to directed

communication, we consider two restricted variants of `comm`, as defined below. Let c be a channel, x_s and x_r local registers, id the process id of the current process and ID the set of all process ids. The event $c.s.r.v$ is composed of the channel c, the ids of the sender s and the receiver r, and the transferred data value v. Finally, let $val(c.s.r.v) = v$ extract the data value of an event.

$$\texttt{comm}_s\, id\, c\, x_s \coloneqq \texttt{comm}\big(\lambda\sigma.\, \{c.id.r.\sigma_{rs}(x_s) \mid r \in ID \wedge r \neq id\}\big)\big(\lambda a\, \sigma.\, \sigma\big)$$
$$\texttt{comm}_r\, id\, c\, x_r \coloneqq \texttt{comm}\big(\lambda\sigma.\, \{c.s.id.v \mid s \in ID \wedge s \neq id\}\big)\big(\lambda a\, \sigma.\, \sigma(x_r \coloneqq val(a))\big)$$

$\texttt{comm}_s$ offers events on its channel c, using its own id as sender, and all possible ids as receiver. The data value is the value of its local storage at x_s. After successful communication, the sender does not change its local state. $\texttt{comm}_r$ offers events on its channel c, using its own id as a receiver, all possible ids as sender, and all possible data values. After successful communication, the receiver updates its local storage at x_r to the value of the communicated event. By using events that explicitly contain the id of the sender or the receiver respectively, we enforce that senders cannot communicate among one another and the same for receivers.

In contrast to CSP and CUC, there is no environment in low-level shared variable communication. Thus, a lone `comm` in CUC should not synchronize with the environment but block. To enforce this in CUC, we only consider programs with at least two components. Furthermore, the synchronization alphabet of each concurrent program c_i is given by $\alpha_i = \{c.s.r.v \in \Sigma \mid (s \in ids(P_i) \vee r \in ids(P_i))\}$.

4 Shared Variable Semantics (SV)

In this section, we present the language *Shared Variables* (SV) and give its operational semantics. The intent of SV is to have a language with a *pure interleaving* semantics (in contrast to CUC) and to implement synchronous communication over shared variables with it. SV contains the instructions `do` f and `cbr` just like CUC, but instead of the abstract communication instruction `comm`, it contains the instructions needed for the low-level implementation of communication and synchronization over shared variables: `read`, `write` and `cas` (Compare-and-Set).

The operational semantics for SV is depicted in Figure 2. For each component, there is a program counter σ_{pc} and a local register store σ_{rs} as in CUC. Furthermore, there is a global state Γ, which holds the values of locks, signals, and shared variables. `do` and `cbr` (as described in rules DO and CBR) have basically the same semantics as in CUC: They change the local state and the program counter, but leave the global state Γ unchanged. `cas` (as described in rules CAS-T and CAS-F) compares the value at a given address sv to a value v_1 and, if they are equal, writes the value v_2 to that address. In either case, the comparison result is written to the local register r. `write` and `read` (as described in rules WRITE and READ) transfer values from local to global storage and vice versa. Finally, the rules CON-LEFT and CON-RIGHT define the interleaving semantics: Whenever a component can take a step, the combination can take it, too. There is *no* synchronous step. In the next section, we consider how to relate the `comm` instruction and its implementation based on a simple handshake protocol.

$$\frac{(\sigma_{pc}, \texttt{do}\ f) \in \mathit{code} \qquad \sigma'_{rs} \in f(\sigma_{rs}) \qquad \sigma'_{pc} = \sigma_{pc} + 1}{(\Gamma, \sigma) \xrightarrow{\mathit{code}} (\Gamma, \sigma')}\ \text{DO}$$

$$\frac{(\sigma_{pc}, \texttt{cbr}\ b\ m\ n) \in \mathit{code} \qquad \sigma'_{rs} = \sigma_{rs} \qquad b\ \sigma \wedge \sigma'_{pc} = m \vee \neg b\ \sigma \wedge \sigma'_{pc} = n}{(\Gamma, \sigma) \xrightarrow{\mathit{code}} (\Gamma, \sigma')}\ \text{CBR}$$

$$\frac{\begin{array}{c}(\sigma_{pc}, \texttt{cas}\ r\ sv\ v_1\ v_2) \in \mathit{code} \\ \Gamma(sv) = v_1 \qquad \Gamma' = \Gamma(sv := v_2) \qquad \sigma'_{rs} = \sigma_{rs}(r := \top) \qquad \sigma'_{pc} = \sigma_{pc} + 1\end{array}}{(\Gamma, \sigma) \xrightarrow{\mathit{code}} (\Gamma', \sigma')}\ \text{CAS-T}$$

$$\frac{(\sigma_{pc}, \texttt{cas}\ r\ sv\ v_1\ v_2) \in \mathit{code} \quad \Gamma(sv) \neq v_1 \quad \sigma'_{rs} = \sigma_{rs}(r := \bot) \quad \sigma'_{pc} = \sigma_{pc} + 1}{(\Gamma, \sigma) \xrightarrow{\mathit{code}} (\Gamma, \sigma')}\ \text{CAS-F}$$

$$\frac{(\sigma_{pc}, \texttt{write}\ sv\ x) \in \mathit{code} \quad \Gamma' = \Gamma(sv := \sigma_{rs}(x)) \quad \sigma'_{rs} = \sigma_{rs} \quad \sigma'_{pc} = \sigma_{pc} + 1}{(\Gamma, \sigma) \xrightarrow{\mathit{code}} (\Gamma', \sigma')}\ \text{WRITE}$$

$$\frac{(\sigma_{pc}, \texttt{read}\ x\ sv) \in \mathit{code} \qquad \sigma'_{rs} = \sigma_{rs}(x := \Gamma(sv)) \qquad \sigma'_{pc} = \sigma_{pc} + 1}{(\Gamma, \sigma) \xrightarrow{\mathit{code}} (\Gamma, \sigma')}\ \text{READ}$$

$$\frac{(\Gamma, \sigma_1) \xrightarrow{c_1} (\Gamma', \sigma'_1)}{(\Gamma, \sigma_1 \parallel \sigma_2) \xrightarrow{c_1 \| c_2} (\Gamma', \sigma'_1 \parallel \sigma_2)}\ \text{CON-LEFT} \qquad \frac{(\Gamma, \sigma_2) \xrightarrow{c_2} (\Gamma', \sigma'_2)}{(\Gamma, \sigma_1 \parallel \sigma_2) \xrightarrow{c_1 \| c_2} (\Gamma', \sigma_1 \parallel \sigma'_2)}\ \text{CON-RIGHT}$$

Fig. 2: Operational Semantics for SV

4.1 Handshake Protocol in SV

In SV, many protocols realizing synchronous communication can be implemented. In this paper, we focus on a handshake protocol over shared variables.

send:		receive:	
1:	**cas** hl_c m_c FREE id	1:	**cas** ss_c sr_c $\top$ $\bot$
2:	**cbr** hl_c 3 1	2:	**cbr** ss_c 3 1
3:	**write** sv_c x_s	3:	**read** x_r sv_c
4:	**write** sr_c $\top$	4:	**write** fr_c $\top$
5:	**cas** ss_c fr_c $\top$ $\bot$		
6:	**cbr** ss_c 7 5		
7:	**write** m_c FREE		

Fig. 3: Send and receive: implementations of the $\texttt{comm}_s$ and $\texttt{comm}_r$ instructions

$\texttt{comm}_s$ and $\texttt{comm}_r$ are implemented in SV by the constructs shown in Figure 3 with a simple handshake protocol. The general idea is that *send* locks the channel to protect the shared variable, and synchronizes over signals with *receive*. The protocol flow is illustrated in Figure 5. The shared variables representing the mutex and the signals are assumed to be exclusive for each channel.[1] We explain the details of the implementations of the sender and the receiver line by line:

[1] That mutexes and signals are only accessed from the corresponding *send* and *receive* blocks can be checked syntactically.

send (1) checks if the mutex m_c belonging to the channel is free, and if it is, writes its *id* to it. (2) If it is not free, it checks again (busy loop). Otherwise it proceeds to (3) write the data value to be sent (from the local register x_s) to the shared variable sv_c. Afterwards, it realizes a synchronization with the read process: It (4) sets the signal sr_c. Then it (5, 6) waits with a busy loop for the signal fr_c and finally (7) releases the mutex.
receive (1,2) waits with a busy loop for the signal sr_c. If it received the signal, it (3) reads the value from the shared variable and then (4) sets the signal fr_c.

Observe that deadlocks from CUC that are due to missing communication partners are implemented as spinlocks in SV: *send* cannot exit the busy loop (line 5, 6) without a receiver on the same channel, and *receive* cannot exit the loop (lines 1, 2) without a sender in the channel.

4.2 Definitions to Relate `comm` and its Implementations

To formally capture that a CUC and an SV program are syntactically the same apart from the implementation of the abstract communication, we define the *program label map*. As the implementation of the abstract communication is inserted, the following labels shift accordingly. We use this definition to define the notion of an SV program *fitting* a CUC program.

Definition 1 (Program label map). *A program label map ψ maps injectively a program label in a CUC program cuc to a corresponding program label in an SV program sv. For the formal requirements to ψ see Figure 4.*

$$
\begin{aligned}
(\ell, \texttt{do}\ f) \in cuc^{id} &\iff \big(\psi(\ell), \texttt{do}\ f\big) \in sv^{id} \wedge \psi(\ell+1) = \psi(\ell) + 1 \\
(\ell, \texttt{cbr}\ b\ m\ n) \in cuc^{id} &\iff \big(\psi(\ell), \texttt{cbr}\ b\ \psi(m)\ \psi(n)\big) \in sv^{id} \\
(\ell, \texttt{comm}_s\ id\ c\ x_s) \in cuc^{id} &\iff \big(\psi(\ell)+0, \texttt{cas}\ m_c\ \text{FREE}\ id\big) \in sv^{id} \\
&\ \ \vdots \qquad\qquad\qquad\quad \vdots \qquad\qquad\qquad\quad \vdots \\
&\ \ '' \qquad\qquad \iff \big(\psi(\ell)+6, \texttt{write}\ m_c\ \text{FREE}\big) \in sv^{id} \\
&\ \ '' \qquad\qquad \implies \psi(\ell+1) = \psi(\ell) + 7 \\
(\ell, \texttt{comm}_r\ id\ c\ x_r) \in cuc^{id} &\iff \big(\psi(\ell)+0, \texttt{cas}\ sr_c\ \top\ \bot\big) \in sv^{id} \\
&\ \ \vdots \qquad\qquad\qquad\quad \vdots \qquad\qquad\qquad\quad \vdots \\
&\ \ '' \qquad\qquad \iff \big(\psi(\ell)+3, \texttt{write}\ fr_c\ \top\big) \in sv^{id} \\
&\ \ '' \qquad\qquad \implies \psi(\ell+1) = \psi(\ell) + 4
\end{aligned}
$$

Fig. 4: Requirements to a program label map ψ

Definition 2 (Fitting program). *We say that an SV program sv* **fits** *a CUC program cuc, if there is a program label map ψ, mapping all the instructions from cuc to sv. Furthermore, we require the state transforming functions f of* `do` f *to only modify the variables available in cuc (i. e. not hl_c and ss_c). Similarly, the boolean conditions b of* `cbr` *instructions in cuc may only depend on variables present in cuc.*

Channel constituents group all variables that belong to a channel.

Definition 3 (Channel constituents). *The following local registers* **belong** *to a channel c: hl_c and ss_c. The following shared variables* **belong** *to a channel c: m_c, sv_c sr_c, and fr_c.*

In the following, we assume that channel constituents are unique for each channel. The registers belonging to a channel are exactly the registers that are present in *sv* but not in *cuc*. Thus, when comparing a local state of *cuc* and *sv*, we ignore those registers. We can now define *similarity* of local state, which we use to relate CUC states and SV states.

Definition 4 (Similarity w.r.t channel constituents). *Let $\sigma \mathrel{\widehat{=}} \hat{\sigma}$ denote that σ and $\hat{\sigma}$ are equal for all registers that do not belong to a channel. This equality also does not include the program counter. We say σ is* **similar** *to $\hat{\sigma}$.*

Note that $\mathrel{\widehat{=}}$ *does* include the register into which *receive* writes the value read from the shared variable, thus receiving a value is *visible* to the $\mathrel{\widehat{=}}$ relation.

Having defined CUC, SV and the protocol we want to verify, in the next section we define our notion of handshake refinement to formally relate CUC and SV programs, ensuring that safety and liveness properties are preserved.

5 Handshake Refinement

The idea of the *handshake refinement* is to extend usual behavioral relations of two states or processes (as in bisimulations or refinements) with a third element (the lockstate) to track the progress of the protocol execution. This enables different treatment in the relation of the same CUC state at different stages of the protocol execution. We use it to indicate which possible events of the CUC state need to be answered by the SV state. The lockstate L is a function from channel names to $\{\text{FREE}\} \uplus ID_{in} \uplus (ID \times ID)_{in} \uplus (ID \times ID)_{un} \uplus ID_{un}$. Every channel has one of five states: It can be FREE, a sender or both a sender and a receiver are in the channel, and after the communication happened, the channel will be eventually unlocked, first with both a sender and a receiver still in the channel, then only a sender. The states of the lockstate within the protocol flow are illustrated in Figure 5 in the rectangular boxes. For each channel, the SV states and possible transitions of *send* (S, S1 to S6; on the left) and *receive* (R, R1 to R3; on the right) are depicted, and in the upper right corner also those of `do` (D) and `cbr` (C), as well as those pointing outside the code (O). N is a placeholder for O, D, C, S, or R. Dotted lines indicate the transition to the next lockstate. The dashed line marks the moment where the communication happens, i. e. all states above are in a relation to the CUC state before the communication, and those below to the CUC state after the communication has happened. The arrows over (S1), (S5'), and (R2) denote whether `cbr` will jump back to the first label or forward to the second label, based on the `cas` instruction before. Note that the transitions of *send* from S4 to S4' and S5 to S5' happen without a step from the sending component, but correspond to the transition of *receive* on the same

channel from R2 to R3. We define the following shorthands for the lockstate: $id \notin L \coloneqq \forall\, c.\ L(c) \neq id_{in/un} \wedge \big(\forall\, id'.\ L(c) \neq (id, id')_{in/un}\big) \wedge L(c) \neq (id', id)_{in/un}\big)$ and $L = \emptyset \coloneqq \forall\, c.\ L(c) = \text{FREE}$.

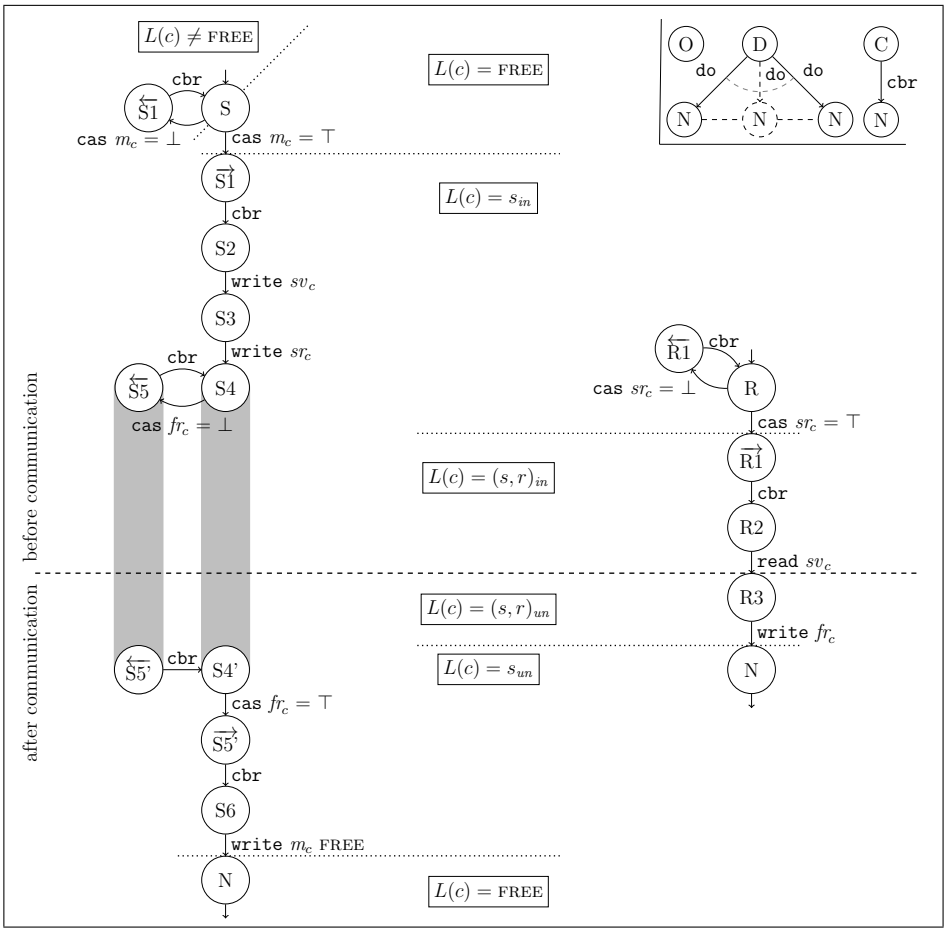

Fig. 5: The flow of the handshake protocol

To define the stable failures on SV independently of the handshake refinement, we cannot use the lockstate. In contrast to the vertical bisimulation [10], the visible event in the protocol implementation is never at the beginning of the implementation (and cannot be, as neither sender nor receiver are determined at the beginning). To overcome this problem, we introduce a special event for the invisible steps of the implementation. As the "usual" invisible event τ is already included (`do`, `cbr`), we denote the invisible instructions of the implementation of communication with τ_c. This way, we can define stable states *before* the execution of the protocol implementation, but let the refusal sets refer to events *during*

the execution of the protocol implementation. This enables us to bridge the gap between abstract synchronous semantics, where the event coincides with both decision points, and the low-level asynchronous semantics, where the event happens after the second decision. To define stable failures semantics for SV, we define a labeling function mapping transitions in sv to events. Transitions are identified by the starting state and the executed instruction. Visible events are only mapped to `read`, τ_c to the invisible instructions of the implementation of the communication. All other instructions (`do` and `cbr`) are invisible with the usual τ.

Definition 5 (Event labeling for sv). *Let EL be a function from state, id of the component executing the next instruction, and its next instruction to events of cuc, τ, or τ_c.*

$$\begin{aligned} EL\big((\Gamma, \_), id, \texttt{read}\ \_\ sv_c\big) &\mapsto c.s.r.v \text{ where } s = \Gamma(mutex_c), r = id, v = \Gamma(sv_c) \\ EL(\_, \_, ins) &\mapsto \tau_c \text{ if } ins \text{ is part of } \text{send} \text{ or } \text{receive} \text{ (see Fig. 3)} \\ EL(\_, \_, \_) &\mapsto \tau \qquad \text{otherwise} \end{aligned}$$

Using the labeling function EL, we can derive SV semantics with visible events:

Definition 6 (SV semantics with events).

$$(\Gamma, \sigma) \xrightarrow{ev}_{sv} (\Gamma', \sigma') :\Leftrightarrow (\Gamma, \sigma) \xrightarrow{sv} (\Gamma', \sigma') \wedge \big(\exists\, id\, ins.\, ev = EL\big((\Gamma, \sigma), id, ins\big)\big)$$

Here, the active component id is determined by the component whose program counter changed, and ins is the instruction the program counter of the active component points to. To ensure that every executed instruction changes the program counter, we require that no `cbr` instruction jumps to its own label.

$\forall\, \big(\sigma, L, (\Gamma, \hat{\sigma})\big) \in \mathcal{B}_{cuc,sv,\psi}.$ a can be visible or τ

Similar local states: $\sigma \mathrel{\widehat{=}} \hat{\sigma}$

Protocol constraints: $\mathcal{P}_{cuc,sv,\psi}\big(\sigma, L, (\Gamma, \hat{\sigma})\big)$ (see Figure 7)

Down-simulation:

$\forall\, a\ \sigma'.\ a \neq \tau \wedge L(chan(a)) = \text{FREE} \wedge \sigma \xrightarrow{a}_{cuc} \sigma' \Longrightarrow$
$\exists\, \Gamma'\ \hat{\sigma}'\ id_s\ id_r\ L'.\ (\Gamma, \hat{\sigma}) \xrightarrow{\tau_c}{}^{*}_{sv} \xrightarrow{a}_{sv} (\Gamma', \hat{\sigma}') \wedge L'(chan(a)) = (id_s, id_r)_{un} \wedge$
$\big(\sigma', L', (\Gamma', \hat{\sigma}')\big) \in \mathcal{B}_{cuc,sv,\psi}$

$\forall\, \sigma'.\ \sigma \xrightarrow{\tau}_{cuc} \sigma' \Longrightarrow$
$\exists\, \Gamma'\ \hat{\sigma}'\ L'.\ (\Gamma, \hat{\sigma}) \xrightarrow{\tau_c}{}^{*}_{sv} \xrightarrow{\tau}_{sv} (\Gamma', \hat{\sigma}') \wedge \big(\sigma', L', (\Gamma', \hat{\sigma}')\big) \in \mathcal{B}_{cuc,sv,\psi}$

Up-simulation:

$\forall (\Gamma', \hat{\sigma}').\ (\Gamma, \hat{\sigma}) \xrightarrow{\tau_c}_{sv} (\Gamma', \hat{\sigma}') \Longrightarrow \exists\, L'.\ \big(\sigma, L', (\Gamma', \hat{\sigma}')\big) \in \mathcal{B}_{cuc,sv,\psi}$
$\forall\, a\ (\Gamma', \hat{\sigma}').\ (\Gamma, \hat{\sigma}) \xrightarrow{a}_{sv} (\Gamma', \hat{\sigma}') \Longrightarrow \exists\, \sigma'\ L'.\ \sigma \xrightarrow{a}_{cuc} \sigma' \wedge \big(\sigma', L', (\Gamma', \hat{\sigma}')\big) \in \mathcal{B}_{cuc,sv,\psi}$

Unlocking-simulation:

$\exists\, c\ id_s.\ L(c) = (id_s)_{un} \vee \big(\exists\, id_r.\ L(c) = (id_s, id_r)_{un}\big) \Longrightarrow$
$\exists\, \Gamma'\ \hat{\sigma}'\ L'.\ (\Gamma, \hat{\sigma}) \xrightarrow{\tau_c}{}^{*}_{sv} (\Gamma', \hat{\sigma}') \wedge L' = L(c := \text{FREE}) \wedge \big(\sigma, L', (\Gamma', \hat{\sigma}')\big) \in \mathcal{B}_{cuc,sv,\psi}$

Fig. 6: Handshake Refinement

$\mathcal{P}_{cuc,sv,\psi}(\sigma, L, (\Gamma, \hat{\sigma})) \coloneqq (L(c){=}\text{FREE} \Longrightarrow \neg\Gamma(sr_c) \wedge \neg\Gamma(fr_c)) \wedge \forall\, id. \mathcal{P}^{id}_{cuc,sv,\psi}(\sigma, L, (\Gamma, \hat{\sigma}))$
$\mathcal{P}^{id}_{cuc,sv,\psi}(\sigma, L, (\Gamma, \hat{\sigma})) \coloneqq O \vee D \vee C \vee S \vee S1 \vee S2 \vee S3 \vee S4 \vee S5 \vee S4' \vee S5' \vee S6 \vee R \vee R1 \vee R2 \vee R3$

O, D, C Have a direct counterpart in CUC, channel variables are not a concern, $id \notin L$

D do f instruction

C cbr

S At the beginning of *send*, $id \notin L$

S1 Branch according to result of `cas` in S. If the component now has the mutex, than also the signals must be inactive.

S2 From now on in this execution of the protocol, the id of the component is in the mutex and in the lockstate.

S3 The data value to be communicated is in the shared variable.

S4 Start reading was set to $\top$ from S3 to S4. If the receiver did start, then start reading will remain $\bot$ from now on. In the first case the lockstate only contains the sender, in the second also the receiver. The first row of the formula ensures, that the SV state is mapped to a CUC state where the pc points to the appropriate `comm`.

$$(\sigma^{id}_{pc}, \texttt{comm}_s\, id\, c\, x_s) \in cuc^{id} \wedge (\hat{\sigma}^{id}_{pc}, \texttt{cas}\, ss_c\, fr_c\, \top\, \bot) \in sv^{id} \wedge \psi(\sigma^{id}_{pc}) + 4 = \hat{\sigma}^{id}_{pc}$$
$$\wedge\, \Gamma(m_c) = id \wedge \Gamma(sv_c) = \hat{\sigma}^{id}_{rs}(x_s) \wedge \neg\Gamma(fr_c)$$
$$\wedge\, \big(\Gamma(sr_c) \wedge L(c) = id_{in} \vee \neg\Gamma(sr_c) \wedge (\exists\, id_r.\, L(c) = (id, id_r)_{in})\big)$$

S5 Branch back to S4, as the communication has not happened yet.

S4' From now on, the communication already has happened. The lockstate is now set to unlocking. Observe, that now the SV state is in a relation with the CUC state that occurs after the communication. Therefore we need to substract 1 from the pc of the SV state, to map with ψ to `comm`.

$$(\sigma^{id}_{pc} - 1, \texttt{comm}_s\, id\, c\, x_s) \in cuc^{id} \wedge (\hat{\sigma}^{id}_{pc}, \texttt{cas}\, ss_c\, fr_c\, \top\, \bot) \in sv^{id} \wedge \psi(\sigma^{id}_{pc} - 1) + 4 = \hat{\sigma}^{id}_{pc}$$
$$\wedge\, \Gamma(m_c) = id \wedge \neg\Gamma(sr_c)$$
$$\wedge\, \big(\Gamma(fr_c) \wedge L(c) = id_{un} \vee \neg\Gamma(fr_c) \wedge (\exists\, id_r. L(c) = (id, id_r)_{un})\big)$$

S5' Branch according to the result of `cas` in S4'.

S6 The signals are $\bot$, in the next step the mutex and the lockstate will be free.

R At the beginning of *receive*, $id \notin L$

R1 Branch according to result of `cas` in R. If the component is now a receiver, both sender and receiver ids are in the lockstate of the channel. The state of the signals is already fixed in the disjunct of the sender where both are in the lockstate.

R2 The lockstate contains the sender and the receiver about to communicate.

R3 The lockstate still contains the sender and the receiver, but now about to unlock the channel. The SV state is now in a relation with the CUC state after the communication.

Fig. 7: Protocol restrictions

We define the *handshake refinement* in Figure 6. It is a relation parametrized over two programs *cuc* and *sv* fitting with ψ. The elements are triplets consisting of a parallel CUC state σ, a lockstate L, and pair of global state Γ and parallel local SV states $\hat{\sigma}$. Our *handshake refinement* consists of two properties describing the states, and three describing the possible transitions. In each triplet, the CUC states and the local SV states are *similar*. Furthermore, they fulfill the protocol constraints $\mathcal{P}_{cuc,sv,\psi}$, which constrain the possible SV states and their relation to CUC states. $\mathcal{P}_{cuc,sv,\psi}$ is defined in Figure 7 and explained below. The

possible transitions are described by the down-, up-, and unlocking-simulation. The **down-simulation** relates transitions in CUC to one or more transitions in SV. Observe that visible events only need to be answered, if the channel is FREE. This precludes triplets where the sender in SV is already decided but the CUC state still could choose a different sender. It is sound to ignore those SV states in the down-simulation, as we are only interested if the implementation (as a whole) allows and offers the same events. Although there is no "equivalent" state in CUC, all other senders were possible right before this choice of a particular sender, so we do not ignore different possible events, only the intermediate states. The **up-simulation** relates transitions in SV to transitions in CUC. τ_c events are related to zero transitions in CUC, all other events to one. Finally, the **unlocking-simulation** ensures (under simple fairness) that, after the communication has happened, the channel will be freed eventually. This allows the down-simulation to only consider states, where the channel is free. In the remainder of this paper, let $\mathcal{B}_{cuc,sv,\psi}$ denote a *handshake refinement*.

Figure 7 describes the protocol constraints $\mathcal{P}_{cuc,sv,\psi}$, which are specific to the handshake protocol at hand. They also ensure that only SV states reachable by protocol execution are included and that the lockstate reflects the current progress of the protocol execution. The overall definition is that for every channel, if the lockstate is FREE, the belonging signals must be $\bot$, and for each component, the disjunction $\mathcal{P}^{id}_{cuc,sv,\psi}$ must hold. The disjuncts describe triplets (cuc, L, sv), providing sufficient conditions to the SV state (program counter and channel related variables) and relating them to a CUC state (program counter) via ψ with the appropriate lockstate. In $\mathcal{P}^{id}_{cuc,sv,\psi}$, the lockstate also "synchronizes" the different components, i. e., excludes illegal state combinations. It follows a description of the disjuncts, from which we provide two formally. [2]

6 Preservation of Safety and Liveness Properties

In this section, we prove that our *handshake refinement* preserves safety and liveness properties of the considered CUC program *cuc* to a fitting SV program *sv*. This implies that *sv* only has behavior allowed by *cuc* (safety), and also preserves the progress (liveness). To this end, we define traces and stable failures semantics for both CUC and SV via an operational characterization and then show the stable failures refinement between *cuc* and *sv*. First, we define traces both for CUC and SV.

Definition 7 (Trace semantics). *We write $P \overset{tr}{\Rightarrow}_{cuc/sv} Q$ to describe that there is an execution path from P to Q in cuc/sv, and during that execution the visible events in tr occur exactly in that order. We call tr the trace from P to Q over cuc/sv. Let $\mathcal{T}(P)_{cuc/sv}$ be all traces starting in P over cuc/sv.*

Not all possible SV states are legal in a *handshake refinement*, i. e., not all states are reachable by execution of the handshake protocol. We consider SV states

[2] For a complete formal version we refer to our Technical Report [2].

$(\Gamma_0, \hat{\sigma}_0)$ as *initial states*, if all components of $\hat{\sigma}_0$ only point to the first instruction of *send* or *receive* (or the second, which is `cbr`, if it jumps back) and all mutexes are FREE and the signals are inactive ($\bot$). An empty lockstate in the *handshake refinement* $\big(\sigma_0, \emptyset, (\Gamma_0, \hat{\sigma}_0)\big) \in \mathcal{B}_{cuc,sv,\psi}$ implies those properties. Using induction on the up-simulation, we can show that every trace in $\mathcal{T}(\Gamma_0, \hat{\sigma}_0)_{sv}$ leads to a triplet in $\mathcal{B}_{cuc,sv,\psi}$ and the same trace is in $\mathcal{T}(\sigma_0)_{cuc}$ leading to the same triplet:

Lemma 1 (All *sv* traces and their *cuc* counterparts are in $\mathcal{B}_{cuc,sv,\psi}$).

$$\big(\sigma_0, \emptyset, (\Gamma_0, \hat{\sigma}_0)\big) \in \mathcal{B}_{cuc,sv,\psi} \wedge (\Gamma_0, \hat{\sigma}_0) \overset{tr}{\Longrightarrow}_{sv} (\Gamma, \hat{\sigma})$$
$$\Longrightarrow \exists\, \sigma\ L'.\ \big(\sigma, L', (\Gamma, \hat{\sigma})\big) \in \mathcal{B}_{cuc,sv,\psi} \wedge \sigma_0 \overset{tr}{\Longrightarrow}_{cuc} \sigma$$

We can directly conclude the preservation of safety properties:

Theorem 1 (Preservation of safety properties).

$$\big(\sigma_0, \emptyset, (\Gamma_0, \hat{\sigma}_0)\big) \in \mathcal{B}_{cuc,sv,\psi} \Longrightarrow \mathcal{T}(\Gamma_0, \hat{\sigma}_0)_{sv} \subseteq \mathcal{T}(\sigma_0)_{cuc}$$

Having shown that our *handshake refinement* preserves safety properties, we proceed to show that it also preserves liveness properties. We capture liveness properties using the notion of stable failures (inspired by CSP). To this end, we define the notions of stable states and refusal sets to finally define the stable failures, both for CUC and SV. We then show that the stable failures of *sv* are included in the stable failures of *cuc*. Thus, all liveness properties from *cuc* are preserved in *sv*. A state is stable if no internal transition is possible.

Definition 8 (Stable states in *cuc*). *A state σ is* **stable** *in cuc ($\sigma{\downarrow_{cuc}}$) if all components either point outside the code, to $\texttt{comm}_s$, or to $\texttt{comm}_r$. Formally:*

$$\sigma{\downarrow_{cuc}} := \forall\, id.\ \big(\nexists\, ins.\ (\sigma(id)_{pc}, ins) \in cuc(id)\big) \vee$$
$$\big(\exists\, c.\ (\sigma(id)_{pc}, \texttt{comm}_s\ id\ c\ x_s) \in cuc(id) \vee (\sigma(id)_{pc}, \texttt{comm}_r\ id\ c\ x_r) \in cuc(id)\big)$$

Refusal sets and stable failures are defined similarly to their CSP counterparts.

Definition 9 (Refusal set in *cuc*). *A state σ* **refuses** *a set of visible events X in cuc, if it cannot perform any $a \in X$. Let $X \subseteq \Sigma$.*

$$\sigma\ \mathrm{ref}_{cuc}\ X := \forall\, a \in X.\ \neg\big(\sigma \xrightarrow{a}_{cuc}\big)$$

Definition 10 (Stable failures of *cuc*). *A* **stable failure** *is a pair of a trace tr and a refusal set X. It denotes that there is a stable state σ which can be reached from the initial state init via the trace tr and refuses X.*

$$(tr, X) \in \mathcal{SF}_{cuc}(init) := \exists\, \sigma.\ init \overset{tr}{\Longrightarrow}_{cuc} \sigma \wedge \sigma{\downarrow_{cuc}} \wedge \sigma\ \mathrm{ref}_{cuc}\ X$$

Next, we define stable states, refusal sets, and stable failures for *sv*. The stable states and failures are similar to the definitions for *cuc*. The refusal sets differ, as they need to account for the invisible execution steps of the handshake protocol.

Definition 11 (Stable states in *sv*). *A state $(\Gamma, \hat{\sigma})$ is* **stable** *in sv ($(\Gamma, \hat{\sigma}){\downarrow_{sv}}$) if all components either point outside the code or to the first instruction of* send *or* receive*. Formally:*

$$(\Gamma, \hat{\sigma}){\downarrow_{sv}} := \forall\, id.\ \big(\nexists\, ins.\ (\hat{\sigma}(id)_{pc}, ins) \in sv(id)\big) \vee$$
$$\big(\exists\, c.\ (\hat{\sigma}(id)_{pc}, \texttt{cas}\ m_c\ \mathrm{FREE}\ id) \in sv(id) \vee (\hat{\sigma}(id)_{pc}, \texttt{cas}\ sr_c\ \top\ \bot) \in sv(id)\big)$$

The stable states in *sv* coincide with the stable states in *cuc* (pointing to $\texttt{comm}_s$, $\texttt{comm}_r$ or outside of the code). They can neither make a visible event step nor a τ step, but might be able to make a τ_c step. As the visible event (i. e. `read`) occurs only in the middle of the execution of the handshake protocol, a finite number of τ_c-steps is allowed before the visible event to consider it "enabled". Assuming fairness, i. e., at any point for any component, there is a finite number of steps after which the component will make a step, possible communication happens after a finite number of τ_c-steps. Conversely, if communication is not possible, i. e., a deadlock occurs in the synchronous setting, the implementation of the handshake protocol will stay in a busy loop, thus the visible event is not reachable.

Definition 12 (Refusal set in *sv*). *A state* **refuses** *a set of visible events in sv, if they are not reachable after a finite number of τ_c steps. Let $X \subseteq \Sigma$.*

$$P \operatorname{ref}_{sv} X \coloneqq \forall\, a \in X.\ \neg\big(P \xrightarrow{\tau_c}{}^{*}_{sv} \xrightarrow{a}_{sv}\big)$$

Definition 13 (Stable failures of SV). *A* **stable failure** *is a pair of a trace tr and a refusal set X. It denotes that there is a stable state $(\Gamma, \hat{\sigma})$ which can be reached from the initial state init via the trace tr and refuses X.*

$$(tr, X) \in \mathcal{SF}_{sv}(init) \coloneqq \exists(\Gamma, \hat{\sigma}).\ init \overset{tr}{\Longrightarrow}_{sv} (\Gamma, \hat{\sigma}) \wedge (\Gamma, \hat{\sigma})\downarrow_{sv} \wedge (\Gamma, \hat{\sigma}) \operatorname{ref}_{sv} X$$

To show the preservation of liveness properties, we first show two lemmas: That stable states in *sv* imply stable states in *cuc*, and the key lemma, that refusals of *sv* imply refusals of *cuc*.

Lemma 2 (Stable states in *sv* imply stable states in *cuc* and $L = \emptyset$).

$$\big(\sigma, L, (\Gamma, \hat{\sigma})\big) \in \mathcal{B}_{cuc,sv,\psi} \wedge (\Gamma, \hat{\sigma})\downarrow_{sv} \Longrightarrow \sigma\downarrow_{cuc} \wedge\ L = \emptyset$$

Proof. As $\mathcal{B}_{cuc,sv,\psi}$ is a *handshake refinement*, $\mathcal{P}_{cuc,sv,\psi}\big(\sigma, L, (\Gamma, \hat{\sigma})\big)$ holds. In $\mathcal{P}_{cuc,sv,\psi}$ the cases where $(\Gamma, \hat{\sigma})\downarrow_{sv}$ holds imply $\sigma\downarrow_{cuc}$ and $L = \emptyset$.

Lemma 3 (Refusals in *sv* imply refusals in *cuc*).

$$\big(\sigma, L, (\Gamma, \hat{\sigma})\big) \in \mathcal{B}_{cuc,sv,\psi} \wedge (\Gamma, \hat{\sigma})\downarrow_{sv} \Longrightarrow (\Gamma, \hat{\sigma}) \operatorname{ref}_{sv} X \Longrightarrow \sigma \operatorname{ref}_{cuc} X$$

Proof. Using Lemma 2, we can apply the down-simulation.[3]

Theorem 2 (Preservation of liveness properties).

$$\big(\sigma_0, \emptyset, (\Gamma_0, \hat{\sigma}_0)\big) \in \mathcal{B}_{cuc,sv,\psi} \Longrightarrow \mathcal{SF}_{sv}(\Gamma_0, \hat{\sigma}_0) \subseteq \mathcal{SF}_{cuc}(\sigma_0)$$

Proof. Using the Lemmas 1, 2, and 3.[3]

Corollary 1 (Liveness properties without sender ID). *An adaption of the handshake protocol given in Figure 3, where in the mutex only* TAKEN *is stored instead of the sender id, also preserves all safety and liveness properties.*

Proof. As the behavior of the protocol does not depend on the sender id being stored in the mutex, only whether the mutex is FREE or not, the behavior of the original and adapted protocols is the same, thus also the same properties are preserved. Note that the information about the sender is only needed for the proofs to reconstruct who the sender was, when the receiver reads the value. □

[3] A more detailed proof can be found in our Technical Report [2].

7 Handshake Refinement for Fitting Programs

In this section, we show that any *cuc* program and fitting *sv* program are in a *handshake refinement* relation. More specifically, we show that all sensible initial states are in a *handshake refinement* relation. This general theorem allows for a scalable approach to the verification of shared variable communication. The proof sketch can be found in [2] and is similar to bisimilarity proofs: all possible transitions of one part can be answered by its counterpart. An important difference is that the down-simulation needs to be shown ("has to answer") *only* in stable states, due to it being a refinement and not a bisimulation.

Theorem 3 (Fitting implies *handshake refinement*). *Let sv be a program fitting cuc with ψ. Then there is a* handshake refinement $\mathcal{B}_{cuc,sv,\psi}$ *containing all initial pairs, i. e., similar CUC and SV states where the program counters of each component match with ψ, all mutexes in Γ are* FREE*, all signals inactive.*

As the *handshake refinement* implies preservation of safety (Theorem 1) and liveness (Theorem 2) properties, we can now conclude with Theorem 3, that all fitting programs preserve safety and liveness properties:

Theorem 4 (Fitting implies preservation). *Let sv be a program fitting cuc with ψ. Then all safety and liveness properties from cuc are preserved to sv.*

8 Conclusion

In this paper, we have presented a method to relate abstract synchronous communication with an asynchronous handshake implementation using shared variable communication and have proved that this relation preserves safety and liveness properties. To this end, we have introduced our novel notion of *handshake refinement*, which is similar to strong bisimulation, apart from the protocol implementation, which is a refinement. It explicitly captures the state of progression through the executions of the implementations of the protocol. Moreover, we have proved in the general Theorem 4, that *all* pairs of CUC and SV programs, where the SV program results from the CUC program by replacing the abstract communication instructions with their handshake implementation, have the same safety and liveness properties. Together with a compositional method to show safety and liveness properties for CUC programs [8], we have a *compositional* framework to prove the preservation of safety and liveness properties from abstract specifications in CSP to down to low-level code, including asynchronous communication mechanisms.

Although we have presented our method for a concrete (handshake) protocol, it provides the foundation for a more generalized notion of relations between abstract synchronous and concrete asynchronous communication based on other communication/synchronization protocols. The presented protocol can be divided into four phases (which match with the four non-FREE lockstates): *1)* registration, *2)* before communication, *3)* after communication, *4)* unregistration. This is also the structure the *handshake refinement* relies upon. As the

presented handshake protocol is intentionally simple, the phases are very short. Our approach can be extended to other protocols that fit in those four phases, e. g. to verify a protocol which supports a "selection on channels" (external choice in CSP). This "selection", i. e. finding a channel with a present communication partner, would happen in phase 1. This way, not only input guards, but also output guards could be supported. Overall, we have shown the preservation of liveness properties using the stable failures model. This does not consider livelocks (divergences). However, as the related CUC and SV programs are the same outside of the protocol implementation and jumps do not occur into our out of the protocol implementation, no livelocks are introduced. Inside the protocol implementation, livelocks in SV are only introduced when unsuccessfully waiting for a communication partner, in which case the CUC program was deadlocked, so no progress is eliminated.

In future work, we plan to investigate relations similar to the *handshake refinement* for different communication protocols. We are currently working on formalizing the entire presented approach in the interactive theorem prover Isabelle/HOL to guarantee the correctness of proofs and to enable the reusability of the formalization, e. g. for other protocols.

References

1. Basu, S., Bultan, T., Ouederni, M.: Synchronizability for verification of asynchronously communicating systems. In: VMCAI'12 Proceedings. LNCS, vol. 7148, pp. 56–71. Springer (2012)
2. Berg, N., Göthel, T., Glesner, S., Danziger, A.: Technical Report accompanying: Preserving Liveness Guarantees from Synchronous Communication to Asynchronous Unstructured Low-Level Languages. DepositOnce (2018). `http://dx.doi.org/10.14279/depositonce-7192`
3. Brookes, S.D.: On the relationship of CCS and CSP. In: Automata, Languages and Programming, Proceedings. LNCS, vol. 154, pp. 83–96. Springer (1983)
4. Broy, M., Olderog, R.: Trace-Oriented Models of Concurrency. In: Handbook of Process Algebra, Chapter 2. Elsevier (2001)
5. de Frutos-Escrig, D., Gregorio-Rodríguez, C.: Process Equivalences as Global Bisimulations. In: J. UCS, vol. 12(11), pp. 1521–1550. (2006)
6. Gardner, W.B.: Bridging CSP and C++ with selective formalism and executable specifications. In: MEMOCODE'03 Proceedings. p. 237. IEEE (2003),
7. Jähnig, N., Göthel, T., Glesner, S.: A denotational semantics for communicating unstructured code. In: FESCA'15 Proceedings. EPTCS, vol. 178, pp. 9–21 (2015)
8. Jähnig, N., Göthel, T., Glesner, S.: Refinement-based verification of communicating unstructured code. In: SEFM'16 Proceedings. pp. 61–75 (2016),
9. Peeters, A.M.G.: Implementation of handshake components. In: Communicating Sequential Processes: The First 25 Years, Symposium on the Occasion of 25 Years of CSP, LNCS, vol. 3525, pp. 98–132. Springer (2004),
10. Rensink, A., Gorrieri, R.: Action refinement as an implementation relations. In: TAPSOFT'97 Proceedings. LNCS, vol. 1214, pp. 772–786. Springer (1997)
11. Roscoe, A.W.: Understanding Concurrent Systems. Texts in Computer Science, Springer (2010)