Technical Report accompanying: Preserving Liveness Guarantees from Synchronous Communication to Asynchronous Unstructured Low-Level Languages

Nils Berg Thomas Göthel Armin Danziger Sabine Glesner

TU Berlin

Abstract

This document is an excerpt (Chapter 6) of the dissertation of Nils Berg [Ber19]. It extends the paper *Preserving Liveness Guarantees from Synchronous Communication to Asynchronous Unstructured Low-Level Languages* with detailed proofs and a complete and formal version of the *protocol constraints.*

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.

6 Relating Abstract Communication to Low-Level Protocols

Our second step out of two to relate CSP with a low-level language is to focus on the low-level implementation of abstract communication. To this end, we define the notion of handshake refinement. It is an implementation relation that allows for the implementation of abstract communication while preserving safety and liveness properties. It relates CUC and SV, a generic low-level language we define with communication over shared variables. SV allows for the implementation of various communication protocols. We use a simple handshake protocol to implement the synchronous communication of CSP/CUC with the asynchronous communication instructions provided by SV. Using our notion of handshake refinement, we show that any SV program, which is obtained from a CUC program using the handshake protocol, has the same safety and liveness properties as the initial CUC program. We show this relation in a general theorem for all such pairs of CUC and SV programs. This general theorem allows us to reduce the proof obligations for the relation from CSP to SV to the proof obligations for the relation from CSP to CUC, which we can prove compositionally.

In this chapter, we first present our generic low-level language with communication over shared variables SV in Section 6.1 and then state the handshake protocol in Section 6.2. In Section 6.3, we derive semantics with events for SV from the structure granted by the handshake protocol, in particular a stable failures semantics for SV. We define our notion of handshake refinement in Section 6.4, which allows us to relate the abstract communication in CUC with implementation over shared variables in SV using the presented handshake protocol. In Section 6.5, we show that it preserves safety and liveness properties and finally show that the handshake protocol induces a handshake refinement. As most proofs in this chapter consist of well-known and easy to reproduce techniques, we give concise proofs containing the essential ideas. We published the content of this chapter in [BGDG18].

6.1 Shared Variables (SV)

In this section, we present our generic language *Shared Variables* (SV) and give its syntax and operational semantics. The intent of SV is to have a language with low-level control flow *and* low-level communication. SV has a *pure interleaving* semantics (in contrast to CUC) and allows us to implement synchronous communication over shared variables. SV contains the instructions `do` 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*). We have reasoned in Section 5.1 that we decide to use the instruction `cas` to model multi-processor synchronization (instead of *load-reserve* and *store-conditional*), as it simplifies our proofs and programs and the semantics is similar enough for our use.

6.1.1 Semantic States and Syntax

Although SV is designed to be a generic low-level language that allows for communication via shared variables, it is intentionally similar to CUC. This facilitates the comparison of the semantics of both languages CUC and SV. To allow for shared variable communication, we extend the concurrent local states ($\sigma \in States$) with a global shared state Γ. The global state Γ is modeled as a data store ($\Gamma \in \mathit{DS}$). Thus, it has the same type as the data stores σ_{ds} of the local states.

Definition 6.1: SV State

$$\mathit{GStates} \coloneqq \mathit{DS} \times \mathit{States}$$

The language SV consists of five instructions (two of them stemming from CUC and three new), which we define in Definition 6.2. The first two instructions stem from CUC. The instruction `do` (non-deterministically) transforms the local state, and the instruction `cbr` conditionally branches to one of two jump targets. Both are as in CUC and restricted to interactions with the local state. The three new instructions allow for interaction with the global state: The instructions `read` and `write` transfer data from the shared memory to the local registers and vice versa. The atomic compare-and-set instruction `cas` allows for synchronization via shared variables of multiple concurrent components. We use γ to denote a global variable.

Definition 6.2: Instructions of SV

$$\begin{array}{rll} \mathit{Instructions} \coloneqq & \texttt{do}\ f & f\colon \mathit{DS} \to \mathscr{P}(\mathit{DS}) \\ & \mid \texttt{cbr}\ b\ m\ n & b\colon \mathit{DS} \to \mathbb{B},\ m, n\colon \mathit{Labels} \\ & \mid \texttt{read}\ x\ \gamma & x, \gamma\colon \mathit{Names} \\ & \mid \texttt{write}\ \gamma\ x & \gamma, x\colon \mathit{Names} \\ & \mid \texttt{cas}\ r\ \gamma\ v_1\ v_2 & r, \gamma\colon \mathit{Names},\ v_1, v_2\colon \mathit{Values} \end{array}$$

We skip the explanations for `do` and `cbr`, as they are already explained in Section 5.2. They cannot modify or read from the global state. The following three instructions can modify the global state or read from it.

`read` $x\ \gamma$ reads the value of a shared variable γ into a local register x.

`write` $\gamma\ x$ writes the value of a local register x into a shared variable γ.

`cas` $r\ \gamma\ v_1\ v_2$ compares the value of the shared variable γ with the value v_1. If they are equal, then the value v_2 is written to the shared variable γ. The result of the comparison, i. e., *true* or *false*, is written to the local register r. The instruction `cas` is atomic, i. e., nothing can (concurrently) happen between the comparison and the possible update of the shared variable.

As in CUC, we define a local program *lp* to be a set of labeled instructions (Definition A.3) and a concurrent program *cp* to be a tree of local programs (Definition A.5). We also require the uniqueness of labels (Assumption A.2) and that the tree structure of a concurrent state matches the tree structure of a program (Assumption A.3). We omit to redefine this here, as the definitions and assumptions directly apply. It will be always clear whether we refer to a CUC or an SV program. We do not define a structuring on SV programs, as we relate the operational semantics of CUC and SV.

Having defined semantic states and programs for SV, we proceed to define the operational semantics of SV in the next section.

6.1.2 Semantics

The operational semantics of SV is depicted in Definition 6.3. It contains four kinds of rules: *1)* The single steps concerned only with the local state (DO, CBR), *2)* the single steps interacting with the global state (CAS-T, CAS-F, READ, WRITE), *3)* the concurrent steps (INTERLEAVING-LEFT, INTERLEAVING-RIGHT), and *4)* those for execution (EXEC-0, EXEC). As in CUC, the operational semantics is defined for local programs $lp \in LP$ and concurrent programs $cp \in CP$, respectively.

The single steps concerned with the local steps (DO, CBR) are exactly as in CUC. They leave the global state Γ unchanged.

The single steps interacting with the global state (CAS-T, CAS-F, READ, WRITE) are used for shared variable communication. In CAS-T, the case where compared values are equal ($\Gamma(\gamma) = v_1$) is defined. The shared variable is updated with v_2, and the result of the comparison (*true*) is stored in the register r. The case where the compared values are not equal is defined in CAS-F. Here, the global state remains unchanged. In both cases, the program counter is increased.

In READ and WRITE, the contents of registers are written from the global state to the local state and vice versa. In READ, the global state remains unchanged, in WRITE, the local state remains unchanged apart from the program counter. For both instructions, the program counter is increased.

The concurrent steps in SV (INTERLEAVING-LEFT, INTERLEAVING-RIGHT) realize a pure interleaving semantics of the concurrent combination of the two (possibly concurrent) programs cp_1 and cp_2. Accordingly, INTERLEAVING-LEFT and INTERLEAVING-RIGHT do not have communication interfaces to consider.

The steps for execution (EXEC-0, EXEC-τ) describe the reflexive, transitive hull of all possible single steps.

The language SV is a suitable model for low-level languages: On one hand, it contains only low-level instructions in contrast to CUC, which has an abstract communication instruction. Thus, all instructions of SV can be instantiated in an actual instruction set architecture. On the other hand, its instructions cover the three groups of low-level instructions as described in 5.1. Thus, we can model all concepts of low-level languages. The operational semantics of SV faithfully expresses the synchronization and communication of multiple components (e. g., threads or processes) on a single processor. Every process can read and write from the global memory. The true interleaving semantics ensures that only one component can be active at the same time. In the following, we relate the low-level communication of SV with the abstract communication mechanism of CSP and CUC.

Observe that the semantics is not labeled, i. e., there are no events or traces attached. In contrast to CSP and CUC, where the abstract events correspond to a step in the semantics, in SV a synchronous abstract event can only be obtained by using the structure and information provided by a communication protocol. To formally relate the labeled semantics of CUC and the semantics of SV, we introduce a handshake protocol

Definition 6.3: Operational Semantics of SV

$$\frac{(\sigma_{pc}, \texttt{do}\ f) \in lp \qquad \sigma'_{ds} \in f(\sigma_{ds}) \qquad \sigma'_{pc} = \sigma_{pc} + 1}{(\Gamma, \sigma) \longrightarrow_{lp} (\Gamma, \sigma')}\ \text{DO}$$

$$\frac{(\sigma_{pc}, \texttt{cbr}\ b\ m\ n) \in lp \qquad \sigma'_{ds} = \sigma_{ds} \qquad b\ \sigma \wedge \sigma'_{pc} = m \vee \neg b\ \sigma \wedge \sigma'_{pc} = n}{(\Gamma, \sigma) \longrightarrow_{lp} (\Gamma, \sigma')}\ \text{CBR}$$

$$\frac{\begin{array}{c}(\sigma_{pc}, \texttt{cas}\ r\ \gamma\ v_1\ v_2) \in lp \\ \Gamma(\gamma) = v_1 \qquad \Gamma' = \Gamma(\gamma \coloneqq v_2) \qquad \sigma'_{ds} = \sigma_{ds}(r \coloneqq \mathit{true}) \qquad \sigma'_{pc} = \sigma_{pc} + 1\end{array}}{(\Gamma, \sigma) \longrightarrow_{lp} (\Gamma', \sigma')}\ \text{CAS-T}$$

$$\frac{(\sigma_{pc}, \texttt{cas}\ r\ \gamma\ v_1\ v_2) \in lp \quad \Gamma(\gamma) \neq v_1 \quad \sigma'_{ds} = \sigma_{ds}(r \coloneqq \mathit{false}) \quad \sigma'_{pc} = \sigma_{pc} + 1}{(\Gamma, \sigma) \longrightarrow_{lp} (\Gamma, \sigma')}\ \text{CAS-F}$$

$$\frac{(\sigma_{pc}, \texttt{read}\ x\ \gamma) \in lp \qquad \sigma'_{ds} = \sigma_{ds}(x \coloneqq \Gamma(\gamma)) \qquad \sigma'_{pc} = \sigma_{pc} + 1}{(\Gamma, \sigma) \longrightarrow_{lp} (\Gamma, \sigma')}\ \text{READ}$$

$$\frac{(\sigma_{pc}, \texttt{write}\ \gamma\ x) \in lp \qquad \Gamma' = \Gamma(\gamma \coloneqq \sigma_{ds}(x)) \qquad \sigma'_{ds} = \sigma_{ds} \qquad \sigma'_{pc} = \sigma_{pc} + 1}{(\Gamma, \sigma) \longrightarrow_{lp} (\Gamma', \sigma')}\ \text{WRITE}$$

INTERLEAVING-LEFT

$$\frac{(\Gamma, \sigma_1) \longrightarrow_{cp_1} (\Gamma', \sigma'_1)}{(\Gamma, \sigma_1 \parallel \sigma_2) \longrightarrow_{cp_1 \parallel cp_2} (\Gamma', \sigma'_1 \parallel \sigma_2)}$$

INTERLEAVING-RIGHT

$$\frac{(\Gamma, \sigma_2) \longrightarrow_{cp_2} (\Gamma', \sigma'_2)}{(\Gamma, \sigma_1 \parallel \sigma_2) \longrightarrow_{cp_1 \parallel cp_2} (\Gamma', \sigma_1 \parallel \sigma'_2)}$$

$$\frac{}{(\Gamma, \sigma) \Longrightarrow_{cp} (\Gamma, \sigma)}\ \text{EXEC-0}$$

$$\frac{(\Gamma, \sigma) \Longrightarrow_{cp} (\Gamma'', \sigma'') \qquad (\Gamma'', \sigma'') \longrightarrow_{cp} (\Gamma', \sigma')}{(\Gamma, \sigma) \Longrightarrow_{cp} (\Gamma', \sigma')}\ \text{EXEC-}\tau$$

```
send:
  1:  cas $hl_c$ $m_c$ FREE $id$
  2:  cbr $hl_c$ 3 1
  3:  write $\gamma_c$ $x_s$
  4:  write $sr_c$ $\top$
  5:  cas $ss_c$ $fr_c$ $\top$ $\bot$
  6:  cbr $ss_c$ 7 5
  7:  write $m_c$ FREE

receive:
  1:  cas $ss_c$ $sr_c$ $\top$ $\bot$
  2:  cbr $ss_c$ 3 1
  3:  read $x_r$ $\gamma_c$
  4:  write $fr_c$ $\top$
```

Figure 1: Implementation of the Handshake Protocol: *send* and *receive*

for SV in Subsection 6.2.1, which in turn allows us to derive a labeled semantics for SV in Subsection 6.2.2.

6.2 Handshake Protocol

In this section, we present a simple handshake protocol to implement abstract synchronous communication with shared variables. To ensure that the CUC programs allow for the implementation with the simple handshake protocol, we consider a subset of CUC by restricting the communication capabilities from multi-way synchronization to unidirectional communication. In general, many protocols realizing synchronous communication can be implemented in SV (e. g., unidirectional communication, bidirectional communication, multi-way synchronization). However, our focus is on the formal implementation relation which relates the abstract communication with its implementation. To investigate how to formally verify such a communication protocol ensuring the preservation of safety and liveness properties, we use a simple handshake protocol to reduce the overhead of the protocol. When defining the handshake refinement in 6.4, we sketch how to apply our approach to other protocols.

First, we introduce the handshake protocol in Subsection 6.2.1. Second, we present how to restrict a CUC program to unidirectional communication with two participants in Subsection 6.2.2.

6.2.1 Description of the Handshake Protocol

The handshake protocol we consider realizes synchronous communication between a *sender* and a *receiver* over a *channel*. The protocol consists of two parts: a protocol *send* for the sender, and a protocol *receive* for the receiver. The channel c is a “namespace” in the shared memory. A channel is formed by the following four shared variables: A mutex variable m_c to lock the channel, two signal variables sr_c (start reading) and fr_c (finished reading) for synchronization, and a shared variable γ_c to store the value. Additionally, two local variables belong to a channel c, which store the results of the `cas` instructions: hl_c (has lock) indicates whether the sender has locked the mutex. ss_c (signal set) indicates whether the signal the sender or the receiver are waiting for has been set to $\top$.

send and *receive* are implemented in SV by the constructs shown in Figure 1. The general idea is that *send* locks the channel c to protect the shared variable, and synchronizes over signals with *receive*. The protocol flow is illustrated in detail in Figure 2 on page 17 when we define the handshake refinement. We explain the details of the implementations of the sender and the receiver line by line; line numbers in parenthesis are followed by a description.

***send*:** (1) The sender checks if the mutex m_c is free, and if it is, writes its *id* to it. The result is stored in hl_c. (2) If it is not free, it checks again (with a busy loop). Otherwise it proceeds to (3) write the data value to be sent from the local register x_s to the shared variable γ_c. Afterwards, it realizes a synchronization with the read process: To this end, 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. It stores the result of the `cas` instruction in line 5 in ss_c.

***receive*:** (1,2) waits with a busy loop for the signal sr_c to be $\top$. If it received the signal, it (3) reads the value from the shared variable and then (4) sets the signal fr_c to $\top$.

Observe that deadlocks in abstract synchronous communication, e. g., in CUC that are due to missing communication partners are implemented as livelocks in SV: *send* cannot exit the busy loop (Lines 5, 6) without a receiver on the same channel, and *receive* cannot exit the loop (Lines 1, 2) without a sender in the channel. As both, deadlocks and livelocks, do not provide communication capabilities, we preserve the offered events, and thereby the liveness properties.

6.2.2 Restriction of CUC

Having introduced the handshake protocol and its implementation in SV, we proceed to discuss the different models of choices of CUC compared to CSP. CUC has non-determinism in the form of the instruction `do`. However, it does not have an internal choice per se. This stems from the fact that internal choice is an abstract modeling construct, and CUC is very close to the implementation level, i. e., we assume that non-determinism was resolved on the CSP level.

CUC has external choice in the form of the abstract communication instruction `comm`. The instruction `comm` can model even so-called *mixed choice*, offering both input and output on different channels (see Example A.3). Synchronous communication where the sender can choose between several channels to output its communication requires *output guards*. Output guards prevent the sender from committing to a channel without a receiver present, which would block the sender, possibly indefinitely. The same is true for the choice of the receiver between multiple, synchronous inputs: *input guards* are needed. The implementation of guards in general requires that components can register and unregister from a channel. Only if enough participants are registered, the communication takes place. Until the communication takes place, all components can unregister from the channel. As mixed choice offers both choices at the same time, it requires both input and output guards to prevent blocking, but it also requires breaking of symmetries to avoid the indefinite search for an available communication partner. The implementation of mixed choice with synchronous communication is considered e. g., by Bougé [Bou88] in form of the leader election problem.

The simple handshake protocol we consider does not support choices, so it neither needs input nor output guards. We point out where guards fit in for future extensions of our formal implementation relation in Subsection 6.4. The protocol supports synchronous, uni-directional communication over a channel with two participants: Sending a value over a channel and synchronizing with any one receiver ready to receive the value. Thus, to use the handshake protocol as implementation of abstract communication, we need to restrict the use of the communication of CUC from synchronous, multi-way communication to synchronous, uni-directional communication over a channel. To this end, we introduce ids for components, define two instantiations of `comm`, namely a sender and a receiver, and we exclude communication with the abstract environment.

We assign each component an identifier id. Let ID be the set of all component ids. As the tree structures for concurrent states and concurrent programs are the same (Assumption A.3), we can define the same function for concurrent states and concurrent programs, which maps the position in the concurrent tree to an id. We write σ_{id} to obtain the id of a local state. We write σ^{id} or cp^{id} to select a specific local state or program with id id from a concurrent state σ or concurrent program cp. Finally, let ids map from a concurrent (sub-) tree to all contained ids.

Definition 6.4: Component Identifier

$$\begin{aligned} ID &\quad \text{(the set of component ids)} \\ \sigma_{id} &: LStates \to ID \\ \sigma^{id} &: States \times ID \to LStates \\ cp^{id} &: CP \times ID \to LP \\ ids(cp) &: CP \to \mathscr{P}(ID) \end{aligned}$$

We define a sender $\texttt{comm}_s$ and a receiver $\texttt{comm}_r$ in CUC as follows.

Definition 6.5: $\texttt{comm}_s$ and $\texttt{comm}_r$

Let c be a channel, x_s and x_r local registers, and id the component id of the current component . The event $c.s.r.v$ is composed of the channel name c, the ids of the sender s and the receiver r, and the transferred data value v of type $\mathbb{T}$. Finally, let $val(c.s.r.v) = v$ extract the data value of an event.

$$\texttt{comm}_s\ c\ x_s \coloneqq \texttt{comm}\ \big(\lambda\sigma.\ \{c.\sigma_{id}.r.\sigma_{ds}(x_s) \mid r \in ID \land r \neq \sigma_{id}\}\big)\big(\lambda ev\ \sigma.\ \sigma\big)$$
$$\texttt{comm}_r\ c\ x_r \coloneqq \texttt{comm}\ \big(\lambda\sigma.\ \{c.s.\sigma_{id}.v \mid s \in ID \land s \neq \sigma_{id} \land v \in \mathbb{T}\}\big)\big(\lambda ev\ \sigma.\ \sigma[x_r \coloneqq val(ev)]\big)$$

$\texttt{comm}_s$ offers events on its channel c, using its own id σ_{id} as sender, and all possible ids but its own 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 σ_{id} as a receiver, all possible ids but its own 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 component id of the sender or the receiver respectively, we are able to 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 single `comm` instruction without a communication partner in CUC should not synchronize with the environment but block. To enforce this in CUC, we only consider concurrent programs with at least two components that are combined with the alphabetized parallel operator. Using the communication interfaces of the alphabetized parallel operator, we ensure that every component may only engage in events that the component's id is part of, expressed by $s \in ids(cp_i) \lor r \in ids(cp_i)$ in the communication interface defined below where s is short for sender and r is short for receiver. Additionally, each component may not communicate with itself, expressed by $s \neq r$. The (maximal) communication interface of each concurrent program cp_i is then given by

$$\alpha_i = \{c.s.r.v \in \Sigma \mid (s \in ids(cp_i) \vee r \in ids(cp_i)) \wedge s \neq r\}.$$

Assumption 6.1 ensures that only $\texttt{comm}_s$ and $\texttt{comm}_r$ are used for communication and that all concurrent components are combined with the aforementioned communication interfaces α_i. As a single component does not require a concurrent composition (and in turn would not be restricted by the communication interface), we require that every program consists of at least two concurrent components. For the rest of this chapter, we assume the following restrictions to hold for CUC programs.

Assumption 6.1: Restrictions to CUC

(I) All instances of `comm` are either $\texttt{comm}_s$ or $\texttt{comm}_r$.

(II) All concurrent CUC programs have at least two components and use communication interfaces that are a subset of the above defined α_i.

With the restrictions of CUC and the component ids defined, we can give an alternative definition of stable states for CUC, which focuses on the instructions instead of the labels. We define the stable states for SV in a similar way. As the only two instructions in CUC that produce the event τ are `do` and `cbr`, we can define stable states alternatively as states pointing to `comm` or outside of the code. The following definition is equivalent to Definition A.14.

Definition 6.6: Stable States in *cuc*

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

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

In this section, we have defined a handshake protocol to implement abstract synchronous communication in our low-level language SV. Furthermore, we have defined restrictions to CUC to ensure that the CUC programs allow for the implementation with the presented handshake protocol. The use of the handshake protocol allows us to talk about the concept of abstract synchronous communication in the context of SV. This enables us to formally relate CUC and SV. In the next section, we define a labeled semantics for SV and related constructs based on the handshake protocol.

6.3 Definitions and SV Semantics with Events

In this section, we lay the foundations to relate CUC and SV programs. Based on the handshake protocol that we defined in the last section, we define several notions to relate different aspects of a CUC program *cuc* and an SV program *sv* where *sv* results from replacing the abstract communication in *cuc* with the handshake protocol. The *program* *label map* (Definition 6.7) relates the syntactic instructions of *cuc* and *sv*. *Similarity* (Definition 6.10) defines how we relate local states of *cuc* and *sv*. Finally, we define a labeled semantics for SV (Definition 6.12), which allows us to define the operational characterization of traces and stable failures semantics for SV (Definitions 6.14 and 6.17). Those semantics allow for comparison of behaviors, especially with respect to safety and liveness properties. All the concepts defined in this section are used in Section 6.4 to define our notion of handshake refinement.

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* in Definition 6.7. Each abstract communication instruction in *cuc* ($\texttt{comm}_s$ or $\texttt{comm}_r$) is related to all the instructions of its protocol implementation.

Definition 6.7: Program Label Map

A **program label map** ψ injectively maps a program label in a CUC program *cuc* to a corresponding program label in an SV program *sv*. The formal requirements, defined below, state that `do` in the component *id* of *cuc* is in a one-to-one correspondence to `do` in the component *id* of *sv*. The same holds for `cbr`. The instruction $\texttt{comm}_s$ is related to *all* instructions of *send*, which implies that the existence of any instruction of *send* implies the existence of the other instructions around it. The same holds true for $\texttt{comm}_r$ and *receive*.

$$
\begin{aligned}
(\ell, \texttt{do}\ f) \in cuc^{id} &\Longleftrightarrow \big(\psi(\ell), \texttt{do}\ f\big) \in sv^{id} \wedge \psi(\ell+1) = \psi(\ell) + 1 \\[1ex]
(\ell, \texttt{cbr}\ b\ m\ n) \in cuc^{id} &\Longleftrightarrow \big(\psi(\ell), \texttt{cbr}\ b\ \psi(m)\ \psi(n)\big) \in sv^{id} \\[1ex]
(\ell, \texttt{comm}_s\ c\ x_s) \in cuc^{id} &\Longleftrightarrow \big(\psi(\ell)+0, \texttt{cas}\ m_c\ \text{FREE}\ id\big) \in sv^{id} \\
'' \qquad\qquad &\Longleftrightarrow \big(\psi(\ell)+1, \texttt{cbr}\ hl_c\ (\psi(\ell)+2)\ \psi(\ell)\big) \in sv^{id} \\
'' \qquad\qquad &\Longleftrightarrow \big(\psi(\ell)+2, \texttt{write}\ \gamma_c\ x_s\big) \in sv^{id} \\
'' \qquad\qquad &\Longleftrightarrow \big(\psi(\ell)+3, \texttt{write}\ sr_c\ \top\big) \\
'' \qquad\qquad &\Longleftrightarrow \big(\psi(\ell)+4, \texttt{cas}\ fr_c\ \top\ \bot\big) \in sv^{id} \\
'' \qquad\qquad &\Longleftrightarrow \big(\psi(\ell)+5, \texttt{cbr}\ ss_c\ (\psi(\ell)+6)\ (\psi(\ell)+4)\big) \in sv^{id} \\
'' \qquad\qquad &\Longleftrightarrow \big(\psi(\ell)+6, \texttt{write}\ m_c\ \text{FREE}\big) \in sv^{id} \\
'' \qquad\qquad &\Longrightarrow\ \psi(\ell+1) = \psi(\ell) + 7 \\[1ex]
(\ell, \texttt{comm}_r\ c\ x_r) \in cuc^{id} &\Longleftrightarrow \big(\psi(\ell)+0, \texttt{cas}\ sr_c\ \top\ \bot\big) \in sv^{id} \\
'' \qquad\qquad &\Longleftrightarrow \big(\psi(\ell)+1, \texttt{cbr}\ ss_c\ \big(\psi(\ell)+2\big)\ \psi(\ell)\big) \in sv^{id} \\
'' \qquad\qquad &\Longleftrightarrow \big(\psi(\ell)+2, \texttt{read}\ x_r\ \gamma_c\big) \in sv^{id} \\
'' \qquad\qquad &\Longleftrightarrow \big(\psi(\ell)+3, \texttt{write}\ fr_c\ \top\big) \in sv^{id} \\
'' \qquad\qquad &\Longrightarrow\ \psi(\ell+1) = \psi(\ell) + 4
\end{aligned}
$$

Using the definition of the program label map, we can define when a CUC program and an SV program fit together.

Definition 6.8: 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.

Given a program label map ψ, it can statically be checked by going through both programs whether two programs cuc and sv are fitting. To relate semantic states of CUC and SV we consider the local (concurrent) states and ignore variables that were added for bookkeeping in the handshake protocol. We define the notion of *channel constituents* to group all variables that belong to a channel.

Definition 6.9: 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, γ_c sr_c, and fr_c.

To exclude other components or instructions from changing the values stored in the channel constituents, we assume in the following that channel constituents are unique for each channel.

Assumption 6.2: Uniqueness of Channel Constituents

All channel constituents from all channels are unique.

It follows from the uniqueness that in a program sv fitting cuc, channel constituents are only changed from within *send* and *receive* of the channel.

Lemma 6.1: Proper Access to Channel Constituents

All channel constituents of a channel c can only be changed by the *send* or *receive* of the channel c.

Proof

Fitting implies a program label map ψ which only allows instructions mapped to $\texttt{comm}_s$ or $\texttt{comm}_r$ to contain channel constituents. ☐

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

Definition 6.10: Similarity with Respect to Channel Constituents

Let $\sigma, \hat{\sigma} \in \mathit{CStates}$ be concurrent local states of a CUC program and an SV program, respectively. Let $\sigma \mathrel{\widehat{=}} \hat{\sigma}$ denote that σ and $\hat{\sigma}$ are equal for all local registers that do not belong to a channel. This equality also does not include the program counters. We say σ is **similar** to $\hat{\sigma}$.

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

Our aim is to show that a program *sv* fitting a program *cuc* preserves the safety and liveness properties of *cuc*. To express safety and liveness properties in SV, we define a semantics with events, stable states, and refusal sets. To this end, we first define an event labeling and an operational semantics for SV with events. Then we define traces and stable failures semantics for SV via an operational characterization. This enables us to show a stable failures refinement between *cuc* and *sv* in the next section. Observe that all definitions regarding the traces and stable failures semantics are very similar to the respective definitions of CUC. This facilitates showing the relation between CUC and SV.

The idea of stable states is that communication is offered in a stable way. This is defined in CSP/CUC as the inability to perform internal steps (τ) as this might disable the communication capabilities. However, CSP/CUC has abstract communication, thus events do not need to be “prepared” to occur. Using the handshake protocol in SV, “administrative” steps happen before and after the visible event occurs. Thus, when labeling the steps of SV, we use a different label for “administrative” steps than for the usual internal steps. We label invisible instructions of the implementation of $\underline{c}$ommunication with τ_c. This allows us to define stable states as the inability to perform internal steps, but allowing the “administrative” steps of the communication to be enabled. 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 the decisions who is the sender and who is the receiver, and the low-level asynchronous semantics where the event happens *after* the sender and the receiver are consecutively decided.

To define a 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. Only `read` is mapped to a visible event. The invisible instructions of the implementation of the communication are mapped to τ_c. All other instructions (`do` and `cbr`) are invisible and mapped to the usual τ.

Definition 6.11: 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.

$$
\begin{array}{rl}
\multicolumn{2}{c}{\mathit{EL}\colon \mathit{GStates} \times \mathit{ID} \times \mathit{Instructions} \to \Sigma \cup \{\tau, \tau_c\}} \\
\mathit{EL}\big((\Gamma, \_), \mathit{id}, \texttt{read}\ \_\ \gamma_c\big) := c.s.r.v & \text{where } s = \Gamma(m_c), r = \mathit{id}, v = \Gamma(\gamma_c) \\
\mathit{EL}(\_, \_, \mathit{ins}) := \tau_c & \text{if } \mathit{ins} \text{ is part of } \mathit{send} \text{ or } \mathit{receive} \text{ (see Fig. 1)} \\
\mathit{EL}(\_, \_, \_) := \tau & \text{otherwise}
\end{array}
$$

Note that the labeling function requires the information about *send* and *receive*, which are directly tied to the abstract communication instructions $\texttt{comm}_s$ and $\texttt{comm}_r$ and the handshake protocol. Using the labeling function *EL*, we can derive an SV semantics with visible events:

Definition 6.12: SV Semantics with Events

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

Here, the active component *id* can be 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.

Assumption 6.3: No Self Loops

$$
\forall\, \mathit{id}\ \ell\ b\ m\ n.\ (\ell, \texttt{cbr}\ b\ m\ n) \in \mathit{cuc}^{\mathit{id}} \vee (\ell, \texttt{cbr}\ b\ m\ n) \in \mathit{sv}^{\mathit{id}} \Longrightarrow \ell \neq m \wedge \ell \neq n
$$

Having labeled single steps, we can now define execution semantics labeled with the visible trace.

Definition 6.13: Operational Traces Semantics of SV

$$
\textsc{exec-0}\quad \frac{}{(\Gamma, \sigma) \xRightarrow{\langle\rangle}_{cp} (\Gamma, \sigma)}
$$

$$
\textsc{exec-ev}\quad \frac{(\Gamma, \sigma) \xRightarrow{tr'}_{cp} (\Gamma'', \sigma'') \qquad (\Gamma'', \sigma'') \xrightarrow{ev}_{cp} (\Gamma', \sigma') \qquad tr' \frown \langle ev \rangle = tr \qquad ev \notin \{\tau, \tau_c\}}{(\Gamma, \sigma) \xRightarrow{tr}_{cp} (\Gamma', \sigma')}
$$

$$
\textsc{exec-}\tau\quad \frac{(\Gamma, \sigma) \xRightarrow{tr}_{cp} (\Gamma'', \sigma'') \qquad (\Gamma'', \sigma'') \xrightarrow{ev}_{cp} (\Gamma', \sigma') \qquad ev \in \{\tau, \tau_c\}}{(\Gamma, \sigma) \xRightarrow{tr}_{cp} (\Gamma', \sigma')}
$$

The visible traces neither contain τ nor τ_c. Visible events are appended at the end of traces. We proceed and define the traces semantics $\mathcal{T}_{sv}$ for SV via an operational characterization. It captures all traces that are possible, starting in σ.

Definition 6.14: Traces Semantics for SV

$$tr \in \mathcal{T}_{sv}(\Gamma, \sigma) \coloneqq \exists\, \Gamma'\ \sigma'.\ (\Gamma, \sigma) \overset{tr}{\Longrightarrow}_{sv} (\Gamma', \sigma')$$

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 6.15: Stable States in *sv*

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

$$\begin{aligned}(\Gamma, \sigma){\downarrow_{sv}} \coloneqq \forall\, id.\ \big(&\nexists\, ins.\ (\sigma^{id}_{pc}, ins) \in sv^{id}\big)\\ &\vee \big(\exists\, c.\ (\sigma^{id}_{pc}, \texttt{cas}\ m_c\ \textsc{free}\ id) \in sv^{id}\\ &\qquad \vee (\sigma^{id}_{pc}, \texttt{cas}\ sr_c\ \top\ \bot) \in sv^{id}\big)\end{aligned}$$

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 (labeling `read`) occurs only in the middle of the execution of the handshake protocol, a finite number of τ_c-steps are allowed before the visible event in order 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. In the following definition of refusal sets let $\xrightarrow{\tau_c}{}^{*}_{sv}$ denote zero or more τ_c steps.

Definition 6.16: 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 $X \subseteq \Sigma$.

$$(\Gamma, \sigma)\ \mathrm{ref}_{sv}\ X \coloneqq \forall\, a \in X.\ \nexists\, \Gamma'\ \sigma'.\ (\Gamma, \sigma) \xrightarrow{\tau_c}{}^{*}_{sv} \xrightarrow{a}_{sv} (\Gamma', \sigma')$$

Having defined stable states and refusal sets for SV, we can finally define stable failures for SV.

Definition 6.17: 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 (Γ', σ') which can be reached from the initial state σ via the trace *tr* and refuses X.

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

This concludes the definition of the SV semantics with events. In this section, we have defined which CUC and SV programs to relate to each other (*fitting*), how states will be compared (*similar*), and a stable failures semantics for SV. In the next section, we define our notion of handshake refinement to formally relate CUC and SV programs. We use the stable failures semantics to show that the handshake refinement ensures that safety and liveness properties are preserved.

6.4 Handshake Refinement

In this section, we define our notion of handshake refinement to relate abstract communication and its low-level implementation with a handshake protocol. 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 channel-state $\mathcal{X}$) to track the progress of the protocol executions for each channel. This enables us to relate SV states at different stages of the protocol execution to the same CUC state. During the execution of each individual protocol, as first the sender and then the receiver are determined, the possible events offered by the SV state may be fewer than those offered by the related CUC state, where neither the sender nor the receiver are yet determined. The channel-state enables different treatment in the relation of the same CUC state at different stages of the protocol execution. We use the channel-state to indicate which possible events of the CUC state need to be answered by the SV state. The channel-state $\mathcal{X}$ is a function from channel names to the states of the channels. If the channel c is clear from the context, we only talk about "the channel-state" and omit "of channel c". Let $\uplus$ denote a disjoint set union.

$$\mathcal{X}\colon \mathit{Channels} \to \{\textsc{free}\} \uplus \mathit{ID}_{in} \uplus (\mathit{ID} \times \mathit{ID})_{in} \uplus (\mathit{ID} \times \mathit{ID})_{un} \uplus \mathit{ID}_{un}$$

Each channel can be in one of five states: It can be FREE, a sender or both a sender and a receiver are $\underline{\text{in}}$ the channel, and after the communication happened, the channel will be eventually $\underline{\text{un}}$locked, first with both a sender and a receiver still in the channel, then only a sender. The states of the channel-state $\mathcal{X}(c)$ for the considered channel c within the protocol flow are illustrated in Figure 2 in the rectangular boxes in the middle column. Figure 2 illustrates the protocol flow for a sender and receiver on a single channel. 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. In the upper right corner, also those of `do` (D) and `cbr` (C) are depicted, as well as those pointing outside the code (O). N (for $\underline{\text{n}}$on-protocol state) is a placeholder for O, D, C, S, or R, thus, all states which do not occur within[1] the execution of the handshake protocol. Dotted lines indicate the boundaries between channel-states. 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 to talk about ids that do not appear in the channel-state at all and completely free

---

[1] We do not treat S and R as states that occur *within* the execution of the protocol. The idea is that leaving the state S or R starts the execution of the protocol.

channel-states.

Definition 6.18: *id* not in the Channel-State

$$id \notin \mathcal{X} \coloneqq \forall\, c\; id'.\, \mathcal{X}(c) \notin \left\{id_{in}, (id, id')_{in}, (id', id)_{in}, (id, id')_{un}, (id', id)_{un}, id_{un}\right\}$$

We call a channel-state *empty*, it if is FREE for all channels:

$$\mathcal{X} = \emptyset \coloneqq \forall\, c.\; \mathcal{X}(c) = \text{FREE}$$

Having introduced the channel-state $\mathcal{X}$, we define the handshake refinement in Definition 6.19. It is a relation parametrized over two programs *cuc* and *sv* fitting with ψ. The elements are triplets consisting of a concurrent CUC state σ, a channel-state $\mathcal{X}$, and pair of global state Γ and concurrent 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* (as defined in Definition 6.10). Furthermore, they fulfill the protocol constraints $\mathcal{P}_{cuc,sv,\psi}$, which constrain the possible SV states and their relation to CUC states. The protocol constraints $\mathcal{P}_{cuc,sv,\psi}$ are defined separately in Definition 6.20 and explained below. The possible transitions within the handshake refinement 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 that were possible in *sv* right before this choice of a particular sender are considered by the down-simulation. Note that we allow any number of "administrative" events τ_c even when answering a τ step, although one could think that the internal τ steps do not require the consideration of the communication protocol. This is necessary, as the τ steps do not have an associated channel and, thus, the corresponding channel state cannot be checked if it is FREE. Therefore, if the event *before* the τ step was a visible step, it is possible that the communication protocol for that event is not yet finished, however the related CUC state is already "after communication". Finishing the communication protocol results in τ_c steps that must occur before the considered τ step can happen. The **up-simulation** relates transitions in *sv* to transitions in *cuc*. The "administrative" event τ_c is related to zero transitions in *cuc*, all other events to one. Finally, the **unlocking-simulation** ensures (assuming 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 Definition 6.20 we define the protocol constraints $\mathcal{P}_{cuc,sv,\psi}$, which are specific to the handshake protocol at hand. The protocol constraints ensure a) that only SV states reachable by the execution of the handshake protocol execution are included, and b) that the channel-state reflects the current progress of the protocol execution. The overall definition is that for every channel, if the channel-state is FREE, the belonging signals must be $\perp$, and for each component with id *id* the disjunction $\mathcal{P}^{id}_{cuc,sv,\psi}$, which is also defined in Definition 6.20, must hold. The disjuncts of $\mathcal{P}^{id}_{cuc,sv,\psi}$ (O, D, …, R3)

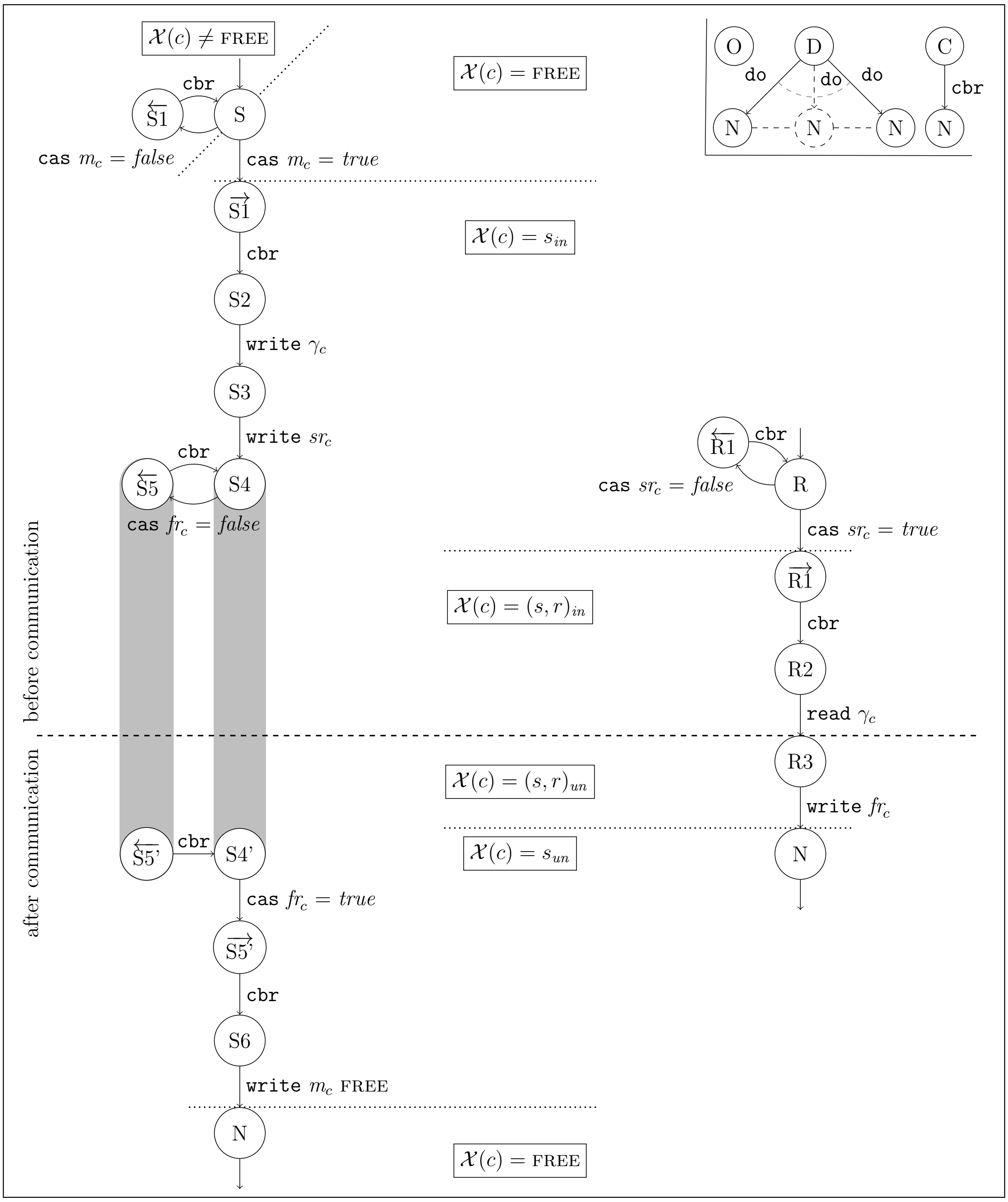

Figure 2: The Flow of the Handshake Protocol

Definition 6.19: Handshake Refinement $\mathcal{B}_{cuc,sv,\psi}$

Let a CUC program *cuc* and an SV program *sv* be fitting with a program label map ψ. A **handshake refinement** is a ternary relation $\mathcal{B}_{cuc,sv,\psi}$ over CUC states (cuc), channel-states ($\mathcal{X}$), and SV states $\big((\Gamma, \hat{\sigma})\big)$, which fulfills the following properties.

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

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

Protocol constraints: $\mathcal{P}_{cuc,sv,\psi}\big(\sigma, \mathcal{X}, (\Gamma, \hat{\sigma})\big)$ (see Definition 6.20)

Down-simulation:

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

Up-simulation:

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

Unlocking-simulation:

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

correspond to the states with the same names in the protocol flow in Figure 2. The disjuncts describe triplets $(cuc, \mathcal{X}, sv)$, consisting of a CUC state *cuc*, a channel-state $\mathcal{X}$, and an SV state *sv*. They provide sufficient conditions to the SV state to be reachable by the execution of the protocol. They constrain the program counters and channel related variables, and thereby relate the SV state via the program label map ψ with the CUC state and the appropriate channel-state. In $\mathcal{P}^{id}_{cuc,sv,\psi}$, the channel-state also "synchronizes" the different components, i. e., excludes illegal state combinations of different components, e. g., two components having a lock on the same channel. It follows a description of the disjuncts, from which we provide two formally. A complete formal definition of the protocol constraints can be found in the Appendix A.2 in Definition A.15.

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

Definition 6.20: Protocol Restrictions

$$\begin{aligned}\mathcal{P}_{cuc,sv,\psi}\big(\sigma,\mathcal{X},(\Gamma,\hat{\sigma})\big) \coloneqq &\big(\forall\, c.\, \mathcal{X}(c) = \textsc{free} \implies \neg\Gamma(sr_c) \wedge \neg\Gamma(fr_c)\big)\\ &\wedge\ \forall\, id.\mathcal{P}^{id}_{cuc,sv,\psi}\big(\sigma,\mathcal{X},(\Gamma,\hat{\sigma})\big)\end{aligned}$$

$$\mathcal{P}^{id}_{cuc,sv,\psi}\big(\sigma,\mathcal{X},(\Gamma,\hat{\sigma})\big) \coloneqq O\vee D\vee C\vee S\vee S1\vee S2\vee S3\vee S4\vee S5\vee S4'\vee S5'\vee S6\vee R\vee R1\vee R2\vee R3$$

O, D, C Have a direct counterpart in CUC, channel variables are not a concern, $id \notin \mathcal{X}$

D do f instruction

C cbr

S At the beginning of *send*, $id \notin \mathcal{X}$

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 stored in the mutex of the channel and in the channel-state.

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

S4 The first row of the following formula ensures that the SV state is mapped to a CUC state where the *pc* points to the appropriate `comm`. The second row ensures that mutex is locked by the considered component, the value of the shared variable is the value to be sent, and the signal indicating that reading is finished (fr_c) is not set. The third row describes the signal sr_c and the channel-state. Start reading was set to $\top$ from S3 to S4. If the receiver did start reading, then start reading will remain $\bot$ from now on. In the first case the channel-state only contains the sender, in the second also the receiver.

$$\begin{aligned}&(\sigma^{id}_{pc}, \texttt{comm}_s\, id\, c\, x_s) \in cuc^{id} \wedge (\hat{\sigma}^{id}_{pc}, \texttt{cas}\; ss_c\, fr_c\, \top\, \bot) \in sv^{id} \wedge \psi(\sigma^{id}_{pc}) + 4 = \hat{\sigma}^{id}_{pc}\\ &\wedge \Gamma(m_c) = id \wedge \Gamma(\gamma_c) = \hat{\sigma}^{id}_{ds}(x_s) \wedge \neg\Gamma(fr_c)\\ &\wedge\ \big(\Gamma(sr_c) \wedge \mathcal{X}(c) = id_{in} \vee \neg\Gamma(sr_c) \wedge (\exists\, id_r.\, \mathcal{X}(c) = (id, id_r)_{in})\big)\end{aligned}$$

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

S4' From now on, the communication already has happened. The channel-state 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 subtract 1 from the program counter of the SV state, to map with ψ to `comm`.

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

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

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

R At the beginning of *receive*, $id \notin \mathcal{X}$

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

R2 The channel-state contains the sender and the receiver about to communicate.

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

(which match with the four non-FREE channel-states): *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, input and output guards could be supported.

In this section, we have presented our notion of handshake refinement. It is an asymmetric implementation relation. The focus of our handshake refinement is on the implementation of abstract communication. Outside of the implementation of abstract communication, it is defined like a strong bisimulation. In the next section, we show that our notion of handshake refinement implies a stable failures refinement. Thus, the handshake refinement preserves safety and liveness properties.

6.5 Preservation of Safety and Liveness Properties

In this section, we prove that every SV program *sv* fitting a CUC program *cuc* preserves all safety and liveness properties of *cuc*. To this end, we first show that the handshake refinement relation preserves safety and liveness properties. Second, we show that all pairs of fitting CUC and SV programs are in a handshake refinement relation.

6.5.1 Handshake Refinement preserves Safety and Liveness Properties

In this subsection, we first show the preservation of safety properties, and then the preservation of liveness properties.

We capture safety properties using the traces semantics. To show the preservation of safety properties, we show that every trace of *sv* is also a trace of *cuc*. To this end, we show that starting with a triplet $\big(\sigma_0, \emptyset, (\Gamma_0, \hat{\sigma}_0)\big) \in \mathcal{B}_{cuc,sv,\psi}$, 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 6.2: All *sv* Traces and Their *cuc* Counterparts are in $\mathcal{B}_{cuc,sv,\psi}$

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

Proof

Using induction of the up-simulation. □

We can directly conclude the preservation of safety properties: All traces of *sv* are also traces of *cuc*.

Theorem 6.1: Preservation of Safety Properties

$$\big(\sigma, \emptyset, (\Gamma, \hat{\sigma})\big) \in \mathcal{B}_{cuc,sv,\psi} \implies \mathcal{T}(\Gamma, \hat{\sigma})_{sv} \subseteq \mathcal{T}(\sigma)_{cuc}$$

Proof

Using the Definitions A.13 and 6.14 of the operational characterizations of the traces semantics of CUC and SV, respectively, and Lemma 6.2. ☐

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. To this end, we 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*. To show the preservation of liveness properties, we first show two lemmas: Lemma 6.3 shows that stable states in *sv* imply stable states in *cuc*. Lemma 6.4 shows that refusals of *sv* imply refusals of *cuc*.

Lemma 6.3: Stable States in *sv* Imply Stable States in *cuc* and $\mathcal{X} = \emptyset$

$$\big(\sigma, \mathcal{X}, (\Gamma, \hat{\sigma})\big) \in \mathcal{B}_{cuc,sv,\psi} \land (\Gamma, \hat{\sigma}){\downarrow_{sv}} \implies \sigma{\downarrow_{cuc}} \land \mathcal{X} = \emptyset$$

Proof

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

The key lemma to prove the theorem of preservation of liveness states that in a triplet in a handshake refinement, if the *sv* state is stable, then any events the *sv* state can refuse can also be refused by the *cuc* state.

Lemma 6.4: Refusals in *sv* Imply Refusals in *cuc*

$$\big(\sigma, \mathcal{X}, (\Gamma, \hat{\sigma})\big) \in \mathcal{B}_{cuc,sv,\psi} \land (\Gamma, \hat{\sigma}){\downarrow_{sv}} \implies (\Gamma, \hat{\sigma}) \text{ ref}_{sv}\ X \implies \sigma \text{ ref}_{cuc}\ X$$

Proof

Using Lemma 6.3, we have $\mathcal{X} = \emptyset$ and can apply the down-simulation. The down-simulation ensures that the SV program *sv* has at least the communication capabilities of the CUC program *cuc*. It follows that the refusals of *sv* are included in the refusals of *cuc*. A more technical proof is in the Appendix A.4. ☐

Now, we can show the preservation of liveness properties, i. e., the inclusion of stable failures.

```
send:
    1: cas $hl_c$ $m_c$ FREE TAKEN
    2: cbr $hl_c$ 3 1
    3: write $\gamma_c$ $x_s$
    4: write $sr_c$ $\top$
    5: cas $ss_c$ $fr_c$ $\top$ $\bot$
    6: cbr $ss_c$ 7 5
    7: write $m_c$ FREE

receive:
    1: cas $ss_c$ $sr_c$ $\top$ $\bot$
    2: cbr $ss_c$ 3 1
    3: read $x_r$ $\gamma_c$
    4: write $fr_c$ $\top$
```

Figure 3: Alternative Implementation of the Handshake Protocol Without Sender Identifier in the Mutex

Theorem 6.2: Preservation of Liveness Properties

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

Proof

To show $\mathcal{SF}_{sv}(\Gamma, \hat{\sigma}) \subseteq \mathcal{SF}_{cuc}(\sigma)$, fix a stable failure in *sv* and find it in *cuc*, i. e., find the same pair of trace *tr* and refusal set X. We show $(tr, X) \in \mathcal{SF}_{sv}(\Gamma, \hat{\sigma})$ is also a stable failure of *cuc*, i. e., $(tr, X) \in \mathcal{SF}_{cuc}(\sigma)$, with the previous lemmas: A trace of *sv* implies a trace of *cuc* (Lemma 6.2), the stable states in *sv* imply stable states in *cuc* (Lemma 6.3), and the refusal sets of *sv* imply refusal sets of *cuc* (Lemma 6.4). ☐

Having shown that the handshake refinement preserves safety and liveness properties, we show that we need the information about the sender, which is stored in the mutex, only for the proofs. It does not affect the semantics of the programs. To demonstrate this, we consider a slightly different program sv', and show that it has the same properties. The program sv' differs from *sv* in that it does not store the id of the component which has the lock in the mutex, but only that the lock is TAKEN. Figure 3 shows the program sv'. In *sv*, we store the information about the sender in the mutex to reconstruct the sender at the time of reading the shared variable. This information is only needed for the labeling and the proof. However, the execution of the (concurrent) program *sv* only depends on the information whether the mutex was taken, not by whom. Thus, sv' has exactly the same executions as *sv* and the following corollary holds.

Corollary 6.1: Liveness Properties Without Sender Identifier

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.

In this subsection, we have shown that the handshake refinement implies a stable failures refinement, and as such, preserves safety and liveness properties. In the next subsection, we show that when replacing all instances of $\texttt{comm}_s$ and $\texttt{comm}_r$ in a CUC program *cuc* with *send* and *receive* according to the handshake protocol, the resulting SV program *sv* is in a handshake refinement relation with *cuc*, and, thus, has the same safety and liveness properties.

6.5.2 Fitting Programs preserve Safety and Liveness Properties

In this subsection, 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 (as defined in Theorem 6.3) are in a handshake refinement relation. The resulting theorem allows for a scalable approach to the verification of shared variable communication, as we show it once for all fitting programs.

Theorem 6.3: Fitting Implies Handshake Refinement

Let *sv* be a program fitting *cuc* with the program label map 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, and all signals are inactive.

$$\begin{gathered}\sigma \mathrel{\hat{=}} \hat{\sigma} \wedge \big(\forall\, id.\ \hat{\sigma}^{id}_{pc} = \psi(\sigma^{id}_{pc})\big) \wedge \big(\forall\, c.\ \Gamma(m_c) = \textsc{free} \wedge \neg\Gamma(sr_c) \wedge \neg\Gamma(fr_c)\big) \\ \implies \big(\sigma, \emptyset, (\Gamma, \hat{\sigma})\big) \in \mathcal{B}_{cuc,sv,\psi}\end{gathered}$$

Proof: Idea

The proof can be found in Appendix A.3 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 (answer visible events) *only* in stable states.

As the handshake refinement implies preservation of safety (Theorem 6.1) and liveness properties (Theorem 6.2), we can now conclude with Theorem 6.3 that all fitting programs share the same safety and liveness properties.

Theorem 6.4: Fitting Implies Preservation

Let *sv* be a program fitting *cuc* with ψ. Then all safety and liveness properties from *cuc* are preserved to *sv*.

Proof

Follows from Theorem 6.3 and Theorems 6.1 and 6.2. □

In this section, we have shown that every pair of CUC and SV programs *cuc* and *sv*, where *sv* can be obtained by replacing the abstract communication in *cuc* with the handshake protocol, has the same safety and liveness properties. The generality of Theorem 6.4 allows for scalability of showing the preservation of safety and liveness properties. The next section concludes this chapter.

6.6 Summary

In this chapter, we have presented a method to relate abstract synchronous communication with an asynchronous handshake implementation using shared variable communication and have proven that this method preserves safety and liveness properties. To this end, we have defined our generic low-level language SV that allows for the implementation of communication protocols using shared variables. The language SV can be instantiated to current instruction set architectures. We have defined traces and stables failures semantics for SV to formalize the preservation of 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 proven in the general Theorem 6.4 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. The generality of the theorem makes it independent of the number of components. Together with our compositional method to show the preservation of safety and liveness properties from CSP to CUC in the previous chapter, we have a *compositional* framework to prove the preservation of safety and liveness properties from abstract specifications in CSP down to low-level code, including asynchronous communication mechanisms. While the handshake refinement, and especially the protocol constraints ($\mathcal{P}_{cuc,sv,\psi}$), depends on the protocol used for the implementation, it is easy to integrate other protocols. We have given pointers how to adapt the definition for use with other protocols in Section 6.4. In the next chapter, we demonstrate the application of our framework using an example with n clients and an arbitrary but fixed number of servers.

A Appendix

A.1 Definitions from Previous Chapters

Definition A.3: Local Program lp

$$LP := \mathscr{P}(\mathit{Labels} \times \mathit{Instructions})$$

Definition A.5: Concurrent Program cp

$$CP := LP \mid CP\ {}_{\mathscr{P}(\Sigma)}\|_{\mathscr{P}(\Sigma)}\ CP$$

Definition A.13: Operational Characterization of the Traces of CUC

The traces semantics of CUC captures all traces tr that can be observed when running the program cuc starting in the state σ.

$$tr \in \mathcal{T}_{cuc}(\sigma) := \exists\, \sigma'.\ \sigma \overset{tr}{\Longrightarrow}_{cuc} \sigma'$$

Definition A.14: CSP-like Stable States of CUC

$$\sigma{\downarrow_{cuc}} := \nexists\, \sigma'.\ \sigma \xrightarrow{\tau}_{cuc} \sigma'$$

Assumption A.2: Uniqueness of Labels

$$(\ell, ins_1) \in lp \wedge (\ell, ins_2) \in lp \Longrightarrow ins_1 = ins_2$$

Assumption A.3: Same Tree Structure

For a given concurrent state and its associated concurrent program, we always assume that they have the same tree structure, i. e., they are isomorphic.

Example A.3: Instantiations of `comm` f_{ev} f_{ds}

The instruction `comm` can be instantiated, e. g., to send a value stored in a variable over channel *out*, to receive a value of type $\mathbb{T}$ over channel *in* and store it in a register, or to select between sending a value on one channel or receiving a value on another channel. Note that, as we use CSP communication, the only difference between “sending” and “receiving” a value is in the number of offered events. In Section 6.2, we introduce restrictions to obtain “true send/receive semantics”. As we use the communication mechanism of CSP, we use the *val* function, as already

Example A.3: Instantiations of `comm` f_{ev} f_{ds}

defined in Section 2.3, to extract the value of an event.

$$\begin{aligned}
\texttt{send}\ x &:= \texttt{comm}\ (\lambda ds.\ \{out.v \mid v = ds(x)\})\ (\lambda ds\ ev.\ ds)\\
\texttt{receive}\ x &:= \texttt{comm}\ (\lambda ds.\ \{in.v \mid v \in \mathbb{T}\})\ (\lambda ds\ ev.\ ds[x := val(ev)])\\
\texttt{select}\ x &:= \texttt{comm}\ (\lambda ds.\ \{in.v \mid v \in \mathbb{T}\} \cup \{out.v \mid v = ds(x)\})\\
&\qquad (\lambda ds\ ev.\ \texttt{if}\ ev = in.v\ \texttt{then}\ ds[x := v]\ \texttt{else}\ ds)
\end{aligned}$$

A.2 Protocol Constraints

Definition A.15 gives the complete formal definition of the protocol constraints $\mathcal{P}_{cuc,sv,\psi}$. We describe here the differences to Definition 6.20, where we have used mostly natural language for the definition.

In each disjunct, the program counters, the instructions they point to, and their relation via ψ are described, e. g., in (D):

$$(\sigma_{pc}^{id}, \texttt{do } f) \in cuc^{id} \wedge (\hat{\sigma}_{pc}^{id}, \texttt{do } f) \in sv^{id} \wedge \psi(\sigma_{pc}^{id}) = \hat{\sigma}_{pc}^{id}$$

This information corresponds to the information from Definition 6.8 of a fitting program label map, and is written in gray in Definition A.15. Observe that for disjuncts, where the communication has already happened (S4', S5', S6, R3), we need to consider the instruction of the previous CUC state ($\sigma_{pc}^{id} - 1$), as ψ always maps $\texttt{comm}_s$ and $\texttt{comm}_r$ to their entire implementations *send* and *receive*, respectively, regardless whether the communication has already happened. As we only consider $\sigma_{pc}^{id} - 1$ in parts of the implementation of *send* and *receive* after the communication has already happened and $\texttt{comm}_s$ and $\texttt{comm}_r$ increase the program counter by one, we know that the previous instruction indeed was a $\texttt{comm}_s$ or $\texttt{comm}_r$ by the definition of the program label map ψ.

The conditions in black cover the channel-state and the global state, e. g., in (S2)

$$\mathcal{X}(c) = id_{in} \wedge \Gamma(m_c) = id \wedge \neg\Gamma(sr_c) \wedge \neg\Gamma(fr_c)$$

The channel-state $\mathcal{X}$ appears in each disjunct. It also “synchronizes’ the different components, i. e., for each component we only need to describe local information and the channel the component is currently using. As the conditions for every component are only concerned with whether the component itself occurs in the channel-state (and where applicable also a communication partner), the condition for free channels (that both signals need to be $\perp$) needs to occur at the top level (see the first line of the figure).

The states of the mutex (m_c), the signals (sr_c and fr_c), the return registers of the `cas` instructions (has_lock hl_c and signal_set ss_c), as well as the data value of the shared variable γ_c are also described where necessary. Due to the “synchronization” of the components via the channel-state $\mathcal{X}$, most conditions only need to be specified in one place, either the sender or the receiver – we chose the sender, as it comes first.

Finally, the symbol $\veebar$ denotes an exclusive or. $a \veebar b \coloneqq a \vee b \wedge \neg(a \wedge b)$

Definition A.15: Protocol Constraints (Full)

$$\begin{aligned}\mathcal{P}_{cuc,sv,\psi}\big(\sigma,\mathcal{X},(\Gamma,\hat{\sigma})\big) \coloneqq &\big(\forall\, c.\, \mathcal{X}(c) = \textsc{free} \Longrightarrow \neg\Gamma(\mathit{sr}_c) \land \neg\Gamma(\mathit{fr}_c)\big)\\ &\land\ \forall\, \mathit{id}. \mathcal{P}^{id}_{cuc,sv,\psi}\big(\sigma,\mathcal{X},(\Gamma,\hat{\sigma})\big)\end{aligned}$$

$$\mathcal{P}^{id}_{cuc,sv,\psi}\big(\sigma,\mathcal{X},(\Gamma,\hat{\sigma})\big) \coloneqq$$

Out of code:

$$\big(\,\nexists\, ins.\ (\sigma^{id}_{pc}, ins) \in cuc^{id}\big) \land \big(\,\nexists\, ins.\ (\hat{\sigma}^{id}_{pc}, ins) \in sv^{id}\big) \land \psi(\sigma^{id}_{pc}) = \hat{\sigma}^{id}_{pc} \land \mathit{id} \notin \mathcal{X} \tag{O}$$

`do` f:

$$\lor\, (\sigma^{id}_{pc}, \texttt{do}\ f) \in cuc^{id} \land (\hat{\sigma}^{id}_{pc}, \texttt{do}\ f) \in sv^{id} \land \psi(\sigma^{id}_{pc}) = \hat{\sigma}^{id}_{pc} \land \mathit{id} \notin \mathcal{X} \tag{D}$$

`cbr`:

$$\lor\, \big((\sigma^{id}_{pc}, \texttt{cbr}\, b\, m\, n) \in cuc^{id} \land (\hat{\sigma}^{id}_{pc}, \texttt{cbr}\, b\, \psi(m)\, \psi(n)) \in sv^{id} \land \psi(\sigma^{id}_{pc}) = \hat{\sigma}^{id}_{pc} \land \mathit{id} \notin \mathcal{X} \tag{C}$$

send:

$$\lor\, (\sigma^{id}_{pc}, \texttt{comm}_s\, \mathit{id}\, c\, x_s) \in cuc^{id} \land (\hat{\sigma}^{id}_{pc}, \texttt{cas}\, hl_c\, m_c\, \textsc{free}\, \mathit{id}) \in sv^{id} \land \psi(\sigma^{id}_{pc}) = \hat{\sigma}^{id}_{pc} \land \mathit{id} \notin \mathcal{X} \tag{S}$$

$$\begin{aligned}&\lor\, (\sigma^{id}_{pc}, \texttt{comm}_s\ \mathit{id}\ c\ x_s) \in cuc^{id} \land \psi(\sigma^{id}_{pc}) + 1 = \hat{\sigma}^{id}_{pc} \land\\ &\quad \big(\hat{\sigma}^{id}_{pc}, \texttt{cbr}\ hl_c\ \big(\psi(\sigma^{id}_{pc}) + 2\big)\ \psi(\sigma^{id}_{pc})\big) \in sv^{id} \land \big(\neg\hat{\sigma}^{id}_{ds}(hl_c) \land \mathit{id} \notin \mathcal{X} \lor\\ &\quad \hat{\sigma}^{id}_{ds}(hl_c) \land \mathcal{X}(c) = \mathit{id}_{in} \land \Gamma(m_c) = \mathit{id} \land \neg\Gamma(\mathit{sr}_c) \land \neg\Gamma(\mathit{fr}_c)\big)\end{aligned} \tag{S1}$$

$$\begin{aligned}&\lor\, (\sigma^{id}_{pc}, \texttt{comm}_s\ \mathit{id}\ c\ x_s) \in cuc^{id} \land (\hat{\sigma}^{id}_{pc}, \texttt{write}\ \gamma_c\ x_s) \in sv^{id} \land \psi(\sigma^{id}_{pc}) + 2 = \hat{\sigma}^{id}_{pc}\\ &\quad \land \mathcal{X}(c) = \mathit{id}_{in} \land \Gamma(m_c) = \mathit{id} \land \neg\Gamma(\mathit{sr}_c) \land \neg\Gamma(\mathit{fr}_c)\end{aligned} \tag{S2}$$

$$\begin{aligned}&\lor\, (\sigma^{id}_{pc}, \texttt{comm}_s\ \mathit{id}\ c\ x_s) \in cuc^{id} \land (\hat{\sigma}^{id}_{pc}, \texttt{write}\ \mathit{sr}_c\ \top) \in sv^{id} \land \psi(\sigma^{id}_{pc}) + 3 = \hat{\sigma}^{id}_{pc}\\ &\quad \land \mathcal{X}(c) = \mathit{id}_{in} \land \Gamma(m_c) = \mathit{id} \land \Gamma(\gamma_c) = \hat{\sigma}^{id}_{ds}(x_s)\ \land \neg\Gamma(\mathit{sr}_c) \land \neg\Gamma(\mathit{fr}_c)\end{aligned} \tag{S3}$$

$$\begin{aligned}&\lor\, (\sigma^{id}_{pc}, \texttt{comm}_s\, \mathit{id}\, c\, x_s) \in cuc^{id} \land (\hat{\sigma}^{id}_{pc}, \texttt{cas}\, ss_c\, \mathit{fr}_c\, \top\, \bot) \in sv^{id} \land \psi(\sigma^{id}_{pc}) + 4 = \hat{\sigma}^{id}_{pc}\\ &\quad \land \Gamma(m_c) = \mathit{id} \land \Gamma(\gamma_c) = \hat{\sigma}^{id}_{ds}(x_s) \land \neg\Gamma(\mathit{fr}_c) \land\\ &\quad \big(\Gamma(\mathit{sr}_c) \land \mathcal{X}(c) = \mathit{id}_{in} \lor \neg\Gamma(\mathit{sr}_c) \land (\exists\, \mathit{id}_r.\, \mathcal{X}(c) = (\mathit{id}, \mathit{id}_r)_{in})\big)\end{aligned} \tag{S4}$$

$$\begin{aligned}&\lor\, (\sigma^{id}_{pc}, \texttt{comm}_s\, \mathit{id}\, c\, x_s) \in cuc^{id} \land \Big(\hat{\sigma}^{id}_{pc}, \texttt{cbr}\, ss_c\ \big(\psi(\sigma^{id}_{pc}) + 6\big)\ \big(\psi(\sigma^{id}_{pc}) + 4\big)\Big) \in sv^{id} \land\\ &\quad \psi(\sigma^{id}_{pc}) + 5 = \hat{\sigma}^{id}_{pc} \land \Gamma(m_c) = \mathit{id} \land \neg\Gamma(\mathit{fr}_c) \land \Gamma(\gamma_c) = \hat{\sigma}^{id}_{ds}(x_s) \land \neg\hat{\sigma}^{id}_{ds}(ss_c) \land\\ &\quad \big(\Gamma(\mathit{sr}_c) \land \mathcal{X}(c) = \mathit{id}_{in} \lor \neg\Gamma(\mathit{sr}_c) \land (\exists\, \mathit{id}_r.\, \mathcal{X}(c) = (\mathit{id}, \mathit{id}_r)_{in})\big)\end{aligned} \tag{S5}$$

$$\begin{aligned}&\lor\, (\sigma^{id}_{pc} - 1, \texttt{comm}_s\, \mathit{id}\, c\, x_s) \in cuc^{id} \land (\hat{\sigma}^{id}_{pc}, \texttt{cas}\, ss_c\, \mathit{fr}_c\, \top\, \bot) \in sv^{id} \land\\ &\quad \psi(\sigma^{id}_{pc} - 1) + 4 = \hat{\sigma}^{id}_{pc} \land \Gamma(m_c) = \mathit{id} \land \neg\Gamma(\mathit{sr}_c) \land\\ &\quad \big(\Gamma(\mathit{fr}_c) \land \mathcal{X}(c) = \mathit{id}_{un} \lor \neg\Gamma(\mathit{fr}_c) \land (\exists\, \mathit{id}_r. \mathcal{X}(c) = (\mathit{id}, \mathit{id}_r)_{un})\big)\end{aligned} \tag{S4'}$$

$$\begin{aligned}&\lor\, (\sigma^{id}_{pc} - 1, \texttt{comm}_s\, \mathit{id}\, c\, x_s) \in cuc^{id} \land \psi(\sigma^{id}_{pc} - 1) + 5 = \hat{\sigma}^{id}_{pc} \land \Gamma(m_c) = \mathit{id} \land \neg\Gamma(\mathit{sr}_c)\\ &\quad \Big(\hat{\sigma}^{id}_{pc}, \texttt{cbr}\, ss_c \big(\psi(\sigma^{id}_{pc} - 1) + 6\big) \big(\psi(\sigma^{id}_{pc} - 1) + 4\big)\Big) \in sv^{id} \land \big((\Gamma(\mathit{fr}_c) \veebar \hat{\sigma}^{id}_{ds}(ss_c))\end{aligned}$$

Definition A.15: Protocol Constraints (Full)

$$\wedge\, \mathcal{X}(c) = id_{un} \vee \neg\Gamma(fr_c) \wedge \neg\hat{\sigma}^{id}_{ds}(ss_c) \wedge \big(\exists\, id_r.\mathcal{X}(c) = (id, id_r)_{un})\big) \tag{S5'}$$

$$\vee\, (\sigma^{id}_{pc} - 1, \mathtt{comm}_s\, id\, c\, x_s) \in cuc^{id} \wedge (\hat{\sigma}^{id}_{pc}, \mathtt{write}\, m_c\, \mathrm{FREE}) \in sv^{id} \wedge \psi(\sigma^{id}_{pc} - 1) + 6 = \hat{\sigma}^{id}_{pc}$$

$$\wedge\, \mathcal{X}(c) = id_{un} \wedge \Gamma(m_c) = id \wedge \neg\Gamma(sr_c) \wedge \neg\Gamma(fr_c) \tag{S6}$$

receive:

$$\vee\, (\sigma^{id}_{pc}, \mathtt{comm}_r\, id\, c\, x_r) \in cuc^{id} \wedge (\hat{\sigma}^{id}_{pc}, \mathtt{cas}\, ss_c\, sr_c \top \bot) \in sv^{id} \wedge \psi(\sigma^{id}_{pc}) = \hat{\sigma}^{id}_{pc} \wedge id \notin \mathcal{X} \tag{R}$$

$$\vee\, (\sigma^{id}_{pc}, \mathtt{comm}_r\, id\, c\, x_r) \in cuc^{id} \wedge (\hat{\sigma}^{id}_{pc}, \mathtt{cbr}\, ss_c\, sr_c \top \bot) \in sv^{id} \wedge \psi(\sigma^{id}_{pc}) + 1 = \hat{\sigma}^{id}_{pc} \wedge$$

$$\Big(\hat{\sigma}^{id}_{ds}(ss_c) \wedge \Big(\exists\, id_s.\, \mathcal{X}(c) = (id_s, id)_{in}\Big) \vee \neg\hat{\sigma}^{id}_{ds}(ss_c) \wedge id \notin \mathcal{X}\Big) \tag{R1}$$

$$\vee\, (\sigma^{id}_{pc}, \mathtt{comm}_r\, id\, c\, x_r) \in cuc^{id} \wedge (\hat{\sigma}^{id}_{pc}, \mathtt{read}\, x_r\, \gamma_c) \in sv^{id} \wedge \psi(\sigma^{id}_{pc}) + 2 = \hat{\sigma}^{id}_{pc} \wedge$$

$$\Big(\exists\, id_s.\, \mathcal{X}(c) = (id_s, id)_{in}\Big) \tag{R2}$$

$$\vee\, (\sigma^{id}_{pc} - 1, \mathtt{comm}_r\, id\, c\, x_r) \in cuc^{id} \wedge (\hat{\sigma}^{id}_{pc}, \mathtt{write}\, fr_c \top) \in sv^{id} \wedge \psi(\sigma^{id}_{pc} - 1) + 3 = \hat{\sigma}^{id}_{pc} \wedge$$

$$\Big(\exists\, id_s.\, \mathcal{X}(c) = (id_s, id)_{un}\Big) \tag{R3}$$

A.3 Proof: Fitting Implies Handshake Refinement

In this section, we prove that all fitting pairs of CUC and SV programs are in a handshake refinement relation. First, we restate Theorem 6.3 and recall the flow of the protocol, as it indicates the transitions between the disjuncts of $\mathcal{P}^{id}_{cuc,sv,\psi}$. Finally, we restate Definition 6.19 of the handshake refinement and prove the theorem.

Theorem 6.3: Fitting Implies Handshake Refinement

Let *sv* be a program fitting *cuc* with the program label map 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, and all signals are inactive.

$$\sigma \mathrel{\widehat{=}} \hat{\sigma} \land \big(\forall\, id.\ \hat{\sigma}^{id}_{pc} = \psi(\sigma^{id}_{pc})\big) \land \big(\forall\, c.\ \Gamma(m_c) = \textsc{free} \land \neg\Gamma(sr_c) \land \neg\Gamma(fr_c)\big)$$
$$\implies \big(\sigma, \emptyset, (\Gamma, \hat{\sigma})\big) \in \mathcal{B}_{cuc,sv,\psi}$$

In Figure 4, we depict the labeled transitions of the protocol. In contrast to Figure 2, which also depicts the flow of the protocol, we show the events as labels and not the instructions. The figure is helpful to visualize how a component passed the disjuncts of the protocol constraints $\mathcal{P}^{id}_{cuc,sv,\psi}$. We recall that (N) is the disjunction of (O), (D), (C), (S), and (R) from the definition of $\mathcal{P}^{id}_{cuc,sv,\psi}$. In (N), the beginning of next instruction implementation, the program counters match with ψ and the current *id* does not occur in the lockstate (cf. Definition A.15). The arrows over (S1), (S5'), and (R1) denote whether `cbr` will jump back to the first label or forward to the second label, based on the `cas` instruction before. Note that *send* cannot progress until the end, until the *receive* reads the value. The dotted transitions from (S4) to (S4') and from (S5) to (S5') indicate that the applying/valid disjuncts change for the sender component, when the receiver component takes the transition from (R2) to (R3).

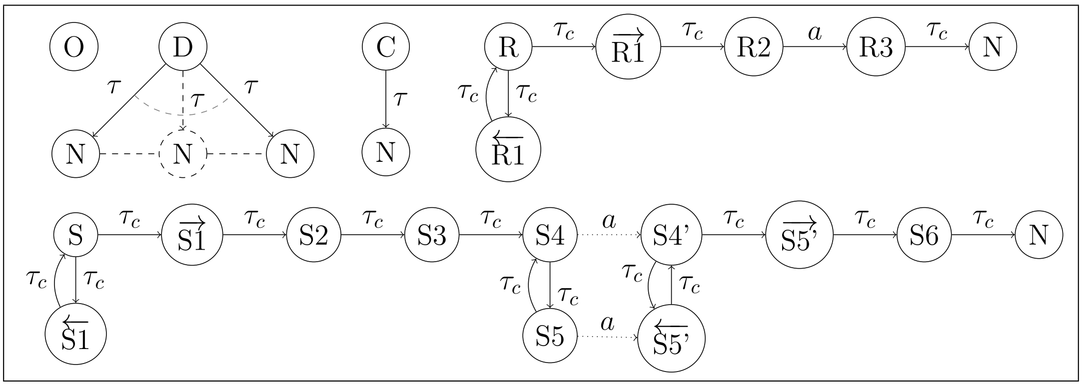

Figure 4: *sv* Transitions Between the Disjuncts of $\mathcal{P}^{id}_{cuc,sv,\psi}$

Definition 6.19: Handshake Refinement $\mathcal{B}_{cuc,sv,\psi}$

Let a CUC program *cuc* and an SV program *sv* be fitting with a program label map ψ. A **handshake refinement** is a ternary relation $\mathcal{B}_{cuc,sv,\psi}$ over CUC states (*cuc*), channel-states ($\mathcal{X}$), and SV states $\big((\Gamma, \hat{\sigma})\big)$, which fulfills the following properties.

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

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

Protocol constraints: $\mathcal{P}_{cuc,sv,\psi}\big(\sigma, \mathcal{X}, (\Gamma, \hat{\sigma})\big)$ (see Definition 6.20)

Down-simulation:

$\forall\, ev\ \sigma'.\ ev \neq \tau \wedge \mathcal{X}(chan(ev)) = \textsc{free} \wedge \sigma \xrightarrow{ev}_{cuc} \sigma' \implies \exists\, \Gamma'\ \hat{\sigma}'\ id_s\ id_r\ \mathcal{X}'.$

$\quad (\Gamma, \hat{\sigma}) \xrightarrow{\tau_c}{}^{*}_{sv} \xrightarrow{ev}_{sv} (\Gamma', \hat{\sigma}') \wedge \mathcal{X}'(chan(ev)) = (id_s, id_r)_{un} \wedge \big(\sigma', \mathcal{X}', (\Gamma', \hat{\sigma}')\big) \in \mathcal{B}_{cuc,sv,\psi}$

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

Up-simulation:

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

$\forall\, ev\ (\Gamma', \hat{\sigma}').\ (\Gamma, \hat{\sigma}) \xrightarrow{ev}_{sv} (\Gamma', \hat{\sigma}') \implies \exists\, \sigma'\ \mathcal{X}'.\ \sigma \xrightarrow{ev}_{cuc} \sigma' \wedge \big(\sigma', \mathcal{X}', (\Gamma', \hat{\sigma}')\big) \in \mathcal{B}_{cuc,sv,\psi}$

Unlocking-simulation:

$\exists\, c\ id_s.\ \mathcal{X}(c) = (id_s)_{un} \vee \big(\exists\, id_r.\ \mathcal{X}(c) = (id_s, id_r)_{un}\big) \implies$

$\quad \exists\, \Gamma'\ \hat{\sigma}'\ \mathcal{X}'.\ (\Gamma, \hat{\sigma}) \xrightarrow{\tau_c}{}^{*}_{sv} (\Gamma', \hat{\sigma}') \wedge \mathcal{X}' = \mathcal{X}[c \coloneqq \textsc{free}] \wedge \big(\sigma, \mathcal{X}', (\Gamma', \hat{\sigma}')\big) \in \mathcal{B}_{cuc,sv,\psi}$

Proof: Theorem 6.3 (Fitting Implies Handshake Refinement)

To prove Theorem 6.3, we define a relation $\mathcal{B}$, show that it contains $\big(\sigma, \mathcal{X}, (\Gamma, \hat{\sigma})\big)$, and show that it is a handshake refinement (even the largest). We use

$$\mathcal{I}\big(\sigma, \mathcal{X}, (\Gamma, \hat{\sigma})\big) \coloneqq \sigma \mathrel{\widehat{=}} \hat{\sigma} \wedge \mathcal{P}_{cuc,sv,\psi}\big(\sigma, \mathcal{X}, (\Gamma, \hat{\sigma})\big)$$

$$\mathcal{B} \coloneqq \Big\{ \big(\sigma, \mathcal{X}, (\Gamma, \hat{\sigma})\big) \,\Big|\, \mathcal{I}\big(\sigma, \mathcal{X}, (\Gamma, \hat{\sigma})\big) \Big\}$$

as an invariant and induction hypothesis. The proof consists of two parts:

1) We show that the initial states are in $\mathcal{B}$.

2) We show that $\mathcal{B}$ is a handshake refinement, i. e., every triplet in $\mathcal{B}$ also satisfies the down-, up-, and unlocking-simulations, i. e., the possible successor triplets are again in $\mathcal{B}$.

Proof: Theorem 6.3 (Fitting Implies Handshake Refinement)

---

1) $\big(\sigma, \emptyset, (\Gamma, \hat{\sigma})\big) \in \mathcal{B}$:

Assumptions:

I) $\sigma \mathrel{\widehat{=}} \hat{\sigma}$
II) $\big(\forall\, id.\ \hat{\sigma}^{id}_{pc} = \psi(\sigma^{id}_{pc})\big)$
III) $\big(\forall\, c.\ \Gamma(m_c) = \textsc{free} \wedge \neg\Gamma(sr_c) \wedge \neg\Gamma(fr_c)\big)$

Want to show (goal):
$\mathcal{I}\big(\sigma, \emptyset, (\Gamma, \hat{\sigma})\big)$, i. e., $\sigma \mathrel{\widehat{=}} \hat{\sigma} \wedge \mathcal{P}_{cuc,sv,\psi}\big(\sigma, \emptyset, (\Gamma, \hat{\sigma})\big)$

Proof:
$\sigma \mathrel{\widehat{=}} \hat{\sigma}$ holds by I).
To show that $\mathcal{P}_{cuc,sv,\psi}\big(\sigma, \emptyset, (\Gamma, \hat{\sigma})\big)$ holds, we have $\neg\Gamma(sr_c) \wedge \neg\Gamma(fr_c)$ from III) and show $\mathcal{P}^{id}_{cuc,sv,\psi}\big(\sigma, \emptyset, (\Gamma, \hat{\sigma})\big)$ for an arbitrary but fixed *id*.
From $\mathcal{X} = \emptyset$ we have $id \notin \mathcal{X}$.
Together with II) we conclude that (N) holds by case distinction over the definition of $\mathcal{P}^{id}_{cuc,sv,\psi}$.

So our initial triplet $\big(\sigma, \emptyset, (\Gamma, \hat{\sigma})\big)$ is an element of $\mathcal{B}$.

---

2) $\mathcal{B}$ is a handshake refinement:

Want to show (goal):
$\mathcal{B}$ fulfills the definitions of the down-, up-, and unlocking-simulation.

Proof:
We fix a component and its *id* and go through all cases of $\mathcal{P}^{id}_{cuc,sv,\psi}$. To be able to look at each component individually, we ensure that we only write to our own local state and that we only assign our id to $\mathcal{X}(c)$ if it was FREE, or add it as a receiver. Also, we may only set $\mathcal{X}(c)$ to unlocking, if we were assigned as a receiver. Furthermore, we may never write to a mutex that is not FREE (ensured by using `cas`), and never write to a shared variable without having the mutex (ensured by $\mathcal{X}(c)$ = own id). All these properties follow from Definition A.15. By doing so, we ensure that no other $\mathcal{P}^{id'}_{cuc,sv,\psi}$ with $id' \neq id$ is changed, unless mentioned. We show, where applicable that the down-, up-, and unlocking-simulations are satisfied, i. e., that the successor triplets again satisfy $\mathcal{I}$, and are thereby in $\mathcal{B}$. For the up- and the down-simulation, we consider in detail that the same event can be communicated.

The **up-simulation** applies in every disjunct of $\mathcal{P}_{cuc,sv,\psi}$. Most cases are simple applications of the SV semantics. Only in (R2) we need additionally that $\mathcal{X}(c) = (id_s, \_)_{in}$ implies that there is a sender waiting, i. e., a component for which (S4) or (S5) holds, to show that *cuc* can communicate the same event.
We prove that *cuc* can communicate the same event:

Proof: Theorem 6.3 (Fitting Implies Handshake Refinement)

We consider the receiver, thus, let $id_r \coloneqq id$.
In (R2) sv communicates the event $ev = c.\Gamma(m_c).id_r.\Gamma(\gamma_c)$, according to the event labeling function (cf. Definition 6.11).
By case analysis of the induction hypothesis $\mathcal{I}$, we show that $\mathcal{X}(c) = (id_s, \_)$ implies that there exists a sender id_s for which (S4) or (S5) holds, and in particular $\Gamma(\gamma_c) = \hat{\sigma}^{id_s}(x_s)$ and $\left(\sigma_{pc}^{id_s}, \texttt{comm}_s\ id_s\ c\ x_s\right) \in cuc^{id_s}$.
Together with $\left(\sigma_{pc}^{id_r}, \texttt{comm}_r\ id_r\ c\ x_r\right) \in cuc^{id_r}$, we have that cuc can synchronize on the event $c.id_s.id_r.\sigma^{id_s}(x_s)$.
With $\Gamma(\gamma_c) = \hat{\sigma}^{id_s}(x_s)$ from (S4) $\vee$ (S5) and $\sigma \mathrel{\widehat{=}} \hat{\sigma}$ we have $\Gamma(\gamma_c) = \sigma^{id_s}(x_s)$.
Together with $\Gamma(m_c) = id_s$ from (R2), we show $c.\Gamma(m_c).id_r.\Gamma(\gamma_c) = c.id_s.id_r.\sigma^{id_s}(x_s)$.
Thus, σ can perform the same event as $\hat{\sigma}$.
After the transition, (R3) holds for the receiver and (S4') or (S5') holds for the sender, i. e.,, the successor state satisfies $\mathcal{I}$ and is in $\mathcal{B}$.

The **down-simulation** applies only where (N) holds. In case of the visible event (`read`), as both a sender id_s and a receiver id_r are ready, we are free to pick an execution of the protocol, e. g., passing (S), (S1), (S2), (S3), (S4) for id_s, and then (R), (R1), (R2), (R3) for id_r.
We prove that sv can communicate the same event:
From the facts that cuc communicates $ev = c.id_s.id_r.\sigma^{id_s}(x_s)$ and the assumption $\mathcal{X}(c) = \textsc{free}$ from the down-simulation, we conclude that (S) holds for id_s as well as (R) for id_r.
Furthermore, from $\mathcal{X}(c) = \textsc{free}$ we know that the channel is free. Thus, we are free to pick an execution of the protocol until we communicate the event. The execution passes the disjuncts in the following sequence: (S), (S1), (S2), (S3), (S4) for id_s, and then (R), (R1), (R2), (R3) for id_r. As the communication of the event transitions (S4) to (S4'), we end up with (S4) for id_s and (R2) for id_r right before the event is communicated and (S4') and (R3) for the successor triplet. As in the up-simulation in the case of (R2), we can show that the events communicated in sv and cuc are the same and the successor state satisfies $\mathcal{I}$ and is in $\mathcal{B}$.

The **unlocking-simulation** applies only after the visible event was communicated, i. e., in (S4'), (S5'), (S6), (R3). Again, we are free to pick an execution of the protocol. The transition from (R3) to (N) should be taken first.

☐

A.4 Refusals imply Refusals

Lemma 6.4: Refusals in *sv* Imply Refusals in *cuc*

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

Proof: Lemma 6.4 (Refusals in *sv* Imply Refusals in *cuc*)

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

Want to show: $(\Gamma, \hat{\sigma}) \text{ ref}_{sv}\ X \implies \sigma \text{ ref}_{cuc}\ X$
Unfold $\text{ref}_{sv/cuc}$: $\forall\, a \in X.\ \neg\big((\Gamma, \hat{\sigma}) \xrightarrow{\tau_c}{}^{*}_{sv} \xrightarrow{a}{}_{sv}\big) \implies \forall\, a \in X.\ \neg\big(\sigma \xrightarrow{a}{}_{cuc}\big)$
If $X = \{\}$, this is true. Assume $X \neq \{\}$.
Pick $a \in X$, insert in assumption: $\neg\big((\Gamma, \hat{\sigma}) \xrightarrow{\tau_c}{}^{*}_{sv} \xrightarrow{a}{}_{sv}\big) \implies \neg\big(\sigma \xrightarrow{a}{}_{cuc}\big)$
Negation: $\sigma \xrightarrow{a}{}_{cuc} \implies (\Gamma, \hat{\sigma}) \xrightarrow{\tau_c}{}^{*}_{sv} \xrightarrow{a}{}_{sv}$
This is implied by the down-simulation, as we have $\mathcal{X} = \emptyset$ from Lemma 6.3. $\square$

References

[Ber19] Nils Berg. *Formal Verification of Low-Level Code in a Model-Based Refinement Process.* PhD thesis, Technische Universität Berlin, 2019. `doi:http://dx.doi.org/10.14279/depositonce-8638`.

[BGDG18] Nils Berg, Thomas Göthel, Armin Danziger, and Sabine Glesner. Preserving liveness guarantees from synchronous communication to asynchronous unstructured low-level languages. In Jing Sun and Meng Sun, editors, *Formal Methods and Software Engineering - 20th International Conference on Formal Engineering Methods, ICFEM 2018, Gold Coast, QLD, Australia, November 12-16, 2018, Proceedings*, volume 11232 of *Lecture Notes in Computer Science*, pages 303–319. Springer, 2018. `doi:10.1007/978-3-030-02450-5_18`.

[Bou88] Luc Bougé. On the existence of symmetric algorithms to find leaders in networks of communicating sequential processes. *Acta Inf.*, 25(2):179–201, 1988. `doi:10.1007/BF00263584`.