Technische Universität Berlin

Formal Verification of Low-Level Code in a Model-Based Refinement Process

vorgelegt von
Dipl.-Math. Nils Erik Berg
(geb. Jähnig)

von der Fakultät IV - Elektrotechnik und Informatik
der Technischen Universität Berlin
zur Erlangung des akademischen Grades
Doktor der Naturwissenschaften
– Dr. rer. nat. –
genehmigte Dissertation

Promotionsausschuss:

Vorsitzender:	Prof. Dr. Axel Küpper
Gutachterin:	Prof. Dr. Sabine Glesner
Gutachterin:	Prof. Dr. Barbara König
Gutachter:	Prof. Dr. Gerald Lüttgen

Tag der wissenschaftlichen Aussprache: 10. Mai 2019

Berlin 2019

Abstract

Embedded systems often control safety critical environments, such as cars, airplanes, traffic control, or pacemakers. Embedded systems are often non-terminating and communicating. To confine the complexity of those systems (e.g. introduced by communication details), comprehensive verification and simulation is usually done on an abstract model (e.g. a formal specification) instead of the implemented system itself. However, it is still an open problem how to relate the actual executable low-level code with the abstract models.

In industrial practice, embedded systems in safety critical areas are commonly designed with model-based approaches. The model-based development of (embedded) systems starts with the verification (or simulation) of properties on an abstract model. There are two important categories of properties: *safety* properties, which exclude bad behavior, and *liveness* properties, which ensure good behavior. The abstract model is implemented and/or compiled to executable low-level code, but there is a large verification gap between the model whose properties are verified and the system that is actually executed. Both manual implementation and the compilation process are complex and error prone. Furthermore, complexity of the verification rises even higher when concurrent systems that communicate with each other are developed.

We address the problem that the executed low-level code is not guaranteed to have the same safety and liveness properties as the abstract model.

To overcome this problem, we present a formal framework that enables rigorous verification of preservation of safety and liveness properties from an abstract model to low-level code. The key idea of our approach is to divide the relation between abstract model and low-level code into two steps to achieve *compositionality*:

- A compositional relation between the abstract model and an intermediate model, and
- a non-compositional relation between the intermediate model and the low-level code.

This separation enables us to reason compositionally about the first half, relating the abstract model and the intermediate model, and use a general theorem for the second half, which results in an overall compositional verification.

With our framework, we enable the verification of the conformance of a program in low-level code to its *Communicating Sequential Processes* (CSP) specification, i. e., that all safety and liveness properties are preserved. To this end, we have transferred and extended the notion of CSP refinement to also cover low-level code. To separate the verification into two steps, we define our low-level language with abstract communication *Communicating Unstructured Code* (CUC) as intermediate model. To relate the specification in CSP and the intermediate model in CUC, we present a Hoare calculus which allows us to reason over the communication behavior of a (possibly non-terminating) CUC program. To relate the intermediate model in CUC with our low-level language *Shared Variables* (SV), we define our notion of handshake refinement and prove in a general theorem the preservation of both safety and liveness properties for related CUC and SV programs. Our formalization in a theorem prover enables users to mechanize and reuse their proofs. Together with the compositionality of our approach, this makes it possible to provide rigorous guarantees of concurrent low-level programs in a way that scales with the number of components.

Zusammenfassung

Eingebettete Systeme steuern häufig sicherheitskritische Umgebungen wie z.B. Autos, Flugzeuge, Verkehrssteuerungen oder Herzschrittmacher. Eingebettete Systeme sind oft nicht-terminierend und kommunizieren miteinander. Um die Komplexität dieser Systeme (die z.B. durch Kommunikationsdetails entsteht) in den Griff zu bekommen, werden normalerweise abstraktere Modelle (z.B. in Form einer formalen Spezifikation) zur Verifikation oder Simulation herangezogen anstelle des Systems selbst. Den ausführbaren Low-Level-Code mit den abstrakten Modellen in Beziehung zu setzen ist ein noch ungelöstes Problem.

In der industriellen Praxis werden eingebettete Systeme in sicherheitskritischen Bereichen im Allgemeinen mit modellbasierten Ansätzen entworfen. Die modellbasierte Entwicklung von (eingebetteten) Systemen beginnt mit der Verifikation (oder Simulation) von Eigenschaften eines abstrakten Modells. Es gibt zwei wichtige Kategorien von Eigenschaften: Sicherheitseigenschaften, die unerwünschtes Verhalten ausschließen, und Lebendigkeitseigenschaften, die erwünschtes Verhalten gewährleisten. Das abstrakte Modell wird durch implementieren und/oder kompilieren in ausführbaren Low-Level-Code überführt. Jedoch ist der Erhalt der Eigenschaften vom abstrakten Modell hin zum tatsächlich ausgeführten System nicht gesichert; es entsteht die sogenannte Verifikationslücke.

Um diese Verifikationslücke zu schließen, stellen wir ein formales Framework vor, das die rigorose Verifikation der Erhaltung der Sicherheits- und Lebendigkeitseigenschaften von einem abstrakten Modell bis hin zum Low-Level-Code ermöglicht. Die Kernidee unseres Ansatzes besteht darin, die Beziehung zwischen dem abstrakten Modell und dem Low-Level-Code in zwei Schritte zu unterteilen, um so *Kompositionalität* zu erreichen:

- Eine Relation zwischen dem abstrakten Modell und einem Zwischenmodell und
- eine weitere Relation zwischen dem Zwischenmodell und dem Low-Level-Code.

Diese Unterteilung erlaubt es uns, kompositional über Relation vom abstrakten Modell zum Zwischenmodell zu argumentieren, und ein allgemeingültiges Theorem für die zweite Hälfte zu verwenden. Insgesamt ist so eine kompositionale Verifikation möglich.

Unser Framework ermöglicht die Verifikation der Konformität eines Programms in Low-Level-Code zu seiner Spezifikation in *Communicating Sequential Processes* (CSP), d.h., dass alle Sicherheits- und Lebendigkeitseigenschaften erhalten bleiben. Zu diesem Zweck haben wir den Begriff der CSP-Verfeinerung auf Low-Level-Code übertragen und erweitert. Um die Verifizierung in zwei Schritte zu unterteilen, definieren wir als Zwischenmodell unsere Low-Level Sprache mit abstrakter Kommunikation *Communicating Unstuctured Code* (CUC). Um die Spezifikation in CSP und das Zwischenmodell in CUC in Beziehung zu setzen, stellen wir einen Hoare-Kalkül vor, der die Verifikation von Kommunikationsverhalten eines (möglicherweise nicht-terminierenden) CUC-Programms ermöglicht. Um das Zwischenmodell in CUC mit unserer Low-Level Sprache *Shared Variables* (SV) in Verbindung zu bringen, definieren wir unseren Begriff der Handshake-Verfeinerung und beweisen in einem allgemeingültigen Theorem den Erhalt von Sicherheits- und Lebendigkeitseigenschaften für in Beziehung stehende CUC- und SV-Programme. Unsere Formalisierung in einem Theorembeweiser ermöglicht es dem Benutzer, Beweise zu mechanisieren und wiederzuverwenden. Zusammen mit der Kompositionalität unseres Ansatzes ermöglichen wir, rigoros Eigenschaften für Low-Level-Programme zu garantieren, und zwar gut skalierbar mit der Anzahl der nebenläufigen Komponenten des Systems.

Acknowledgment

This work has been developed in the Software and Embedded Systems Engineering group at Technische Universität Berlin in the context of the DFG funded VATES project.

There are a lot of people without whom this thesis would not have been started, continued or finished.

First, I want to thank my supervisor Prof. Dr. Sabine Glesner, who made it all possible, and my reviewers Prof. Dr. Barbara König and Prof. Dr. Gerald Lüttgen for taking the time and giving me valuable feedback.

All people at the SESE group (past and present) were a blast to work with and always had time to discuss work-related and personal matters. Thank you all! In particular, I was lucky to discuss my work at great lengths with our three post-docs: Dr. Thomas Göthel, Dr. Björn Bartels, and (meanwhile) Prof. Dr. Paula Herber. Their feedback, inspirations, pep talks and the sheer amount of time they spent for and with me was invaluable for me.

Last but not least, I want to thank my family. My wife Mascha helped me at every stage of my thesis and took care of the non-thesis part of my life (while working on her own thesis!). With respect to my children Leo and Wim, sincerely I hope that I will never have as little time for them as I had in the last six months of finishing this thesis. And finally, I want to thank my parents, who let me always do what I want and supported me.

Contents

Chapter 1

Introduction

Embedded systems often control safety critical environments, such as cars, airplanes, traffic control, or pacemakers. Embedded systems are often non-terminating and communicating. To confine the complexity of those systems (e.g. introduced by communication details), comprehensive verification and simulation is usually done on an abstract model (e.g. a formal specification) instead of the implemented system itself. However, it is still an open problem how to relate the actual executable low-level code with the abstract models.

In industrial practice, embedded systems in safety critical areas are commonly designed with model-based approaches. A prominent example is the automotive industry, where official regulations such as the ISO 26262 [ISO09] prescribe the use of model-based development. The model-based development of (embedded) systems starts with the verification (or simulation) of properties on an abstract model. There are two important categories of properties: *safety* properties, which exclude bad behavior, and *liveness* properties, which ensure good behavior. The abstract model is implemented and/or compiled to executable low-level code, but there is a large verification gap between the model whose properties are verified and the system that is actually executed. Both the manual implementation and the compilation process are complex and error prone. Furthermore, complexity of the verification becomes even more challenging when concurrent systems that communicate with each other are developed.

In this thesis, we address the problem that the executed low-level code is not guaranteed to have at least the same safety and liveness properties as the abstract model. This is still an open problem, due to the following reasons:

- The complexity and the inherent non-compositional semantics of communication protocols hinder scalability, due to shared resources, which affect multiple components.
- The change of abstraction level from abstract communication to a communication protocol and the structural difference between the model, which has a higher level structure, and low-level code, which is unstructured, entail different models of computation, which need to be related.
- The infinite or very large state spaces that may arise from non-termination or from reading data from external sources require special care to be represented in a finite

form that allows for compositional treatment.

The aim of this thesis is to develop a formal framework that enables rigorous verification of preservation of safety and liveness properties from an abstract model to a low-level model, i. e., the semantic representation of low-level code. We require the formal framework to fulfill the following criteria:

- Our framework should enable *model-based development*.
- To minimize the verification gap, our framework should be able to relate an abstract model with a *low-level model*, which contains the typical constructs of executable low-level code, such as conditional branches and shared variable communication.
- Our framework should ensure the preservation of both *safety* and *liveness* properties.
- The framework should be applicable to systems that consist of *concurrent* and *communicating* components.
- Due to the high complexity of concurrency and communication, we require our verification to be *scalable* with respect to concurrent combination.
- Our framework is required to be able to handle *infinite or very large state spaces*, arising from non-termination of programs or reading of data from external sources.
- We require our framework to be *rigorous*, i. e., to preclude the manual introduction of errors.
- Our framework should facilitate *reuse* of the results and enable (semi-) automation.

To achieve the objectives defined above, we propose a formal framework to relate a low-level model with an abstract model. The key idea of our approach is to divide the relation between abstract model and low-level model into two steps to achieve *compositionality*:

- A compositional relation between the abstract model and an intermediate model, and
- a non-compositional relation between the intermediate model and the low-level model. We show this relation for a class of programs in a general theorem, resulting in an overall compositional verification.

This separation enables us to reason compositionally about the first half, relating the abstract model and the intermediate model, and provide a general theorem for the second half. As we prove the general theorem for all programs of the considered class (which uses a verified channel implementation), this results in compositional verification.

Our approach and the resulting framework can be divided into five parts, which are shown in Figure 1.1.

L1) The abstract model is defined in the language CSP (Communicating Sequential Processes), which is a process algebra capable of describing *concurrent* and *non-terminating* systems with compositional semantics. The notion of CSP's stable failures refinement

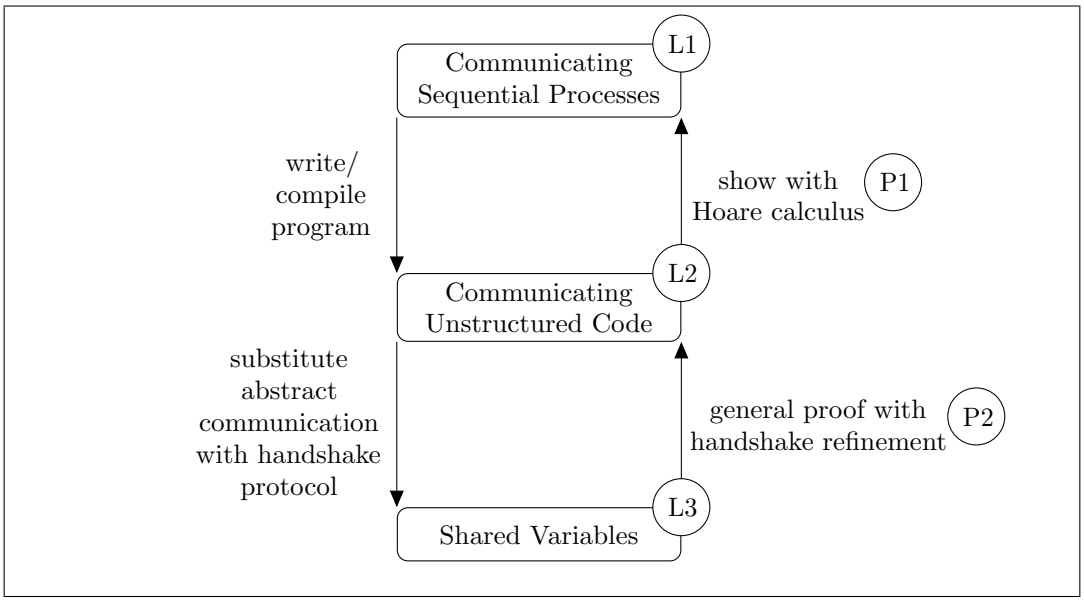


Figure 1.1: Overview

allows one to relate processes by their behaviors and enables an iterative *model-based development* approach. CSP refinement is compositional and preserves *safety* and *liveness* properties. We propose to extend the stable failures refinement from CSP to a low-level model.

L2) To define intermediate models, we propose the language CUC (Communicating Unstructured Code), alongside formal semantics for it. It is close to low-level code but has an abstract communication primitive. The communication semantics is similar to the communication semantics of CSP, which enables the inheritance of its compositionality. Otherwise, the semantics captures the challenging unstructured nature of low-level code.

L3) To define low-level models, we propose the language SV (for "shared variables") in which any communication employing shared variables can be modeled, together with formal low-level semantics for it. The semantics is precise yet general and can be adapted to, e. g., a specific memory model. The semantics can model a large and relevant subset of current instruction set architectures, e. g., RISC-V.

P1) We define a Hoare calculus to facilitate compositional reasoning about the relation of a CSP model and a CUC model. As this part requires interaction of the designer/developer, we formalize it in the theorem prover Isabelle/HOL to ensure the rigorousness of our framework.

P2) We instantiate the abstract communication from CUC to a concrete protocol in SV and give a general theorem that any SV program using the protocol has the same safety and liveness properties as its CUC counterpart. The main challenge for the instantiation of the abstract communication instruction with a concrete communication protocol is to

relate the synchronous communication from CSP/CUC to the asynchronous communication over shared variables in a way that preserves safety and liveness properties. Because of the different abstraction levels, ordinary CSP refinement is not sufficient. To this end, we propose our notion of handshake refinement, which is an asymmetric relation that takes into account the implementation of synchronous communication. It allows decisions to be distributed over execution steps and components. It is weak enough to relate those two behaviors but still strong enough to preserve safety and liveness properties. As this part does not require the interaction of the designer/developer, we decided to not formalize it in a theorem prover and give concise proofs containing the essential ideas.

The main contribution of this thesis is a framework for compositional verification of the refinement relation between a specification in CSP and an implementation in SV. This includes:

- Our two-part approach of the framework, allowing for compositional handling of shared variable communication.
- The definition of our low-level language CUC with abstract communication and its operational, traces and stable failure semantics.
- A Hoare calculus which allows us to reason about the communication behavior of a (possibly non-terminating) CUC program.
- Our notion of handshake refinement together with a proof that the handshake refinement preserves both safety and liveness properties.
- A general proof that a CUC program and an SV program using the handshake protocol are in a handshake refinement relation.

Prior to this thesis, we have published a precursor to CUC without communication at *NFM'14* [BJ14] and CUC itself as well as its operational and traces semantics at *FESCA'15* [JGG15]. The stables failures semantics, the Hoare calculus and the first part of the presented framework are published at *SEFM'16* [JGG16]. The language SV with operational semantics, the handshake refinement and the verification of the protocol for all instances are published at *ICFEM'18* [BGDG18]. In the publications at *FESCA'15*, *SEFM'16*, and *ICFEM'18* I am the main author. In the publication at *NFM'14* I am a coauthor with about 50% of own contributions, mainly the formalization.

We give short summaries to the remaining chapters of this thesis:

In Chapter 2, we cover the background for this thesis. We explain our understanding of low-level code by introducing the term Instruction Set Architecture. We introduce CSP with its syntax and semantics. We give the definitions for the relations between CSP processes that are relevant in the context of this thesis, namely bisimulations and refinements. Closely related, the concepts of safety and liveness properties as we use them are defined. Finally, we briefly introduce the verification tools used in this thesis, namely Isabelle/HOL and FDR4.

Chapter 3 gives on overview of related work. We cover low-level semantics aimed at verification, compositional proof calculi for concurrent systems, comparisons of synchrony and asynchrony, liveness-preserving relations, and implementations of synchronous communication. The liveness-preserving relations are inspirations to our handshake refinement.

In Chapter 4, we describe our approach and the resulting two-part framework in more detail and consider the role of each part within the framework. We focus on the compositionality of the overall framework.

Chapter 5 describes the first part of the framework, where an abstract specification is related to a low-level language with an abstract communication mechanism. To this end, we discuss our understanding of a low-level code model and define CUC and its various semantics. We show the correspondence between the various semantics and their conformance to the CSP semantics. Our Hoare calculus is introduced. Finally, we present how to show a stable failures refinement between a CSP process and a CUC program.

Chapter 6 describes the second part of the framework, where the abstract communication is related to its implementation with a protocol. To this end, we define SV and its semantics. We use an exemplary handshake protocol to implement the abstract communication. The core of this chapter is the definition of the handshake refinement to verify the protocol. The important theorems assert that the handshake refinement preserves safety and liveness properties as well as the fact that all "fitting" CUC and SV programs are related by the handshake refinement.

In Chapter 7, we evaluate our framework by applying it to a system which is concurrent, communicating and non-terminating. As an exemplary system we choose redundant m-servers-n-clients: n clients request computations from m (in our case three) redundant server components. We show a stable failures refinement between the abstract specification of the system and its low-level implementation.

Chapter 8 summarizes and discusses the achievements of this thesis and gives an outlook on future work.

Chapter 2

Background

In this chapter, we introduce the background that is the basis for our framework. In Section 2.1, we introduce the concept of low-level code and define the term instruction set architecture. The introduced concept of low-level code is the basis for our low-level code model, which is described in 5.1. In Section 2.2, we define the notions of simulation and bisimulation as traditional means to relate systems, possibly at different levels of abstraction. Simulations and bisimulations are the basis of our handshake refinement in Chapter 6.1. In Section 2.3, we introduce the process algebra *Communicating Sequential Processes* (CSP) with its syntax and its semantics. We cover the operational semantics and two denotational semantics in detail: The traces semantics and the stable failures semantics. The denotational semantics enables the notion of CSP refinement, which we also introduce. In our approach, we use CSP as a specification language. Our intermediate language CUC (defined in Chapter 5) uses the communication mechanism of CSP and defines a low-level code model in all other aspects. Finally, we briefly introduce the theorem prover Isabelle/HOL in Section 2.4. We have formalized the results from Chapter 5 in Isabelle/HOL.

2.1 Low-Level Code

In this section, we define and explain the term *Instruction Set Architecture* (ISA). The ISA defines the machine code or assembly language, i. e., the instructions a processor can execute. It is the interface between software and hardware, and as such, the instructions defined by an ISA form the lowest level of software. We use the ISA as a basis to define our low-level code model in Section 5.1. The content of this section is mainly based on [Pag09, HP17].

An ISA defines an instruction set, the semantics for the instructions and also the number of general purpose registers. Additional data can be stored in the memory. We give an example of machine code and how it is executed and then briefly discuss various aspects of ISAs. A low-level program is a sequence of instructions stored in memory. A *program counter* (*pc*), a special register, points to the instruction which is to be executed next. As an example, we use the machine code presented in Figure 2.1, which uses the ISA RISC-V [WA17]. RISC-V is a recent open source instruction set architecture developed at UC Berkeley. In Figure 2.1, each line of code starts with the address of the instruction in memory. This is

```
1: ld %r1 %a            ## load the content from memory address a
                           into register r1
2: ld %r2 %b            ## load the content from memory address b
                           into register r2
3: add %r3 %r1 %r2      ## add the content of registers r1 and r2
                           and store the result in register r3
4: sd %r3 %c            ## store the value from register r3
                           at the memory address c
5: halt                 ## termination of the program
```

Figure 2.1: Example Program in Machine Code: Addition of Two Values from Memory

usually not part of the machine code, but implicitly given by the placement of the program in memory. We write it here for clarity. Next is the name of the instruction, followed by the arguments. The name of the first instruction of the example is `ld`, short for *load double*, as it loads a *double word* (64 bits) from the memory address `%a` to the register `%r1`. `%a` and `%r1` specify addresses. When the program is executed, the instruction to which the *pc* points to is first fetched, then the program counter is increased, and then the fetched instruction is executed. Finally, the results are written to the registers and/or the memory. This cycle is repeated until the fetched instruction is `halt`. Apart from data transfer instructions (load and store) and arithmetic and logic instruction (in this example add), there are also *control flow* instructions. Control flow instructions can change the value of *pc* to allow for a non-linear execution of a program, e. g., loops or branches. Interesting for the context of this thesis are also *multi-processor synchronization* instructions that allow multiple processors to work concurrently on a shared memory. Multi-processor synchronization instructions allow for two sequential actions in an atomic operation, i. e., without the interference of another process. They test whether a shared variable has a desired value or is unchanged and then subsequently modify the variable.

Closely related to the ISA is the *micro architecture* (or hardware), which specifies the actual implementation of an ISA. The same ISA can be implemented by different micro architectures. There are several different ISAs, differing in many aspects. Two common categories of ISAs are *Complex Instruction Set Computers* (CISC) and *Reduced Instruction Set Computers* (RISC). Both approaches differ in various aspects of their ISA.

CISC architectures (such as the x86 architecture commonly used for desktop machines), have instructions of variable length, which execute complex tasks. For example, a single CISC instruction can read a value from memory, execute an arithmetic operation, and write the result back to memory. Architectures where most instructions can access the memory are called *register-memory* architectures. Instructions in a CISC architecture can take many clock cycles. CISC architectures typically contain many specialized instructions which have to be implemented in hardware. The advantage of CISC is that programs are shorter, reducing the amount of storage needed. The disadvantage is that instructions have a variable length, thus, optimizations for preloading subsequent instructions are much harder.

RISC architectures (such as the *Advanced RISC Machine* (ARM) processors that are found in many mobile devices), have fewer instructions, which are combined to form more specialized functionality. The instructions can typically be executed within one clock cycle, which makes optimizations such as pipelining easier to implement. Several RISC architectures have a *fixed length encoding* of instructions, making preloading of subsequent instructions more predictable. Similarly, one can choose between fixed or variable length of data. RISC architectures are usually *load-store* architectures (in contrast to the register-memory architectures). Only the instructions *load* and *store* can access the memory and transfer data to and from the registers.

The difference between CISC and RISC architectures becomes blurred, as many CISC implementations use internally a RISC layer (e. g., Intel Pentium), and RISC architectures can have specialized extensions, effectively building complex instructions à la CISC. For our purposes, we choose a RISC architecture, namely RISC-V, as it allows us to formalize fewer instructions. In this section, we have introduced the term Instruction Set Architecture, which defines the machine code of a processor. We will use the concept of an instruction set as a basis for our low-level code model in Section 5.1.

2.2 Simulations and Bisimulations

In this section, we introduce the concepts of simulation and bisimulation, which relate behaviors of systems defined as labeled transition system. The goal of simulation and bisimulation relations is to relate systems with similar behavior. We start with the definition of labeled transition systems and then proceed to define simulations and bisimulations. The material presented in this section is based on [Mil89].

A *labeled transition system* (LTS) describes transitions between states. Those transitions are labeled with events.

Definition 2.1: Labeled Transition System

A **labeled transition system** $(A, \rightarrow)$ consists of a set of states A and a set of transitions $\rightarrow \subseteq A \times \Sigma \times A$, which connect the states and are labeled each with an event from the set Σ.

A common way to relate behaviors expressed as LTS are simulations and bisimulations. Informally, an LTS $(A, \rightarrow_1)$ simulates an LTS $(A, \rightarrow_2)$ if it can perform at least the same events from states related by the relation. We first define the notion of simulation.

Definition 2.2: Simulation

A **simulation** $R \subseteq A \times A$ is a relation of states. It relates two LTS $(A, \rightarrow_1)$ and $(A, \rightarrow_2)$ with $\rightarrow_1, \rightarrow_2 \subseteq A \times \Sigma \times A$. The following property is required to hold for a simulation R:

$$\forall (a, b) \in R. \forall\, ev \in \Sigma. \forall\, b' \in A.\, b \xrightarrow{ev}_2 b' \Longrightarrow \exists\, a'.\, a \xrightarrow{ev}_1 a' \wedge (a', b') \in R$$

We can extend the definition of an LTS to include a set of initial states. Considering the sets of initial states A_1^0 and A_2^0, we say $(A, \rightarrow_1, A_1^0)$ simulates $(A, \rightarrow_2, A_2^0)$ if (all) their initial states $a_1^0 \in A_1^0$ and $a_2^0 \in A_2^0$ are in a simulation relation $(a_1^0, a_2^0) \in R$. We furthermore say that in every pair of the simulation, $(A, \rightarrow_1)$ can answer all events of $(A, \rightarrow_2)$.

It is common for LTS to allow transitions to be unlabeled or have a special event for internal or invisible transitions. The special event τ is used for internal transitions that are not externally observable. This allows for a weaker formulation of simulation, where the answering transition can be surrounded by any number of internal transitions and the internal transitions can be answered by no transitions at all. A weaker formulation allowing additional internal steps is desirable, e. g., when a single higher-level action is implemented with multiple lower-level actions of which only one is visible. Let $\xrightarrow{\tau}^*$ denote zero or more transitions labeled with τ.

Definition 2.3: Weak Simulation

A **weak simulation** $R \subseteq A \times A$ is a relation of states for which the following property holds:

$$\begin{aligned}
&\forall (a, b) \in R. \forall\, ev \in \Sigma. \forall\, b' \in A. \\
&\qquad b \xrightarrow{ev}_2 b' \wedge ev \neq \tau \Longrightarrow \exists\, a'.\, a \xrightarrow{\tau}_1^* \xrightarrow{ev}_1 \xrightarrow{\tau}_1^* a' \wedge (a', b') \in R \\
&\qquad \wedge\, b \xrightarrow{ev}_2 b' \wedge ev = \tau \Longrightarrow \exists\, a'.\, a \xrightarrow{\tau}_1^* a' \wedge (a', b') \in R
\end{aligned}$$

Note that the LTS with the empty set of transitions is simulated by anything. The symmetric pendant to (weak) simulation is (weak) bisimulation. A bisimulation requires for *every pair* that the same events are possible.

Definition 2.4: Bisimulation

A **bisimulation** $R \subseteq A \times A$ is a relation of states, for which the following property holds:

$$\begin{aligned}
&\forall (a, b) \in R. \forall\, ev \in \Sigma. \\
&\qquad \forall\, a' \in A.\, a \xrightarrow{ev}_1 a' \Longrightarrow \exists\, b'.\, b \xrightarrow{ev}_2 b' \wedge (a', b') \in R \\
&\qquad \wedge\, \forall\, b' \in A.\, b \xrightarrow{ev}_2 b' \Longrightarrow \exists\, a'.\, a \xrightarrow{ev}_1 a' \wedge (a', b') \in R
\end{aligned}$$

We say $(A, \rightarrow_1, A_1^0)$ is bisimilar to $(A, \rightarrow_2, A_2^0)$ $\big((A, \rightarrow_1, A_1^0) \sim (A, \rightarrow_2, A_2^0)\big)$ if (all) their initial states a_1^0, a_2^0 are in a bisimulation relation $(a_1^0, a_2^0) \in R$.

We also define a weak variant of bisimulation to allow for additional internal transitions.

Definition 2.5: Weak Bisimulation

A **weak bisimulation** $R \subseteq A \times A$ is a relation of states, for which the following property holds:

$$
\begin{aligned}
&\forall (a,b) \in R. \forall\, ev \in \Sigma. \\
&\quad \forall\, a' \in A.\, a \xrightarrow{ev}_1 a' \wedge ev \neq \tau \Longrightarrow \exists\, b'.\, b \xrightarrow{\tau}_2^* \xrightarrow{ev}_2 \xrightarrow{\tau}_2^* b' \wedge (a',b') \in R \\
&\wedge \forall\, a' \in A.\, a \xrightarrow{ev}_1 a' \wedge ev = \tau \Longrightarrow \exists\, b'.\, b \xrightarrow{\tau}_2^* b' \wedge (a',b') \in R \\
&\wedge \forall\, b' \in A.\, b \xrightarrow{ev}_2 b' \wedge ev \neq \tau \Longrightarrow \exists\, a'.\, a \xrightarrow{\tau}_1^* \xrightarrow{ev}_1 \xrightarrow{\tau}_1^* a' \wedge (a',b') \in R \\
&\wedge \forall\, b' \in A.\, b \xrightarrow{ev}_2 b' \wedge ev = \tau \Longrightarrow \exists\, a'.\, a \xrightarrow{\tau}_1^* a' \wedge (a',b') \in R
\end{aligned}
$$

We say $(A, \rightarrow_1, A_1^0)$ is weakly bisimilar to $(A, \rightarrow_2, A_2^0)$ $\big((A, \rightarrow_1, A_1^0) \approx (A, \rightarrow_2, A_2^0)\big)$ if (all) their initial states a_1^0, a_2^0 are in a weak bisimulation relation $(a_1^0, a_2^0) \in R$.

In this section, we have introduced the notions of simulation and bisimulation. They can be used to formally relate systems that are described as LTS. In the next section, we introduce *Communicating Sequential Processes* (CSP) and its notion of refinement. The refinement models of CSP are less discriminating than the bisimulation relation. They enable more differences between abstract and concrete models, while still preserving properties of interest.

2.3 CSP

In this section, we introduce *Communicating Sequential Processes* (CSP). CSP is a process algebra, originally introduced by Hoare in [Hoa78]. It has since been in substantial development and current versions of CSP can be found in [Sch99, Ros10]. It is designed to model concurrent processes that communicate via events. Communication is synchronous and can, thus, be used to synchronize processes or exchange data. The major advantage of CSP are its semantical models which allow for compositional reasoning. In the following, we introduce the syntax of CSP we use in this thesis, as well as its operational and denotational semantics. We also introduce the concept of CSP refinement. The material presented in this section is based on [Sch99].

2.3.1 Syntax and Operational Semantics

In Definition 2.6 we define the grammar for a CSP process P. Processes can be constructed from the basic processes *STOP* and *SKIP* and using operators such as event prefixing ($\rightarrow$), sequential composition (;), external ($\Box$) and internal choice ($\sqcap$), interleaving ($|||$) and alphabetized parallel composition ($\|$). Finally, process variables (X) are used to express (mutual) recursion.

Definition 2.6: Grammar of CSP Processes

Let P be a process, $ev \in \Sigma$ an event from the set of all events Σ, also called the *communication alphabet*. Let α be a communication interface $\alpha \subseteq \Sigma$ and let X express a process variable.

$$P \coloneqq STOP \mid SKIP \mid ev \rightarrow P \mid P \,;\, P \mid P \,\Box\, P \mid P \sqcap P \mid P \,|||\, P \mid P \;{}_{\alpha}\|_{\alpha}\; P \mid X$$

To limit the scope of this thesis, we do not use the operators hiding, renaming, and interface parallel. They can be incorporated into the framework in future work. The basic process *STOP* has no behavior. For all following CSP operators, we give their operational semantics in form of an inference rule after a short explanation. Labeled arrows, e.g. $\xrightarrow{ev}$, denote a semantical step. The basic process *SKIP* can successfully terminate. Successful termination is expressed with the special symbol ✓, which is not part of Σ. To refer to all events including ✓ we write $\Sigma^{\checkmark}$.

$$\frac{}{SKIP \xrightarrow{\checkmark} STOP}$$

The prefix operator $(ev \rightarrow P)$ models the occurrence of an event ev, after which the process behaves as P.

$$\frac{}{(ev \rightarrow P) \xrightarrow{ev} P}$$

Sequential composition $(P_1 \,;\, P_2)$ allows us to chain processes together. The second process is only executed if the first process has successfully terminated. The event ✓ that indicates the successful termination is turned into a τ event, as the combined process is no longer terminated.

$$\frac{P_1 \xrightarrow{ev} P_1'}{(P_1 \,;\, P_2) \xrightarrow{ev} (P_1' \,;\, P_2)} \; ev \neq \checkmark \qquad\qquad \frac{P_1 \xrightarrow{\checkmark} P_1'}{(P_1 \,;\, P_2) \xrightarrow{\tau} P_2}$$

External choice $(P_1 \,\Box\, P_2)$ offers externally (i. e., to other processes or to the abstract environment) the choice whether the process should continue as P_1 or as P_2. Strictly speaking, the external choice operator offers the choice between the first visible event of each of P_1 and P_2. If the first visible event of both P_1 and P_2 is the same, then it is undefined (nondeterministically chosen) which process continues. If one of the processes performs an internal event, then the external choice is not resolved.

$$\frac{P_1 \xrightarrow{ev} P_1'}{\begin{array}{c}(P_1 \,\Box\, P_2) \xrightarrow{ev} P_1' \\ (P_2 \,\Box\, P_1) \xrightarrow{ev} P_1'\end{array}} \qquad\qquad \frac{P_1 \xrightarrow{\tau} P_1'}{\begin{array}{c}(P_1 \,\Box\, P_2) \xrightarrow{\tau} (P_1' \,\Box\, P_2) \\ (P_2 \,\Box\, P_1) \xrightarrow{\tau} (P_2 \,\Box\, P_1')\end{array}}$$

Internal choice ($P_1 \sqcap P_2$) resolves the choice between P_1 and P_2 internally.

$$\frac{}{(P_1 \sqcap P_2) \xrightarrow{\tau} P_1}$$
$$(P_1 \sqcap P_2) \xrightarrow{\tau} P_2$$

Internal choice can be used to model (yet) underspecified behaviors (in terms of alternative behaviors) or unpredictable environments.

In CSP, *channels* are introduced as syntactic sugar on events using the dot-notation: The event $c.v$ is said to communicate the value v over channel c. To describe an input in CSP, $c?x\colon \mathbb{T}$ is used, which denotes an external choice over all events of the form $c.v$ with $v \in \mathbb{T}$.

$$c?x\colon \mathbb{T} \to P_x \coloneqq \mathop{\Box}_{v \in \mathbb{T}} c.v \to P_v$$

The semantics of $c?x\colon \mathbb{T}$ can be formulated also as a derived rule:

$$\frac{}{(c?x\colon \mathbb{T} \to P_x) \xrightarrow{c.v} P_v}\ v \in \mathbb{T}$$

An output is denoted as $c!v$ and simply means $c.v$. To extract a value from an event $c.v$ we define $val(c.v) \coloneqq v$. Note that there is no actual native concept of sending and receiving in CSP, only synchronization. Therefore, CSP needs only a single communication operator $\to$. As we have seen, other operators can be derived. On the other hand, as ? and ! are only syntactic sugar, it is perfectly valid to synchronize two "senders" or two "receivers".

Example 2.1: Synchronizing Two "Senders"

Consider the two "sending" processes $P = c!5 \to P'$ and $Q = c!5 \to Q'$. If those two processed are synchronized over the channel c, then they can agree to synchronize on the event $c.5$. So in CSP "sender" and "receiver" denote only the amount of events that are offered. We take up this concept later in Section 6.2 when we construct a sender that can only synchronize with receivers (and vice versa) within the CSP communication mechanism.

To combine processes concurrently, CSP provides multiple operators, of which we consider interleaving and alphabetized parallel. Interleaving ($P_1 \mathbin{|||} P_2$) does not allow any communication between the processes P_1 and P_2 and allows both processes to take turns in performing any number of events.

$$\frac{P_1 \xrightarrow{ev} P_1'}{(P_1 \mathbin{|||} P_2) \xrightarrow{ev} (P_1' \mathbin{|||} P_2)}\ ev \neq \checkmark$$
$$(P_2 \mathbin{|||} P_1) \xrightarrow{ev} (P_2 \mathbin{|||} P_1')$$

The interleaving process can only terminate if both processes terminate.

$$\frac{P_1 \xrightarrow{\checkmark} P_1' \qquad P_2 \xrightarrow{\checkmark} P_2'}{(P_1 \mathbin{|||} P_2) \xrightarrow{\checkmark} (P_1' \mathbin{|||} P_2')}$$

Alphabetized parallel $(P_1 \ {}_{\alpha_1}\|_{\alpha_2} \ P_2)$ allows each process P_i to perform events from their communication interface (or alphabet) α_i. Events in the intersection of both communication interfaces $(\alpha_1 \cap \alpha_2)$ can only be performed synchronously, i. e., by both processes at the same time. The processes perform the remaining events from their respective communication interfaces in an interleaving fashion.

$$\frac{P_1 \xrightarrow{ev} P_1' \qquad P_2 \xrightarrow{ev} P_2'}{(P_1 \ {}_{\alpha_1}\|_{\alpha_2} \ P_2) \xrightarrow{ev} (P_1' \ {}_{\alpha_1}\|_{\alpha_2} \ P_2')} \ ev \in (\alpha_1^{\checkmark} \cap \alpha_2^{\checkmark})$$

$$\frac{P_1 \xrightarrow{ev} P_1'}{(P_1 \ {}_{\alpha_1}\|_{\alpha_2} \ P_2) \xrightarrow{ev} (P_1' \ {}_{\alpha_1}\|_{\alpha_2} \ P_2)} \ ev \in (\alpha_1 \cup \{\tau\}) \setminus \alpha_2$$

$$(P_2 \ {}_{\alpha_2}\|_{\alpha_1} \ P_1) \xrightarrow{ev} (P_2 \ {}_{\alpha_2}\|_{\alpha_1} \ P_1')$$

A process that is prevented from communicating an event from its communication interface is called *blocked*. If it is blocked eternally, it is said to be in a *deadlock*. Common scenarios for a deadlock are that the other process in an alphabetized parallel has stopped or is blocked itself.

CSP allows for building (mutually) recursive processes using process variables. The process $P = a \rightarrow Q$ communicates the event a and then behaves as the process Q.

$$\frac{P \xrightarrow{ev} P'}{X \xrightarrow{ev} P'} \ X = P$$

To illustrate the definition of a CSP process, we give an example.

Example 2.2: Counter Process

Using the concepts defined above, we can construct a process C_0 that holds a number that can be incremented or decremented via the events *up* and *down*.

$$C_0 = up \rightarrow C_1$$
$$C_n = down \rightarrow C_{n-1} \ \Box \ up \rightarrow C_{n+1} \qquad n > 0$$

Here, C_i for $n \geq 0$ are process variables to which processes are assigned in a mutually recursive fashion. C_0 offers the event *up* and then behaves as C_1. C_n for $n > 0$ offers both the events *up* and *down* and then behaves as C_{n-1} if *down* was communicated and behaves as C_{n+1} if *up* was communicated. Note that we defined an infinite number of process variables using parametrized process names, also called a process scheme. Using parameters allows CSP processes to store information without having an explicit state.

Note that there is a special interpretation of events and the abstract environment: In CSP, we can imagine an abstract *environment* E that is willing to synchronize on any event. The environment can be thought of being in parallel to the topmost process. For a process P with events in α this may look like this:

$$P \ {}_{\alpha}\|_{\Sigma} \ E$$

This process can engage exactly in the same events as just the process P. Visible events can be divided into two categories: observational events and communication events (similar to external and waiting states in [PS92]). Communication events are meant to synchronize two or more components (and possibly also exchange data) and observational events only indicate to an observer what is happening. Technically, they have the same semantics (they synchronize with other components), but observational events only synchronize with the environment and, in the complete system, communication events always synchronize with other components. For the construction of CUC in Chapter 5, we focus on communication events.

Having introduced the syntax and operational semantics of CSP, we proceed to its denotational semantics. The advantage of the denotational semantics is that they are compositional and allow for compositional reasoning about properties of CSP processes. CSP has three main denotational models (traces, stable failures and failures-divergences), which increasingly discriminate more behaviors. All three denotational semantics are accompanied by their notion of refinement ($\sqsubseteq$), which defines behavioral inclusion: one process behaves within the bounds defined by another process. The important aspect of the CSP refinements is that they are compositional with respect to contexts[1] ($\mathcal{C}$), i. e., if only a part of the system is refined, the whole system is also in a refinement relation:

$$\forall \mathcal{C}.\, A \sqsubseteq B \Longrightarrow \mathcal{C}(A) \sqsubseteq \mathcal{C}(B)$$

As parallel composition can also be part of the context $\mathcal{C}$, this allows for modular verification of concurrent systems. The refinement relations of CSP are similar in intent to the simulation relation.

2.3.2 Traces Semantics

The least discriminating denotational semantics of CSP is the traces semantics. A trace is a sequence of visible events.

[1] A context is a "process with a hole" and is formalized as a function from CSP process to CSP process.

Definition 2.7: Traces

A **trace** tr is a sequence of events. We write

$$tr \in \Sigma^*$$

Similar, for traces that can "successfully terminate", we write

$$tr \in (\Sigma^*)^{\checkmark}$$

To obtain the events that occur within a given trace tr, we write $set(tr)$.

The traces semantics can be characterized operationally.

Definition 2.8: Operational Characterization of the Traces Semantics of CSP

With the following expression, we denote that there is an execution of process P with the observed trace tr after which the process behaves as P'.

$$P \overset{tr}{\Longrightarrow} P'$$

The traces semantics $\mathcal{T}$ of a CSP process P captures all traces tr that can be observed when the process P is executed.

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

Definition 2.9 shows the denotational traces semantics of CSP. For the basic processes *STOP* and *SKIP* the possible traces are directly defined (only the empty trace for *STOP* and additionally also the trace containing only $\checkmark$ for *SKIP*). Whenever the set of traces is defined, the empty trace is included, as every process can be observed to do nothing. The traces semantics for the other operators are defined based on the traces semantics of the component processes. The traces of the prefix operator ($\rightarrow$) include every trace of the following process P prefixed with the event ev. The traces of the sequential composition are the traces of the first process without $\checkmark$. If the first process successfully terminates, then the traces of the second process may be appended. Both internal ($\sqcap$) and external choice ($\Box$) combine the traces of the components by set union. The equivalent definition of both choices implies that the traces semantics cannot distinguish between both choices. The traces of the interleaving operator ($|||$) are defined via the predicate **interleaves**. Informally speaking, a trace is an interleaving of two given traces, if it is a piece-wise combination of the two given traces. The traces of the alphabetized parallel operator ($_{\alpha_1}\|_{\alpha_2}$) are related with the traces of the components via projections ($\upharpoonright$). For each component P_i, when only considering the events from the respective communication alphabet ($\upharpoonright\ \alpha_i$), the combined trace is required to be part of the traces of the component P_i. Additionally, all events of the combined trace must be events of one of the components' communication interface α_i or the termination indicator $\checkmark$. A process that is defined using process variables can be recursive. If it is not recursive, the process variable can be replaced by the designated process and the traces can

Definition 2.9: Traces Semantics of CSP

$$
\begin{aligned}
\mathcal{T}(STOP) &= \{\langle\rangle\} \\
\mathcal{T}(SKIP) &= \{\langle\rangle, \langle\checkmark\rangle\} \\
\mathcal{T}(ev \to P) &= \{\langle\rangle\} \cup \{\langle ev\rangle^\frown tr \mid tr \in \mathcal{T}(P)\} \\
\mathcal{T}(P_1 \,;\, P_2) &= \{tr \mid tr \in \mathcal{T}(P_1) \wedge \checkmark \notin set(tr)\} \cup \\
&\qquad \{tr_1{}^\frown tr_2 \mid tr_1{}^\frown\checkmark \in \mathcal{T}(P_1) \wedge tr_2 \in \mathcal{T}(P_2)\} \\
\mathcal{T}(P_1 \,\Box\, P_2) &= \mathcal{T}(P_1) \cup \mathcal{T}(P_2) \\
\mathcal{T}(P_1 \sqcap P_2) &= \mathcal{T}(P_1) \cup \mathcal{T}(P_2) \\
\mathcal{T}(P_1 \,|||\, P_2) &= \{tr \mid \exists\, tr_1\ tr_2.\, tr_1 \in \mathcal{T}(P_1) \wedge \\
&\qquad\qquad tr_2 \in \mathcal{T}(P_2) \wedge \\
&\qquad\qquad tr \textbf{ interleaves } tr_1\ tr_2\} \\
\mathcal{T}(P_1 \,{}_{\alpha_1}\|_{\alpha_2}\, P_2) &= \{tr \mid tr \upharpoonright \alpha_1 \in \mathcal{T}(P_1) \wedge \\
&\qquad\qquad tr \upharpoonright \alpha_2 \in \mathcal{T}(P_2) \wedge \\
&\qquad\qquad set(tr) \subseteq (\alpha_1 \cup \alpha_2)^\checkmark\} \\
\mathcal{T}(X = F(X)) &= \bigcup_n \mathcal{T}(F^n(STOP))
\end{aligned}
$$

where

$$
\begin{aligned}
\langle\rangle \textbf{ interleaves } tr_1\ tr_2 &\Leftrightarrow tr_1 = tr_2 = \langle\rangle \\
\langle\checkmark\rangle \textbf{ interleaves } tr_1\ tr_2 &\Leftrightarrow tr_1 = tr_2 = \langle\checkmark\rangle \\
\langle ev\rangle^\frown tr \neq \langle\checkmark\rangle &\Longrightarrow \\
\langle ev\rangle^\frown tr \textbf{ interleaves } tr_1\ tr_2 &\Leftrightarrow head(tr_1) = ev \wedge tr \textbf{ interleaves } tail(tr_1)\ tr_2 \wedge \\
&\qquad head(tr_2) = ev \wedge tr \textbf{ interleaves } tr_1\ tail(tr_2)
\end{aligned}
$$

be determined. If the process is recursive, then we need a fixpoint. In this case, we can consider the union over all traces, which are defined by repeatedly applying the process function F to the basic process $STOP$.

Example 2.3: Traces of a Recursive CSP Process

Consider the process

$$P = a \to P$$

We can formulate the same process with a function F from process to process

$$
\begin{aligned}
F(X) &\coloneqq a \to X \\
P &= F(P)
\end{aligned}
$$

Example 2.3: Traces of a Recursive CSP Process

Let us now consider the sequence of traces of multiple applications of F to $STOP$:

$$\begin{aligned}
\mathcal{T}\big(F^0(STOP)\big) &= \mathcal{T}(STOP) && = \{\langle\rangle\} \\
\mathcal{T}\big(F^1(STOP)\big) &= \mathcal{T}(a \to STOP) && = \{\langle\rangle, \langle a\rangle\} \\
\mathcal{T}\big(F^2(STOP)\big) &= \mathcal{T}(a \to a \to STOP) && = \{\langle\rangle, \langle a\rangle, \langle a, a\rangle\} \\
&\vdots && \vdots \\
\bigcup_n \mathcal{T}\big(F^n(STOP)\big) &= \mathcal{T}(a \to a \to \ldots \to STOP) && = \{\langle a\rangle^n \mid n \geq 0\}
\end{aligned}$$

The traces semantics and its operational characterization coincide, so we always use $\mathcal{T}(P)$ to denote the traces of the process P. It is now possible to compare two processes with regard to their traces semantics.

Definition 2.10: Traces Refinement

$$P \sqsubseteq_{\mathcal{T}} Q \coloneqq \mathcal{T}(Q) \subseteq \mathcal{T}(P)$$

Note the inverse positioning of P and Q. P is refined by Q if the traces of Q are a subset of the traces of P.

A safety property describes all behaviors that are allowed to happen (and thereby excludes the behaviors that are not allowed). A safety property can describe, e. g., that only allowed values are communicated. We can describe safety properties with processes.

Example 2.4: Safety Property as Process

Let $bad \in \Sigma$ be an undesirable event. We can formulate the safety property S that bad should not be communicated as follows.

$$S = \bigsqcap_{ev \in \Sigma \setminus \{bad\}} ev \to S$$

We use the indexed internal choice $\bigsqcap_{ev \in EV} ev \to P_{ev}$, which denotes an internal choice with the option to communicate ev and then behave as P_{ev} for each $ev \in EV$. In the case of our safety property S, we allow any event but bad and then behave again as S, describing any sequence of events that do not contain the event bad.

As the traces refinement is a subset relation of behaviors, it can be used to verify that a process P has a certain *safety property* S.

$$S \sqsubseteq_{\mathcal{T}} P$$

Furthermore, as the refinement relation is transitive, if we refine the process P to P', than S

holds also for P'.

$$S \sqsubseteq_{\mathcal{T}} P \wedge P \sqsubseteq_{\mathcal{T}} P' \Longrightarrow S \sqsubseteq_{\mathcal{T}} P'$$

Thus, the refinement relation allows both to verify properties of processes as well as the preservation of properties from one process to another.

The traces refinement is similar to the simulation refinement in that it (only) sets an upper bound to the behavior. In contrast to the simulation relation, the traces refinement cannot discriminate the branching behavior of two processes. A CSP specific example is that the traces semantics for both internal choice and external choice are the same, thus they are trace equivalent.

Definition 2.11: Trace Equivalence

$$P \equiv_{\mathcal{T}} Q := P \sqsubseteq_{\mathcal{T}} Q \wedge Q \sqsubseteq_{\mathcal{T}} P$$

Example 2.5: External and Internal Choice Cannot be Differentiated by the Traces Semantics

$$(a \rightarrow P \,\Box\, b \rightarrow Q) \equiv_{\mathcal{T}} (a \rightarrow P \sqcap b \rightarrow Q)$$

Note that although they have the same traces semantics, they offer different events. The process with the external choice offers both the events a and b *at the same time*. The process with the internal choice offers either the event a or the event b depending on the resolution of the internal choice.

The traces semantics considers only events that have been communicated by a process. To consider liveness properties, we also need to consider events that are offered by a process. The stable failures semantics, which we cover next, considers the offered events.

2.3.3 Stable Failures Semantics

The stable failures semantics covers, in addition to the traces taken, also the information which events are offered. This allows for the verification of liveness properties, i. e., properties that talk about the progress of processes. The stable failures semantics records at every stable state all events that can be refused.

Definition 2.12: Stable Process

A process is **stable** if no internal transitions are possible. Formally

$$P\downarrow := \nexists P'.\, P \xrightarrow{\tau} P'$$

In stable processes, the sets of events that can be refused are considered.

Definition 2.13: Refusal Set

$$P \text{ ref } X \coloneqq \forall\, ev \in X.\, \nexists\, P'.\, P \xrightarrow{ev} P'$$

Note that P ref $\emptyset$ is always true. Furthermore, any subset of a refusal set is also a refusal set.

The information about the events that are refused and the information about the trace that was executed to get to the current state of the process are combined. To ensure that the events are offered stably, i. e., the process cannot make an internal transition which changes the offered events, only stable processes are considered. We first show the operational characterization of the stable failures and then give its denotational semantics.

Definition 2.14: Operational Characterization of the Stable Failures Semantics of CSP

A **stable failure** of a CSP process P is a pair of a trace tr and a refusal set X. It denotes that there is a stable process P' which can be reached from the initial state P via the trace tr and which refuses X.

$$(tr, X) \in \mathcal{SF}(P) \coloneqq \exists\, P'.\, P' \overset{tr}{\Longrightarrow} P' \wedge P'\!\downarrow \wedge\, P' \text{ ref } X$$

Definition 2.15 shows the denotational stable failures semantics of CSP. The stable failures for the basic processes are directly defined, the stable failures for the other operators are defined based on the stable failures semantics of the component processes. The basic process $STOP$ can be observed to do nothing (empty trace) and afterwards it can refuse anything. We define another basic process DIV, which is the process that diverges, i. e., can engage in infinitely many internal transitions. As it is never stable, its stable failures semantics is the empty set. The basic process $SKIP$ can refuse anything but successful termination after the empty traces, and after the successful termination it can refuse anything. The stable failures of the prefix operator ($\rightarrow$) include every stable failure of the following process P, where the trace is prefixed with the event ev. Additionally, after the empty trace anything but the event ev can be refused. The stable failures of the sequential composition are the stable failures of the first process that can refuse to successfully terminate. If the first process successfully terminates, then the traces of the second process may be appended and the respective refusal sets are inherited. In the stable failures semantics, internal ($\sqcap$) and external choice ($\Box$) have different semantics. They differ in the refusal set that occur after the empty trace. The stable failures semantics of the external choice include only those refusal sets after the empty trace, which occur in *both* of the stable failures of the components. In contrast, the stable failures of the internal choice includes all refusal sets that occur in *any* of the stable failures of the components. After the first visible event, both choices combine the stable failures of the components. The stable failures of the interleaving operator ($|||$) is defined via the predicate **interleaves** as in the traces semantics. Additionally, if any component refuses successful termination, the interleaving process can refuse it, too. The stable failures of the alphabetized parallel operator ($_{\alpha_1}\|_{\alpha_2}$) are related with the traces of the components via projections ($\upharpoonright$) as in the trace semantics. Additionally, the refusal sets of the combined process need to match the combination of existing refusal sets of the components, when restricted to the respective communication interfaces. The stable failures semantics for

Definition 2.15: Stable Failures Semantics of CSP

$$
\begin{aligned}
\mathcal{SF}(STOP) &= \{(\langle\rangle, X) \mid X \subseteq \Sigma^{\checkmark}\} \\
\mathcal{SF}(DIV) &= \{\} \\
\mathcal{SF}(SKIP) &= \{(\langle\rangle, X) \mid \checkmark \notin X\} \cup \{(\langle\checkmark\rangle, X) \mid X \subseteq \Sigma^{\checkmark}\} \\
\mathcal{SF}(ev \to P) &= \{(\langle\rangle, X) \mid ev \notin X\} \cup \{(\langle ev\rangle \frown tr, X) \mid (tr, X) \in \mathcal{SF}(P)\} \\
\mathcal{SF}(P_1 \,;\, P_2) &= \{(tr, X) \mid (tr, X \cup \{\checkmark\}) \in \mathcal{SF}(P_1)\} \cup \\
&\quad \{(tr_1 \frown tr_2, X) \mid tr_1 \frown \checkmark \in \mathcal{T}(P_1) \wedge (tr_2, X) \in \mathcal{SF}(P_2)\} \\
\mathcal{SF}(P_1 \,\Box\, P_2) &= \{(\langle\rangle, X) \mid (\langle\rangle, X) \in \mathcal{SF}(P_1) \cap \mathcal{SF}(P_2)\} \cup \\
&\quad \{(tr, X) \mid tr \neq \langle\rangle \wedge (tr, X) \in \mathcal{SF}(P_1) \cup \mathcal{SF}(P_2)\} \\
\mathcal{SF}(P_1 \sqcap P_2) &= \mathcal{SF}(P_1) \cup \mathcal{SF}(P_2) \\
\mathcal{SF}(P_1 \,|||\, P_2) &= \{(tr, X_1 \cup X_2) \mid \exists\, tr_1\ tr_2.\ X_1 \upharpoonright \Sigma = X_2 \upharpoonright \Sigma \wedge \\
&\qquad (tr_1, X_1) \in \mathcal{SF}(P_1) \wedge \\
&\qquad (tr_2, X_2) \in \mathcal{SF}(P_2) \wedge \\
&\qquad tr \textbf{ interleaves } tr_1\ tr_2\} \\
\mathcal{SF}(P_1 \,{}_{\alpha_1}\|_{\alpha_2}\, P_2) &= \{(tr, X) \mid \exists\, X_1\ X_2. \\
&\qquad X \cap (\alpha_1 \cup \alpha_2) = (X_1 \cap \alpha_1) \cup (X_2 \cap \alpha_2) \wedge \\
&\qquad (tr \upharpoonright \alpha_1, X_1) \in \mathcal{SF}(P_1) \wedge \\
&\qquad (tr \upharpoonright \alpha_2, X_2) \in \mathcal{SF}(P_2) \wedge \\
&\qquad set(tr) \subseteq (\alpha_1 \cup \alpha_2)^{\checkmark}\} \\
\mathcal{SF}(X = F(X)) &= \bigcup_n \mathcal{SF}(F^n(DIV))
\end{aligned}
$$

recursive processes are defined as the union over all stable failures, by repeated application of the process function F. In contrast to the traces semantics, the initial process used for the fixpoint of the stable failures semantics is DIV. DIV diverges immediately and therefore has no stable failures at all.

Example 2.6: Stable Failures of a Recursive CSP Process

Consider the process

$$P = a \to P$$

We can formulate the same process with a function F from process to process

$$
\begin{aligned}
F(X) &\coloneqq a \to X \\
P &= F(P)
\end{aligned}
$$

Example 2.6: Stable Failures of a Recursive CSP Process

Let us now consider the sequence of stable failures of multiple applications of F to DIV:

$$\begin{aligned}
\mathcal{SF}\big(F^0(DIV)\big) &= \mathcal{SF}(DIV) && = \{\} \\
\mathcal{SF}\big(F^1(DIV)\big) &= \mathcal{SF}(a \to DIV) && = \{(\langle\rangle, X) \mid a \notin X\} \\
\mathcal{SF}\big(F^2(DIV)\big) &= \mathcal{SF}(a \to a \to DIV) && = \{(\langle\rangle, X) \mid a \notin X\} \cup \{(\langle a\rangle, X) \mid a \notin X\} \\
&\vdots && \vdots \\
\bigcup_n \mathcal{SF}\big(F^n(DIV)\big) &= \mathcal{SF}(a \to a \to \ldots \to DIV) && = \{(\langle a\rangle^n, X) \mid n \geq 0 \wedge a \notin X\}
\end{aligned}$$

In CSP, there are two kinds of stable processes (i. e., where no internal transitions are possible): Processes ready to communicate and $STOP$. Let us call failures resulting from the former *communication* failures and from the latter *terminal* failures. Considering the sequential composition $P_1 \,;\, P_2$, the main difference between the communication failures and the terminal failures is that the terminal failures of P_1 are removed by the sequential composition operator.

The stable failures semantics and its operational characterization coincide, so we always use $\mathcal{SF}(P)$ to denote the stable failures of the process P. It is now possible to compare two processes with respect to their stable failures semantics.

Definition 2.16: Stable Failures Refinement

$$P \sqsubseteq_{\mathcal{SF}} Q \coloneqq \mathcal{T}(Q) \subseteq \mathcal{T}(P) \wedge \mathcal{SF}(Q) \subseteq \mathcal{SF}(P)$$

Note the inverse positioning of P and Q. P is refined by Q in the stable failures model if *a)* the traces of Q are a subset of the traces of P (as in the traces refinement) and *b)* the stable failures Q are a subset of the stable failures of P.

The part $\mathcal{T}(Q) \subseteq \mathcal{T}(P)$ (which is equivalent to $P \sqsubseteq_{\mathcal{T}} Q$) ensures that the traces also are a subset for the reachable processes from Q that are not captured by the stable failures semantics, i. e., reachable unstable processes. The part $\mathcal{SF}(Q) \subseteq \mathcal{SF}(P)$ expresses that all reachable stable processes Q' from Q have a corresponding process P' reachable from P, and that the events that can be refused in Q' can also be refused in P'.

Example 2.7: Resolution of Non-Determinism

In the stable failures model, external choice and internal choice are no longer equivalent, as the internal choice can refuse more than the external choice.

$$P \,\Box\, Q \not\sqsubseteq_{\mathcal{SF}} P \sqcap Q$$

Example 2.7: Resolution of Non-Determinism

The other direction is still valid.

$$P \sqcap Q \sqsubseteq_{\mathcal{SF}} P \,\Box\, Q$$

This is also the main idea of CSP refinement: to resolve non-determinism. Following this idea, it is also valid to refine only one branch of the internal choice.

$$P \sqcap Q \sqsubseteq_{\mathcal{T}/\mathcal{SF}} P$$

$$P \sqcap Q \sqsubseteq_{\mathcal{T}/\mathcal{SF}} Q$$

A liveness property (in the sense of CSP) is a property that ensures the availability of events that can be stably offered. A liveness property can describe, e. g., deadlock freedom (by stating that there must be always an event available). Again, we can describe a liveness property as a process.

Example 2.8: Liveness Property as a Process

Let *emergency_brake* be an event that needs to be offered at all times. We can formulate the liveness property L that *emergency_brake* should be always available as follows.

$$L = \left(\bigsqcap_{ev \in \Sigma \setminus \{emergency\_brake\}} ev \rightarrow L \right) \Box\, emergency\_brake \rightarrow STOP$$

The indexed internal choice allows for any combination of events other than *emergency_brake*. The external choice ensures that the event *emergency_brake* cannot be refused by L.

This time we use the stable failures refinement to verify that a process P has a fulfills a certain liveness property L:

$$L \sqsubseteq_{\mathcal{SF}} P$$

The stable failures refinement (and the failures-divergences refinement from the next subsection) are also transitive.

Compared to weak simulations and weak bisimulations, the stable failures refinement ensures availability of events similar to a weak bisimulation (and is as such roughly more discriminative than a weak simulation) but does not require the whole branching structure to be equivalent. In particular, if in a process two internal transitions are available (e. g., occurring due to an internal choice), all but one branch can be removed, and it is still a stable failures refinement. This allows for the reduction of non-determinism.

Other definitions of liveness also include the notion of *livelock freedom* or *divergence freedom*, which is covered in CSP by the failures-divergences semantics, which we briefly describe in the next subsection.

2.3.4 Failures-Divergences Semantics

In this subsection, we describe the failures-divergences semantics. The failures-divergences semantics records in addition to the traces and failures also the *divergences*, i. e., traces that lead to processes where infinite internal loops are possible. These infinite loops have the effect that all visible events can be refused.

As the failures-divergences semantics distinguishes between a process that has divergences and a process that does not have divergences, the failures-divergences refinement ($\sqsubseteq_{FD}$) can be used to verify that a process is divergence free or livelock free. More specifically, $P \sqsubseteq_{FD} Q$ implies that Q has at most the divergences that P has. In this thesis, we consider only divergence free CSP processes unless otherwise noted.

That concludes the introduction of the semantics of CSP. In the next subsection, we introduce wellformedness properties that are required for semantics of CSP.

2.3.5 Properties

In this subsection, we briefly discuss the wellformedness properties that are required for the CSP semantics, in order for the refinement notion of CSP to preserve safety and liveness properties. The CSP semantics are defined in way that the properties hold true. So they can be understood as theorems that hold for the semantics in Definitions 2.9 and 2.15. We refer to [Sch99, Ros10] for a comprehensive description of the wellformedness properties.

The first two properties ensure that the empty trace (Property T1) and traces of partial executions (Property T2) are always included in the traces semantics. Let P be a process, tr and tr' traces.

Property T1

$$\langle\rangle \in \mathcal{T}(P)$$

The empty trace must always be contained. Any process can be observed to do nothing.

Property T2

$$\forall\, tr\ tr'.\ tr' \leq tr \wedge tr \in \mathcal{T}(P) \Longrightarrow tr' \in \mathcal{T}(P)$$

(where $\leq$ is the prefix relation: $tr' \leq tr := \exists\, tr''.\ tr' \frown tr'' = tr$.)

$\mathcal{T}$ is prefix closed. All partial behaviors can be observed.

The properties for the stable failures semantics are concerned with the connection to the traces semantics (Property SF1), ensure that there are "enough" refusal sets so that resolution of non-determinism can be expressed with a subset relation (Properties SF2 and SF3), and that terminated processes can refuse anything (Property SF4).

Property SF1

$$(tr, X) \in \mathcal{SF}(P) \Longrightarrow tr \in \mathcal{T}(P)$$

All traces are included in the traces semantics. This property ensures that the stable failures semantics "acts" within the boundaries of the traces semantics. However, this does not cover all traces from the traces semantics, which is why the trace refinement is a requirement for the stable failures refinement.

Property SF2

$$(tr, X) \in \mathcal{SF}(P) \wedge X' \subseteq X \Longrightarrow (tr, X') \in \mathcal{SF}(P)$$

Refusal sets are subset closed. This property ensures, e. g., that internal choice can be refined by external choice.

Property SF3

$$(tr, X) \in \mathcal{SF}(P) \wedge \forall\, a \in X'.\ tr^\frown\langle a\rangle \notin \mathcal{T}(P) \Longrightarrow (tr, X \cup X') \in \mathcal{SF}(P)$$

All events that can be refused occur in a refusal set. More specifically, the refusal sets can be augmented with refused events. This is the crucial property ensuring that the stable failures are "enough" refusals to show liveness properties, because it ensures that all events that can be refused are contained in the stable failures semantics.

Property SF4

$$tr^\frown\langle\checkmark\rangle \in \mathcal{T} \Longrightarrow (tr^\frown\langle\checkmark\rangle, X) \in \mathcal{SF}$$

Terminal failures are stable.

As the refinement notions of CSP are based on set inclusion, we need to ensure that the semantics of the compared processes are indeed useful descriptions of a system. If we only would use the subset relation of the refinement, and not have the properties defined above of the process semantics, than we would not have, e. g., that the empty trace or traces of partial executions are part of the semantics, or that we can refuse only a subset of the refusals, which allows for the refinement of internal choice to external choice.

In this subsection, we have presented the wellformedness properties for the traces and the stable failures semantics of CSP. When designing the traces and stable failures semantics for CUC in Chapter 5, we ensure that they meet the wellformedness properties as well. In the next subsection, we briefly introduce FDR4, an automatic refinement checker for CSP.

2.3.6 Failures-Divergences Refinement Checker (FDR4)

In this subsection, we briefly introduce the *Failures-Divergences Refinement Checker* (FDR4). FDR4 takes as inputs programs specified in CSPm, a machine readable dialect of CSP that is enhanced with functional programming constructs. FDR4 is basically a model checker. To show a refinement, e. g., $P \sqsubseteq Q$, it constructs a finite state machine for each of the processes. The state machines are then used to show the refinement in a game fashion: The state machine for P has to answer all transitions of the state machine of Q. The state machines enable the refinement-checking of non-terminating processes when state matching is possible. Due to the nature of model checking, it can only check finite state machines.

We mention FDR4 because it allows for the automatic verification of refinement relations between CSP processes and is an important part of the CSP ecosystem. We do not use FDR4 in this thesis, as we are interested in refinement relations between CSP processes and low-level code. In a model-driven design process, the application of FDR4 would take place before the application of our framework.

2.4 Isabelle/HOL

In this section, we briefly introduce Isabelle/HOL [NPW02]. Isabelle is a theorem prover with which mathematical theories can be formalized to formally reason about them. The proofs in Isabelle are machine checked, i. e., all steps are checked to be correct with respect to the used logic and theories. It is built around a small kernel formalizing constructive logic. With that small kernel other logics can be formalized using datatypes, functions, definitions, axioms and theorems. There are many logics formalized in Isabelle. A well-developed logic with a lot of submitted theories is *higher order logic*, hence the name Isabelle/HOL. Isabelle/HOL not only allows for the formalization and mechanical proof checking. Its tool suite also provides search algorithms to help finding proofs and proof procedures (tactics and tacticals) to (semi-) automate (parts of) proofs. We have formalized the contents of Chapter 5 in Isabelle/HOL, e. g., our syntax of CUC and its operational and denotational semantics, as well as our Hoare calculus, theorems, lemmas, and their proofs. For the formalization of CSP within Isabelle/HOL, we use the CSP-Prover library [IR05].

2.5 Summary

In this chapter, we have introduced the background for this thesis. We have covered Instruction Set Architectures to explain our understanding of low-level code. We have defined simulations and bisimulations as traditional means to relate system behaviors and as a basis for the handshake refinement we define in Chapter 6.1. We have introduced CSP, its syntax, its semantics, and its notion of refinement. A crucial property of the CSP refinement is that it is compositional. As such, it allows us to verify single components individually and compose the results. We define semantics similar to CSP's for our language CUC in Chapter 5 and ensure that they fulfill the wellformedness properties to extend the refinement notion from CSP-only to also cover CUC. Finally, we have briefly introduced the theorem prover Isabelle/HOL, which we have used to formalize the results from Chapter 5, i. e., the theory

of CUC. The formalization in Isabelle/HOL ensures the rigorousness of our framework even in the case of user provided proofs, which are required in the first part of our framework. Furthermore, the formalization in Isabelle/HOL allows for the (semi-) automation of proofs to help the user. In the next chapter, we discuss related work.

Chapter 3

Related Work

In this chapter, we present the related work of this thesis in four parts. In 3.1, we consider approaches that model low-level languages for formal verification. This entails approaches that define syntax and semantics for low-level languages and calculi for verification of properties on programs in that language. As we aim at showing properties for non-terminating systems, we are particularly focusing on denotational semantics, which are suitable to describe infinite behaviors and are compositional. In Section 3.2, we consider compositional proof calculi for concurrent systems. Our aim is to define a proof calculus for CUC to relate it to CSP. In Section 3.3, we consider fundamental work in the area of relating and comparing synchronous and asynchronous languages, as our aim is to relate a specification in the synchronous language CSP to an implementation in low-level code, which is asynchronous. We consider implementation relations that preserve liveness properties in 3.4 as our aim is to implement the synchronous communication of CSP while preserving liveness properties. In 3.5, we look at other approaches to implement synchronous communication concepts.

3.1 Formalizations of Low-Level Languages Geared Towards Compositional Verification

In this section, we give an overview over approaches concerned with the formalization of low-level code for compositional verification.

Saabas and Uustalu [SU05] present a compositional big step semantics of an unstructured language. To this end, a generic structuring mechanism for the code is presented, which makes the semantics (sequentially) compositional and also allows for a compositional proof calculus. Although they formally relate a high-level language (WHILE) and a low-level language, communication is not considered and behaviors of non-terminating systems are also not captured. We use their structuring mechanism for CUC.

In [BG11], Bartels and Glesner formalize a subset of the compiler intermediate-representation LLVM [LA04] to relate low-level code and TimedCSP (an extension of CSP with a notion of time). They relate the two languages with a weak timed bisimulation. A timed labeled transition system (TLTS) is constructed for the LLVM program and annotated by hand with labels matching the labels in the TLTS of the TimedCSP process. The annota-

tions of the TLTS for the LLVM program assert properties about the program state and can be verified with the Hoare calculus they provide, which is an adaptation of the work of Saabas and Uustalu [SU05]. They consider communication in LLVM as synchronous, therefore the receiving process is blocking until the data to be communicated is written to a register. Building on this assumption, they relay the interplay of the communication to TimedCSP, without having a concurrent semantics and modeling the synchronization mechanism needed for this. Our goal is to verify the communication and synchronization mechanisms themselves. Thus, their approach is too abstract to perform this. We use the idea to relay the communication to CSP (to inherit its concurrent compositionality), but we prove the correctness of this approach using our concurrent semantics and our verification of the synchronization protocol.

Tews [Tew04] developed a compositional semantics for a C-like language with `goto` statements, which is used to verify Duff's device. Duff's device is a programming technique invented to copy large amounts of data, dealing in an interesting way with the leftovers of loop unrolling: A jump into the body of a while loop expressed with a not nested but intermingled use of a switch construct and a while loop. However, this approach does not model communication.

A different approach, but also including a formalization of a low-level language for verification is the certified compiler CompCert [Ler09]. The verification of a compiler is scalable in the sense that the transformation is verified once, yielding the verification of all transformed programs. In the CompCert project, executable low-level code (Assembly) is related to high-level code (Clight). While Clight can express low-level communication, it does not have an abstract concept of communication. Therefore, they cannot consider liveness properties in the sense of stably offering communication events.

In contrast to our approach, the presented approaches either do not consider liveness or no communication, or neglect the low-level implementation of communication and synchronization.

3.2 Compositional Proof Calculi for Concurrent Systems

In this section, we give an overview over compositional proof calculi used for concurrent systems.

A denotational semantics and a Hoare-style proof calculus for a *high-level* language with communication is defined by Zwiers [Zwi89]. The semantics deals with traces and ready sets, which are similar in intention to refusals, i. e., they allow for the formulation of liveness properties. As low-level code is not considered, the semantics is not applicable. The important part of such semantics and the accompanying proof calculus is the looping construct, which is different in low-level languages, where sequential composition and looping construct have to be combined because jumps into and out of code are possible almost everywhere. The key for us to achieve (sequential) compositionality when reasoning about low-level code is the structuring mechanism from [SU05].

Session types, for example used in [LNTY17] to reason about safety and liveness of programs in `Go` [DK15], are also a proof calculus to reason about communication in concurrent

systems. The language `Go` considered in [LNTY17] has a native concept of channels for communication and a very restricted `goto` instruction. We do not consider it as a low-level language, as is used high-level communication instructions and and high-level control-flow instructions. Although the channel interactions are verified, the preservation of properties of channel implementations are not verified. To our awareness, there are no session types for low-level languages.

Rely/Guarantee (also known as Assume/Guarantee or Assumption/Commitment) was first published in [Jon81]. It is a proof calculus to reason about concurrent components that communicate via shared variables. It is a descendant of the traditional Hoare calculus. In addition to the pre- and postcondition, it also defines invariants that are used to formalize the effect a process has on the shared resources (*guarantee*) and the state of the shared resources it can *rely* on. When combining the reasoning of concurrent components, the guarantees of the components need to imply the relied upon assertions of the other components. The Rely/Guarantee approach is geared towards shared variable communication and therefore suited for a formalization of the proofs about our language using shared variables (SV) and the handshake refinement in Section 6.1 in Isabelle/HOL in future work. A Rely/Guarantee calculus for the `x86` instruction set was formalized in [Rid10]. We do not use the Rely/Guarantee approach for our proof calculus in Chapter 5 because the invariants consider changes between pairs of states and their successor states, whereas we are interested in the capturing of the abstract communication behavior in the form of stable failures.

In this section, we have presented compositional proof calculi for concurrent systems. Because of our design of our framework, we do not need to consider concurrent components in the phase where we want to employ a proof calculus. We only consider communication behavior of single components and can compose the results due to the compositionality our framework inherited from CSP. Different formalizations of a concurrent proof calculus for CUC are possible. We choose to extend a Hoare logic with communication invariants, as it goes along well with the stable failures semantics, and yields an elegant solution (in our opinion). Sessions types might be a viable alternative solution. To simplify the reasoning about further communication protocols to implement the abstract communication in future work, the Rely-Guarantee approach to designing a calculus is well suited.

3.3 Relating Synchrony and Asynchrony

In this section, we present fundamental work concerned with the relation between languages with synchronous and asynchronous semantics and/or communication and their (relative) expressiveness. Note that synchrony of semantics does not imply synchrony of communication. Where synchronous semantics denotes that all components must take their steps at the same time (e. g., SCCS, see below), synchronous communication denotes that components taking part in a communication have to wait until all partaking components are ready to communicate. Examples are multi-way synchronization (CSP) or handshake communication also called two-way rendez-vous (CCS). The counterpart to synchronous semantics are interleaving semantics, which can have a synchronous step (as in CSP or CCS) or be pure interleaving semantics without a synchronous step (such as our language SV). Communication

can be classified as synchronous (all wait), blocking (receivers wait) and non-blocking (nobody waits). Non-blocking communication is asynchronous communication as neither sender nor receiver wait for another. In blocking communication, there is *some* sort of synchronization on the communication, so it can be viewed both as synchronous communication and asynchronous communication.

In [Mil83], Milner introduces a process calculus with synchronous semantics (SCCS). He shows that his previously defined *Calculus of Communicating Systems* (CCS) [Mil80], which has an interleaving semantics, can be embedded. However, the work is not concerned with the question whether *communication* is synchronous or asynchronous but only uses synchronous communication. Subsequently, it does not investigate synchronization mechanisms for communication.

Broy and Olderog investigate in [BO01] the relationship between synchronous and asynchronous communication where asynchronous communication is *buffered*, e. g., via an additional buffer process. However, as we want to verify the low-level implementation of communication, we do not consider high-level constructs such as buffers in our implementation language. Apart from the different abstraction level used by Broy and Olderog, their transformation from synchronous to asynchronous systems is to introduce buffers for all (previously synchronous) communication. In doing so, they lose synchronicity and the "refusal structure" of the synchronous specification, i. e., the transformation does not preserve liveness properties.

In [Pet12], Peters compares the expressiveness of different synchronous and asynchronous process calculi (variants). She uses the concept of translations with certain requirements from one language into another, resulting in what she calls *Translational Expressiveness*. Her focus is to define and show encodings (or their absence) with a good quality (defined by several criteria), whereas our focus in this thesis is to bridge the gap between process calculi and low-level code. As in [Pet12] both target and source languages are process calculi, there is a more generous use of parallel composition, whereas we allow only top-level parallelism, and at least for our handshake protocol, refrain from using additional concurrent components. Although it can be argued that conceptually we use additional components (the channels), whose logic is included in the existing components and whose data is stored in the shared memory. In future work, a classification of our encoding (implementation) with similar criteria as used by Peters seems reasonable, especially when considering different synchronization protocols and comparing them.

Protocol implementations usually require multiple semantical steps, as writing and reading of data and synchronizations are individual steps. Peters defines the notion of *emulation*, which denotes all steps in the implementation required to perform similar behavior to the emulated step in the specification. We define emulation in a similar way:

Definition 3.1: Emulation

All steps in the implementation required to implement a step of the specification together are called **emulation**. The step in the specification is then called *emulated*. Usually, the emulation consists of all the steps of the protocol execution.

The fact that the implementation emulates the synchronization behavior of the specifica-

tion gives rise to an asymmetric relation between them. The liveness-preserving relations presented in the next section all deal with this asymmetry.

In this section, we have presented approaches to encode one process calculus (variant) in another. In contrast, we provide a methodology to show that the protocols implemented in a low-level language are indeed encodings of their more abstract specifications. We are not concerned with the question of encodability itself. As our low-level language contains mechanism for synchronization over shared variables, and is therefore more general, all process calculi should be encodable in it. We leave the question of the quality of those encodings for future work.

3.4 Liveness-Preserving Implementation Relations

In this section, we discuss existing relations that aim at relating implementations of synchronization to their abstract counterpart. In particular, we focus on the distribution of simultaneous decisions over multiple steps. Before we present the approaches, we illustrate the problem of splitting up simultaneous decisions.

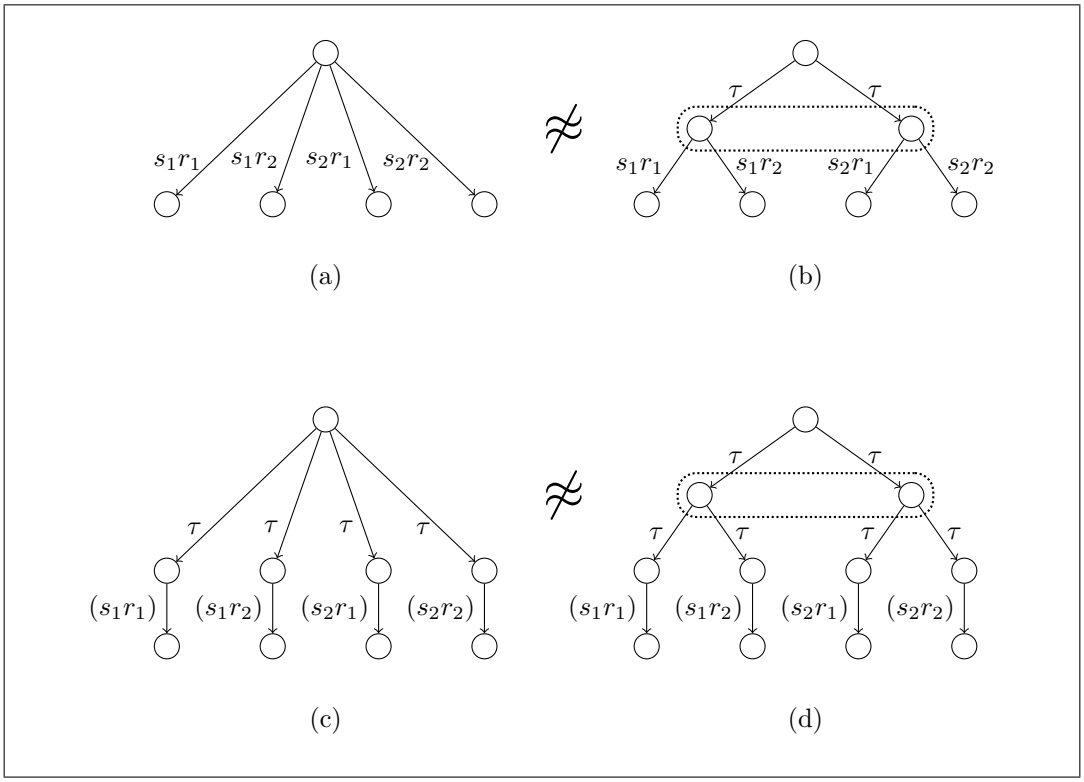


Figure 3.1: Simultaneous Decisions and Their Split-up Implementations

Consider two senders s_1, s_2 and two receivers r_1, r_2 ready to communicate synchronously over the same channel. The channel can only process one communication between a single

sender and single receiver at the time. Using abstract communication, where both sender and receiver are determined at the same time, we have four possible semantical steps that coincide with the simultaneous decisions, one for each combination of senders and receivers. In Figure 3.1a this situation is depicted for CSP-style communication (communication remains visible). In Figure 3.1c, the same situation is depicted for CCS-style communication (successful communication is internal), where we have added observational events to observe who has communicated. When implementing the abstract communication with a protocol, the sender and the receiver are determined not simultaneously, but sequentially. This situation is depicted in Figures 3.1b and 3.1d. We first have an internal event, where the sender is selected, and then the receiver is selected and the communication can happen. In the case of CCS-style communication, we have added again the observational events. Bisimulation relations, both weak and strong, preserve the branching structure of an LTS. As apparent from the different branching structures, 3.1a and 3.1b as well as 3.1c and 3.1d are not (weakly) bisimilar, respectively. The reason for this is that the middle layer in 3.1b and 3.1d (surrounded by dotted lines) has no (weakly) bisimilar counterpart in 3.1a and 3.1c, respectively. In other words, we cannot find a state in 3.1a/c that only offers the choice between the receivers.

The following relations are all concerned with splitting of single events into multiple events (often internal). The latter two especially are concerned with splitting up of decisions and tackle the problem of split-up decisions for CCS style communication. All three have their roots in bisimulation and respect the refusal structure, and, thus, preserve safety and liveness properties.

3.4.1 Vertical Bisimulation

Vertical Bisimulation by Rensink and Gorrieri [RG97] provides a congruence relation for action refinements whose implementations can interleave. The idea is to extend a bisimulation with parts where a single action in the specification can be implemented by multiple steps. These additional steps can lead to intermediate states, which do not have a bisimilar counterpart in the specification. Their definition is different from the standard bisimulation in that it keeps track of started executions of the implementations. This is facilitated by extending the usual binary relation of processes P and Q with a third place for the residual state R. Their relation is asymmetric, i. e., if the triplet (P, R, Q) is in a *vertical bisimulation* relation, then P is implemented by Q (but usually not vice versa). The progress in Q for the emulation of a single event from P is recorded in R. This allows for relating successors of Q at different stages all to be related to the same P, and at the same time being treated differently as they can be discriminated by R (the progress of the emulation).

A relation is a *vertical bisimulation*, if it fulfills the definitions of three relations, which we briefly describe: The *up-simulation*, the *down-simulation*, and the *strict residual simulation*.

The **up-simulation** ensures that the implementation is simulated by the specification, i. e., that the behavior of the implementation is bounded by the behavior of the specification. The *up-simulation* ensures the preservation of safety properties.

The **down-simulation** ensures that the implementation does at least what is specified. The specification is simulated by the implementation, but the *down-simulation* considers only states where no emulation is currently in progress, i. e., where the residual state is empty.

This enables the asymmetry of the *vertical bisimulation* (the "vertical" part). In the *vertical bisimulation*, all steps of an emulation are required to be labeled with different visible events. The first step is labeled with the emulated event in the specification. The reachability of the next state where no emulation is in progress is ensured by the *strict residual simulation*.

The **strict residual simulation** states that while there are unfinished emulations, those emulations can progress while the specification remains in its state. Assuming terminating emulations, this ensures that a state is reachable where every emulation currently in progress is finished (and no new one is started). Note that only the emulations (protocol implementations) are assumed to be terminating, not the entire programs. Furthermore, the *strict residual simulation* only ensures reachability and does not guarantee that those states are visited during an execution. In a concurrent implementation, multiple emulations can interleave. The *strict residual simulation*, thus, ensures the reachability of "emulation free" states and with that the linearizability of the emulations.

In the *vertical bisimulation*, Rensink and Gorrieri did not consider the refinement of the synchronization mechanism itself: Both source language and target language use the same CSP-like synchronization. We use the idea of augmenting a (bisimulation) relation with a third element which keeps track of the progress of emulations to be able to relate intermediate states of the emulation to the specification, although they have different communication capabilities.

3.4.2 Coupled Simulation

Coupled simulation was introduced by Parrow and Sjödin [PS92] to relate the emulation of a multi-way synchronization to its specification. The intent is to create an equivalence where "choice can be resolved gradually over several steps"[PS92]. In the resulting relation, internal choice is associative, i. e., internal decisions which occur at the same time can be split up into multiple, sequential internal decisions.

Coupled simulation consists of two mutual weak simulations, which are coupled at certain pairs. The underlying idea is similar to the *vertical bisimulation* where also two simulations are combined. Where the *vertical bisimulation* weakens a bisimulation (a single relation must fulfill the simulations in both directions), the coupled simulations strengthen mutual simulations. Formally:

Definition 3.2: Coupled Simulation

Two weak simulations $\mathcal{S}_1$ and $\mathcal{S}_2^{-1}$ are a **coupled simulation** if

1) $(P, Q) \in \mathcal{S}_1 \wedge Pstable \Longrightarrow (P, Q) \in \mathcal{S}_2$

2) $(P, Q) \in \mathcal{S}_2 \wedge Qstable \Longrightarrow (P, Q) \in \mathcal{S}_1$

In contrast to *vertical bisimulation* and our handshake refinement, coupled simulation allows for implementations in both directions due to its symmetric definition.

Although Parrow and Sjödin implement the CSP-typical multi-way synchronization, their specification language differs in an important aspect from CSP: Successful communication is hidden to the outside. This is common for CCS and its descendants. Hiding successful

communication leads to different semantics. Consider a parallel combination of multiple components which successfully communicate. It can be "unfolded" with external choice in CSP, but it is "unfolded" with internal choice in CCS. Due to successful communication being internal, they use observational events to show behavioral equivalence whereas we do not use observational events but relate the communication events directly. This enables us to make assertions about their availability.

Furthermore, coupled simulation relates only stable states and not concrete implementations of communication. It does not distinguish between internal steps that are necessary for the implementation of the communication and other internal steps. Therefore, it loses information, which can be valuable, e. g., when considering divergences outside of the implementation of communication. In contrast, our handshake refinement also relates internal steps outside of the implementation of the communication, where it ensures structural equivalence like a strong bisimulation. Thus, it ensures that divergences cannot be introduced outside of the implementation of the communication.

3.4.3 Global Bisimulation

In [dFG06], Frutos-Escrig et al. present *global (timed) bisimulation*, which allows for associativity of internal choices, i. e., nested internal choices are globally bisimilar to the single flattened internal choice between all the possibilities of the different branches. The problem of relating nested choices with the corresponding flattened version is related to our problem, as we need to decouple the decisions who is the sender and who is the receiver, thus, trying to relate two decisions at once and two consecutive decisions. However, the nested choices we want to flatten are not internal choices, but visible events (different possibilities of components synchronizing). The idea of global bisimulation is to augment the original transition system with so called *dynamic transitions*, which model unifying multiple levels of internal choices and also separating internal choices into multiple levels for the inverse direction. It is a symmetric relation, weaker than standard bisimulations. According to the authors, it is specifically designed as a simpler alternative to the symmetric relation obtained by applying an asymmetric relation in both directions, e. g., how traces equivalence between two processes A and B is defined as A being a traces refinement of B and B being a traces refinement of A. The approach solves a similar problem as coupled simulations (while being much more complicated in proofs due to the huge amount of additional dynamic transitions to be considered) and is also not applicable to our problem, as it only allows distribution of internal choices whereas we consider visible events. We do not benefit from the symmetry of the relation, as we want to relate specifications with their implementations.

None of the presented implementation relations solves our problem. However, we take the *vertical bisimulation* as a basis for our relation and incorporate the idea to only "apply" the down-simulation at stable states of the emulation from the coupled simulation.

3.5 Implementing Synchronous Communication

In this section, we give an overview over approaches to implement synchronous communication similar to the communication of CSP. The related work in this section is related to the second

half of our approach, the relation between CUC and an implementation of a synchronization protocol in SV.

As already mentioned in Section 3.4.2, Parrow and Sjödin [PS92] implement multi-way synchronization in a process calculus with asynchronous, non-blocking communication. They use additional components (mediators and ports) to implement a distributed synchronizer. The mediators communicate on the behalf of the processes via the ports. The ports ensure that the synchronization of the multiple components only takes place if all potential components participate. To his end, they use a two-phases-synchronization protocol. However, their target language is an asynchronous process calculus, and not a low-level language. Furthermore, the details of the protocol execution are implemented in a predicative way, underlining the abstract nature of their implementation. We use a low-level language as a target language and opted for a simpler protocol (unidirectional communication with two participants) for the sake of readability. In future work, we could extend our approach to multi-way synchronization by adapting their protocol.

Basu et al. [BBO12] define *synchronizability* of asynchronous systems. They use this notion to show that it is appropriate to consider a synchronized version of a synchronizable asynchronous system. In their setting, the synchronous system uses handshake communication (similar to CCS), and the asynchronous system communicates via non-blocking queues. Their main result is an induction over the length of blocking queues, generalizing to blocking queues of arbitrary length. However, queues are a high-level construct, and as such, not part of a low-level language.

In [Pee04], Peeters models CSP-like communication in hardware using handshake protocols. In hardware, low-level communication is synchronous (a wire from sender to recipient). Thus, he uses synchronization primitives for the protocol implementation, which is not the problem we are considering.

A different approach is taken by Gardner [Gar03]. He constructs a communication backbone from a CSP process, which can then be enriched with C++ code. Although the idea of this framework is akin to correct by construction design, the verification of the framework itself is not addressed.

In this section, we have presented four different approaches concerned with the implementation of synchronous communication as used in CSP. None of the approaches considers a *low-level* implementation with asynchronous primitives.

In this chapter, we have given an overview of related work. We have presented approaches for formalizations of low-level languages, compositional proof calculi for concurrent systems, how to relate synchrony and asynchrony, liveness preserving implementation relations, and implementations of synchronous communication. We have shown that none of the approaches covered our problem of relating abstract specifications with low-level executable code while preserving safety and liveness properties. In the next chapter, we illustrate our approach.

Chapter 4

Approach

In this chapter, we present our approach to relate an abstract specification and its low-level implementation. One of the main challenges when verifying concurrent systems is the exponential increase in the number of possible interactions and the number of possible combined states with the number of components. We solve this problem by extending the notion of *Communicating Sequential Processes* (CSP) refinement and defining compatible semantics and relations. Doing so, we inherit the compositionality of CSP. One of the main advantages of our approach is that interactions only need to be verified on an abstract level. Then, each concurrent component can be refined individually, preserving the result of the verification of the interactions.

Our approach to relate an abstract specification and a low-level implementation is a two-step process. First, we focus on the transition from an abstract process in CSP to a low-level program and keep the abstract communication and the ability to reason (concurrently) compositional. This results in a compositional relation of CSP and our intermediate language *Communicating Unstructured Code* (CUC), a low-level language with abstract CSP-like communication. In a second step, we focus on the transition from abstract communication to a low-level implementation of communication, which we solve for all programs that use a communication protocol we verify. This results in a relation of CUC and our low-level language *Shared Variables* (SV), which is similar to CUC, however has low-level communication instructions instead of the abstract communication mechanism. Thus, SV contains only low-level instructions. We verify a handshake protocol and show that all CUC programs and similar SV programs that use the verified protocol are related. Finally, we can combine the relation from CSP to CUC and the relation from CUC to SV to ensure the preservation of safety and liveness properties from an abstract specification in CSP to a low-level implementation in SV. The entire framework is rigorous and scales well with the number of components.

Our proposed framework has five parts as depicted in Figure 4.1. The Figure is identical to Figure 1.1 in the introduction. However, this time we will go through the parts of the framework in the order they are presented in this thesis. The order of the five parts is as follows:

L1) The specification in CSP

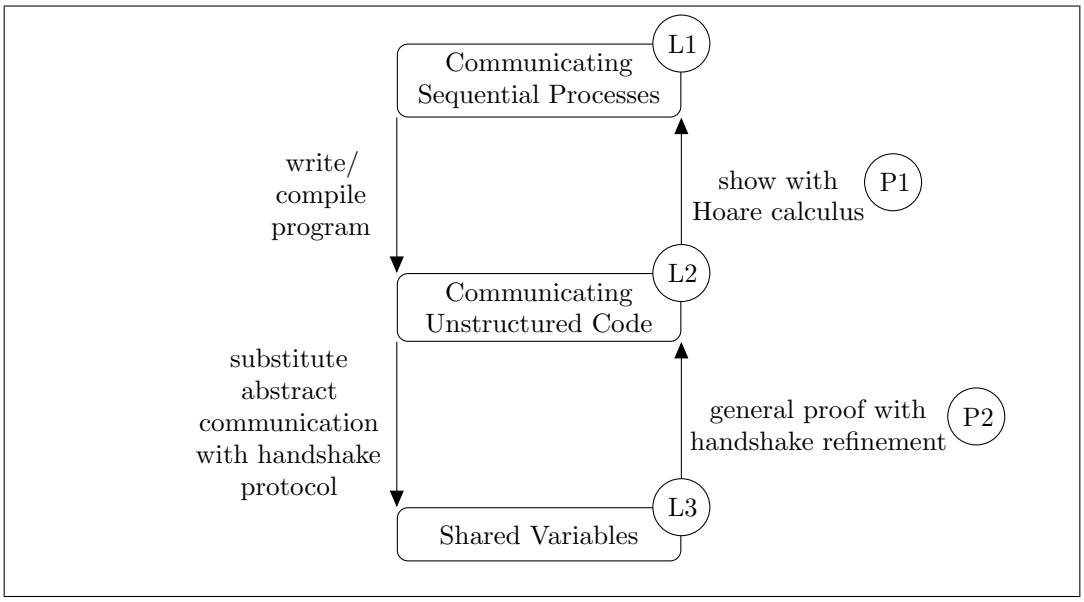


Figure 4.1: Overview

L2) Our intermediate language CUC

P1) Our Hoare calculus to relate CSP and CUC

L3) Our low-level language SV

P2) Our handshake refinement to relate CUC and SV

L1) We use CSP as abstract specification language and extend the notion of CSP refinement to unstructured low-level code. We require our specification language to model concurrency, communication and non-termination. Especially important in concurrent settings is (concurrent) compositionality: The ability to reason about the components individually and compose the results. All of these properties are core to CSP. CSP is also suited for an iterative development process. Its notion of refinement allows us to relate abstract and more specific CSP models while preserving safety and liveness properties across abstraction levels. However, CSP refinement is only defined between CSP processes. With our extension of the refinement notion of CSP to relate CSP processes and unstructured low-level code, we close the verification gap from abstract specification language to executable low-level language.

We focus on liveness properties in terms of stable failures. We discuss in Future Work (see Section 8.3) that our framework can be extended to also reason about divergences, which additionally capture livelocks

We assume the CSP specification to be refined to an "implementation"-level within the standard CSP refinement. In particular, we only permit top-level parallelism. Furthermore, as we relate the specification and the implementation by their communication behavior, we focus on communication events. We ignore observational events, which do not have a direct counterpart in programs and, thus, would require annotations. Our approach is fully compatible with observational events and can be extended if needed.

L2) We define our intermediate language CUC, which combines low-level instructions to change the state and the control flow with the abstract communication mechanism of CSP. This enables us to focus on the internals of each component separately while retaining the compositionality of the communication mechanism of CSP. The abstract communication can later be instantiated with different protocols. We define operational and denotational semantics for CUC and show their correspondence. The denotational semantics for CUC consist of a traces semantics to formulate safety properties and a stable failures semantics to formulate liveness properties.

P1) We present a Hoare calculus to relate a CSP process and a CUC program via their communication behavior (their stable failures). As CUC uses the communication mechanism of CSP, we can rely on the compositionality of the parallel operator of CSP and relate the CSP process and the CUC program for each component individually, retaining the (concurrent) compositionality of CSP. CSP processes and low-level programs are structurally different. Especially due to the fact that in CSP internal computations are abstracted away and in a low-level program they usually constitute the largest part. To overcome this problem, we first state an intermediate property, i. e., a property that is *only* true for the stable failures of the CSP process under consideration, and prove that it is sufficient (in the mathematical sense). We call this intermediate property "sufficient property". Then, we use our Hoare calculus to show this sufficient property correct for the CUC program. This approach has the benefits that *a)* it is modular, i. e., another low-level representation can be used to fulfill this sufficient property, and *b)* using our Hoare calculus, we can reason (sequentially) compositionally about the low-level code. With our Hoare calculus, we can reason about communicating and non-terminating programs, by using and showing the sufficient property as an invariant. Currently, the formulation of the sufficient property has to be done manually. Its proof with respect to the CSP process can be conducted with mechanical assistance in Isabelle/HOL.

L3) We define our low-level language SV. One way to implement multiple communicating components is to use a single machine with a thread-like structure and shared memory. Our low-level language SV has local memory for each component and shared memory for inter-thread communication. SV differs from CUC in that it does not have the abstract communication instruction. Instead, it has instructions to read from and write to shared memory and an atomic compare-and-set instruction. With SV, we cover a large subset of common low-level instructions, i. e., instructions from common *Instruction Set Architectures*.

P2) We present an example of how to show the relation between CUC and SV. To this end, we consider a simple handshake protocol that implements the synchronous communication of two parties over a channel. Therefore, we restrict the multi-way communication of CUC to unidirectional channels. We define our notion of handshake refinement. Its idea is rooted in bisimulation. However, it takes into account the emulation of the communication, which is distributed over multiple steps and multiple components, making it asymmetric. Our handshake refinement preserves safety and liveness properties. We define the notion of an SV program *fitting* a CUC program, which can be automated with a simple syntax check. We prove a theorem that relates *every* CUC program to its fitting SV programs. This enables us to consider the whole system without losing the compositional nature of the overall approach

to relate CSP and SV. The only global properties we need for the application of the theorem are a separation of local and shared memory as well as the injectivity of the mapping from names of channel variables to actual shared variables within SV.

In this chapter, we have given an overview of our proposed framework and its parts. We enable compositional reasoning by first keeping the compositional communication of CSP from CSP to CUC, and then using a general theorem from CUC to SV. Thus, our framework scales very well with the number of components, and even better for homogeneous components, as individual refinements of components can be reused. In the next chapter, we introduce CUC and show how to relate it with CSP.

Chapter 5

Communicating Unstructured Code (CUC)

In order to bridge the verification gap between abstract specification and executable low-level code, we choose a two-step approach. Our first step out of two to relate CSP with a low-level language is to focus on the internal computations and keep the CSP communication. This enables us to keep the (concurrent) compositionality of the communication of CSP, and yet relate each process with a low-level program. To this end, we define *Communicating Unstructured Code* (CUC), a language that combines low-level control flow with abstract communication. The key idea is an abstract communication instruction `comm` that models the CSP communication mechanism and thereby allows for concurrent compositionality. Another key idea of CUC are its generic instruction schemes that can be instantiated to many different instructions of actual instruction set architectures. Apart from the genericity, the small number of instruction schemes allows for more manageable formalizations and proofs.

In this chapter we present our notion of low-level code model (5.1), the syntax and semantics of CUC (5.2 and 5.3), a Hoare calculus to reason about its possibly non-terminating behaviors (5.4), and finally a method to show a stable failures refinement between a CSP process and a CUC program (5.5).

We published a precursor to CUC without communication in [BJ14], the syntax and its operational and traces semantics in [JGG15], and the stable failures semantics, the Hoare calculus and the method to show the stable failures refinement in [JGG16].

All results of this chapter apart from the concurrent cases in Theorems 5.1 and 5.2 are formalized in Isabelle/HOL in [BBD+19], although we reworked the semantics in this thesis for better presentation. The proof for the concurrent cases in Theorems 5.1 and 5.2 can be found in the Appendix A.1. All definitions, assumptions, lemmas and theorems that are included in the Isabelle/HOL formalization in one form or another (sometimes only implicitly) are marked with the following symbol: . If they are only partly included in the Isabelle/HOL formalization (such as the aforementioned theorems), they are marked with *. In Appendix A.3, we give a mapping from the formalization in this thesis to the formalization in the Isabelle/HOL formalization.

5.1 Low-Level Code Model

In this section, we describe our understanding of a low-level code model and discuss briefly which concepts our framework supports. The aim of our framework is to cover all abstract concepts of low-level languages. Our framework aims at the level of Instruction Set Architectures, as these are the lowest level of software. Current instructions sets all cover similar abstract concepts. We choose the instruction set of RISC-V [WA17] as an exemplary assembly language for illustration for the following reasons: RISC-V is the subject of current research (e. g., [PKND18, FRC+18, Cho18]) and it is also very interesting for the industry: The board of directors of the RISC-V foundation includes companies like Google, NVIDIA, NXP, and Western Digital and several large scale implementations are planned [Ros18].

We aim at formalizing low-level user level programs (i. e., not interactions with the operating system). According to [Wat16], the user level instructions of RISC-V can be divided into the following categories of instructions (including both the base instruction set and the extensions). We group the categories into three kinds. We discuss afterwards how we intend to cover each kind of instructions.

I: State Transformations:

- System instructions, which include a system call, a breakpoint, and read and/or write access to control and status registers.
- Computations, of arithmetic or logical nature
- (Optimized) Integer Multiplication and Division
- Floating-Point Arithmetics.
- Memory Access, e. g., load and store which transfer data between the registers and the memory.

II: Control Flow, i. e., computed[1] and uncomputed jumps

III: Multi-Processor Synchronization

I State Transformations: We consider the program counter, the (local) registers, and the (shared) memory as the state of the system under consideration. The instructions grouped under State Transformations modify the state. All instructions additionally increase the program counter by one instruction. As we are interested in covering the abstract concepts of low-level languages represented by the three kinds of instructions I, II and III, we use an abstract representation of the local state consolidating both registers and the local memory. This enables the instantiation of our framework with different memory models and simplifies our proofs. We use a mapping from names to values to model the state (apart from the program counter). In low-level terms, this would correspond to a direct addressing of separated memory blocks (i. e., pointer arithmetics is not possible). However, in Section 8.3, we discuss the possibility to replace the abstract representation of the state with a more fine grained representation, taking the separation of register and memory into account, as well as block sizes. Most instructions, e. g., arithmetic and boolean operations,

[1] They are called *conditional* branches in RISC-V. However, they read the jump target from a register. To avoid confusion with the conditional branch instruction we define in Section 5.2 which has fixed jumps targets, we call the RISC-V command *computed* jump.

transfer of data between different parts of the state, or setting of special registers, can be modeled as a transformation of the function representing the state. We use an instruction scheme to model state transformations. This instruction scheme can be instantiated to match the desired instructions. This enables us to represent all state transformations by a single instruction scheme, which vastly reduces the formalization efforts, while being applicable to many instruction sets. We do not model system calls, as we do not model the operating system.

II Control Flow: Control flow at the ISA level consists of modifying the program counter. There are uncomputed jumps, which always jump to the same target, and there are conditional and computed jumps, which jump to different targets based on a condition or a previous computation. Our framework covers conditional jumps with fixed jump targets. Thus, control flow depending on the current state is possible, but the jump targets cannot be computed (i. e., read from a regular register). Our framework does not cover computed jumps. However, in Section 8.3 we argue that computed jumps can be integrated into our framework using the technique of *Control Flow Integrity* [ABEL05] that limits the legal jump targets for computed jumps and is a standard feature of current compilers. The same expressiveness of choosing from a list of possible jump targets can be constructed in the current framework by chaining conditional jumps. Uncomputed jumps can be modeled using conditional jumps. Note that we cannot model directly the *jump-and-link* instructions of RISC-V, which modify the program counter *and* a register, as we choose to separate control flow from state transformations for cleaner semantics and proofs. Conceptually, we can combine control flow and state transformations into one instruction scheme, which we leave for future work.

III Multi-Processor Synchronization: The category of multi-processor synchronization contains instructions to explicitly synchronize components or avoid data races. To this end, it is necessary to inspect a shared variable (called a semaphore) and set it subsequently in an atomic fashion, i. e., without another component modifying the resource in the meantime. In the first step of our framework, which is described in this chapter, we use an abstract communication instruction. In the second step of our framework, which is described in Chapter 6, we model the multi-processor synchronization with the instruction *compare-and-set* (`cas`) to cover this concept. It compares the value of a shared variable with a given value. If the two values match, the shared variable is set to a third value. RISC-V uses the two instructions *load-reserve* and *store-conditional* to construct atomic operations. The instruction *load-reserve* loads a value from memory and marks the address as reserved. The instruction *store-conditional* attempts to store a value, but only succeeds, if the address in memory is still reserved. The *compare-and-set* operation can be implemented using *load-reserve* and *store-conditional*. While the combination of *load-reserve* and *store-conditional* is a bit more flexible than *compare-and-set*, they fulfill the same role to construct atomic complex operations. The drawback of the combination of *load-reserve* and *store-conditional* is that in general its naive use tends to create livelocks. According to [Wat16], the livelocks can be prevented by complying to certain restrictions, e. g., limiting the number of instructions between *load-reserve* and *store-conditional* and requirements to the implementation of the architecture. We use the instruction *compare-and-set* as it requires less instructions than

the combination of *load-reserve* and *store-conditional*, which helps us to keep the instruction set and programs small and simple. Furthermore, we do not have to consider the additional requirements to prevent the livelocks. We leave the replacement of the `cas` instruction with the *load-reserve* and *store-conditional* instructions in our framework for future work.

Having discussed how we intend to cover the different aspects of low-level code, we make a brief note on the unstructured nature of low-level code.

We consider unstructured programs, which do not have structuring statements known from high-level languages such as *if-then* or *while*. We consider branching instructions as "normal" instructions that modify the program counter, which points to the instruction which is to be executed next. Thus, when we combine two instructions, we might create a loop. Therefore, our sequential composition operator $\oplus$ which we introduce in Section 5.2 also serves as looping construct.

In this section, we have described our understanding of low-level code and discussed to what extend and in which manner we aim at modeling it. We identified three kinds of instructions and cover all of them with some restrictions. In the next sections, we present the syntax and semantics of our low-level language with abstract communication CUC.

5.2 Syntax and Semantic States

In this section, we present the semantic states of CUC, its instructions and consequently its programs. We give informal semantics for the instructions.

In Definition 5.1, we provide basic data types that we use throughout the thesis. The semantics operates on (concurrent) states ($\sigma \in States$) that are either local states ($\sigma \in LStates$) or a concurrent composition of states ($States \parallel States$). We define the concurrent states recursively with a tree structure (instead of in a set or list), as it makes it more amenable for future definitions and proofs. If needed, the "concurrency trees" can be flattened to a set or list. The local states each have a data store ($ds \in DS$) and a program counter ($pc \in Labels$). The data store is modeled as a function from names to values. If needed, it can model separate parts, e. g., registers and memory. We usually simply refer to registers stored in the data store. The program counter points to the instruction that is to be executed next. We denote the data store of a local state σ with σ_{ds} and its program counter with σ_{pc}. Communication is done via events ev from an alphabet Σ, which we define as in CSP. We also define the set of boolean values $\mathbb{B}$.

Definition 5.1: Basic Data Types

$$
\begin{aligned}
DS &:= Names \to Values \\
Labels &:= \mathbb{N} \\
LStates &:= Labels \times DS \\
States &:= LStates \mid States \parallel States \\
\Sigma & \quad \text{(the set of all events)} \\
\mathbb{B} &:= \{true, false\}
\end{aligned}
$$

The language CUC consists of three instruction schemes: A (non-deterministic) state transformation `do`, a conditional branch instruction `cbr`, and an abstract communication instruction `comm`. Let $\mathscr{P}(S)$ denote the power set of the set S.

Definition 5.2: Instructions of CUC

$$
\begin{aligned}
Instructions := \ & \texttt{do}\ f && f\colon DS \to \mathscr{P}(DS) \\
\mid\ & \texttt{cbr}\ b\ m\ n && b\colon DS \to \mathbb{B},\ m, n\colon Labels \\
\mid\ & \texttt{comm}\ f_{ev}\ f_{ds} && f_{ev}\colon DS \to \mathscr{P}(\Sigma),\ f_{ds}\colon DS \times \Sigma \to DS
\end{aligned}
$$

$\underline{\texttt{do}\ f}$ takes the current state and defines the possibly multiple successor states by f. The program counter is increased by one. We restrict f to always yield at least one successor state.

Assumption 5.1: At Least One Successor State

$$\forall\, ds \in DS.\ f(ds) \neq \varnothing$$

Example 5.1: Instantiations of `do` f

`do` f can be instantiated to model common instructions such as e. g., `add`, load (`ld`) or store (`sd`) from RISC-V. Let $ds[x := v]$ be the function that maps x to v and otherwise maps names to values as ds does. For load and store, we model the difference between registers and memory with name spaces, i. e., $R.x$ is a register address and $M.x$ is a memory address. `do` f can also be used to model reading data of type $\mathbb{T}$ from a sensor.

$$
\begin{aligned}
\texttt{add}\ c\ a\ b &:= \texttt{do}\ \lambda ds.\ \{ds[c := ds(a) + ds(b)]\} \\
\texttt{sd}\ R.v\ M.x &:= \texttt{do}\ \lambda ds.\ \{ds[M.x := ds(R.v)]\} \\
\texttt{ld}\ R.x\ M.v &:= \texttt{do}\ \lambda ds.\ \{ds[R.x := ds(M.v)]\} \\
\texttt{read\_sensor}\ x &:= \texttt{do}\ \lambda ds.\ \{ds[x := v] \mid v \in \mathbb{T}\}
\end{aligned}
$$

`cbr` b m n evaluates the predicate b on the current state and, if *true*, sets the *pc* to m, otherwise to n.

Example 5.2: Instantiations of `cbr` b m n

Under the assumption that each instruction has a unique label ℓ, `cbr` can be instantiated to `BEQ` (branch if eqal) or `CJ` (compressed jump). `BEQ` branches if the values stored in the registers r_1 and r_2 are equal and `CJ` unconditionally jumps. Let "_" denote an unused and therefore arbitrary value.

$$\ell\colon \texttt{BEQ}\ r_1\ r_2\ \mathit{offset} \coloneqq \texttt{cbr}\ \big(\lambda ds.\ ds(r_1) = ds(r_2)\big)\ (\ell + \mathit{offset})\ (\ell + 1)$$
$$\ell\colon \texttt{CJ}\ \mathit{offset} \coloneqq \texttt{cbr}\ (\lambda ds.\ \mathit{true})\ (\ell + \mathit{offset})\ \_$$

`comm` f_{ev} f_{ds} uses f_{ev} to determine from the current state which events it offers to communicate. f_{ds} is used to update the data store depending on the communicated event. In contrast to `do` f, the successor state is determined deterministically for simplicity.

Example 5.3: 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 defined in Section 2.6, to extract the value of an event.

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

Based on these three instruction schemes, we can define entire programs as follows. A local CUC program $lp \in LP$ is a set of labeled instructions, i. e., pairs of labels and instructions.

Definition 5.3: Local Program lp

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

Example 5.4: Local Program lp_{ex}

$$lp_{ex} := \Big\{ \big(1, \texttt{do}\ (\lambda ds.\ \{ds[x := 5]\})\big), \\ \big(2, \texttt{comm}\ (\lambda ds.\ \{a.ds(x)\})\ (\lambda ds\ ev.\ ds)\big), \\ \big(3, \texttt{cbr}\ (\lambda ds.\ true)\ 2\ 2\big) \Big\}$$

A program is a set of labeled instructions. The instruction at Label 1 sets the register x to 5. The instruction at Label 2 communicates the value of register x over channel a. The instruction at Label 3 jumps back to label 2. If started with a state with the program counter set to 1, the overall program communicates $a.5$ indefinitely.

We require the labels for $lp \in LP$ to be unique, i. e., the same labels imply the same instructions.

Assumption 5.2: Uniqueness of Labels

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

To talk about the labels of a program and in particular to determine whether a pc does point into a program, we define a function that returns the labels of a program.

Definition 5.4: Labels of a Program *labels*

$$labels(lp) := \{\ell \mid \exists\, ins.\ (\ell, ins) \in lp\}$$

We write $\ell \in_{pc} lp := \ell \in labels(lp)$.

Example 5.5: Labels of a Program

$$labels(lp_{ex}) = \{1, 2, 3\}$$

$$2 \in_{pc} lp_{ex}$$

To define concurrent programs $cp \in CP$ from multiple local programs, we define an operator to combine programs into concurrent programs. For compatibility with CSP, we use its *alphabetized parallel* operator, which defines the communication interfaces for each of the two (concurrent) programs.

Definition 5.5: Concurrent Program cp

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

Note that we only allow the concurrent combination of local or concurrent programs. Inside a local program, we do not allow for concurrency. This allows for reasoning about purely sequential components. We could flatten the tree and obtain a set of local programs. Creation of a finite number of new components can be mimicked by activating idle components via communication.

Example 5.6: Concurrent Program cp_{ex}

$$\begin{aligned} lp'_{ex} &:= \Big\{ \big(1, \texttt{comm}\ (\lambda ds.\ \{a.v | v \in \mathbb{T}\})\ (\lambda ds\ ev.\ ds[y := val(ev)])\big), \\ &\qquad \big(2, \texttt{cbr}\ (\lambda ds.\ true)\ 1\ 1\big) \Big\} \\ cp_{ex} &:= \quad lp_{ex\ \{a.v|v\in\mathbb{T}\}} \|_{\{a.v|v\in\mathbb{T}\}}\ lp'_{ex} \end{aligned}$$

For a given concurrent state and its associated concurrent program we always assume that they have the same $\|$-tree structure. This allows for a uniform handling of the concurrent structures.

Assumption 5.3: 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.

To facilitate (sequentially) compositional reasoning about CUC programs, we use a technique from Saabas and Uustalu [SU05] to define a tree structure on CUC programs which yields *structured programs* $sp \in SP$. To "forget" the structure we define the unstructuring function $\mathcal{U}$. Similar to the labeled instructions, a single instruction is a pair of a label (from *Labels*) and an instruction (from *Instructions*). We use the operator $\oplus$ to denote *sequential composition*. We will use this structure for the denotational semantics (in 5.3.3 and 5.3.4) and the Hoare calculus (in 5.4). Different structures of the same labeled instructions have the same semantics, as we show in Corollary 5.1.

Definition 5.6: Structured Program sp

$$SP := Labels \times Instructions \mid SP \oplus SP$$

Example 5.7: Structured Program sp_{ex}

$$\begin{aligned} sp_{ex} := \quad & \big(1, \texttt{do}\ (\lambda ds.\ \{ds[x := 5]\})\big) \\ & \oplus \Big(\big(2, \texttt{comm}\ (\lambda ds.\ \{a.ds(x)\})\ (\lambda ds\ ev.\ ds)\big) \\ & \quad \oplus \big(3, \texttt{cbr}\ (\lambda ds.\ true)\ 2\ 2\big) \Big) \end{aligned}$$

It follows from Corollary 5.1 that $\oplus$ is associative, so we can use parentheses as we see

Example 5.7: Structured Program sp_{ex}

fit. Note that $\oplus$ allows for the construction of loops. In this example, there is a loop containing the instructions at Label 2 and Label 3.

Definition 5.7: Unstructuring Function, $\mathcal{U}$

$$\begin{aligned} \mathcal{U}(\ell, ins) &:= \{(\ell, ins)\} \\ \mathcal{U}(sp_1 \oplus sp_2) &:= \mathcal{U}(sp_1) \cup \mathcal{U}(sp_2) \end{aligned}$$

To talk about the labels of a structured program, and in particular to determine whether a *pc* does point into a structured program, we overload the *labels* function.

Definition 5.8: Labels of a Structured Program *labels*

$$labels(sp) := labels(\mathcal{U}(sp))$$

Again we write $\ell \in_{pc} sp := \ell \in labels(sp)$.

We also require uniqueness of labels for sp.

Assumption 5.4: Uniqueness of Labels for sp

$$(\ell, ins_1) \in \mathcal{U}(sp) \wedge (\ell, ins_2) \in \mathcal{U}(sp) \Longrightarrow ins_1 = ins_2$$

We define a *structured concurrent program* $scp \in SCP$ similar to the (unstructured) concurrent programs. Again, we only allow the concurrent composition of (local) structured programs and structured concurrent programs (also called top-level parallelism/concurrency), not within (local) structured programs. We also assume that the concurrent states and the associated structured concurrent programs have the same tree structure (Assumption 5.3).

Definition 5.9: Structured Concurrent Program *scp*

$$SCP := SP \mid SCP \ {}_{\mathscr{P}(\Sigma)}\|_{\mathscr{P}(\Sigma)} \ SCP$$

Example 5.8: Structured Concurrent Program scp_{ex}

We define sp'_{ex} and compose it concurrently with sp_{ex} from Example 5.7.

$$\begin{aligned} sp'_{ex} := \quad & \big(1, \texttt{comm}\ (\lambda ds.\ \{a.v | v \in \mathbb{T}\})\ (\lambda ds\ ev.\ ds[y := val(ev)])\big) \\ & \oplus \big(2, \texttt{cbr}\ (\lambda ds.\ true)\ 1\ 1\big) \\ scp_{ex} := \quad & sp_{ex}\ {}_{\{a.v|v\in\mathbb{T}\}}\|_{\{a.v|v\in\mathbb{T}\}}\ sp'_{ex} \end{aligned}$$

Having defined semantic states and CUC programs, we proceed to define their semantics in the next section.

5.3 Semantics

We give three semantics for CUC. First, an operational (small-step) semantics in Subsection 5.3.1. It reflects the intentions of the instructions clearly and is well-suited for proofs that reason about single steps (e. g., bisimilarity proofs). To account for the fact that *all* instructions are possible jump targets, our operational semantics is only defined for the whole code. It follows that it is not sequentially compositional. Second, we also define two compositional (both sequentially and concurrently), denotational semantics: The traces semantics in Subsection 5.3.3 and the stable failures semantics in Subsection 5.3.4. They share important properties with their CSP counterparts that we use to formally relate CSP and CUC in Section 5.5. The traces semantics enables the traces refinement which preserves safety properties. The stable failures semantics enables the stable failures refinement which preserves liveness properties. The Hoare calculus in Section 5.4 is defined on top of the stable failures semantics.

5.3.1 Operational Semantics

The operational semantics of local and concurrent programs in CUC is depicted in Definition 5.10. It contains three kinds of rules: 1) Those for local steps (DO, CBR, COMM), 2) those for concurrent steps (SYNC, INTERLEAVING-LEFT, INTERLEAVING-RIGHT), and 3) those for executions (EXEC-0, EXEC-EV, EXEC-τ). It is defined on programs in LP and CP, respectively.

The local steps (DO, CBR, COMM in Definition 5.10) agree on the informal descriptions from the last section. The first assumption of each rule ensures that the respective instruction is in the labeled instruction set and that the pc points to it. `do` and `cbr` are internal transitions and labeled with τ. In DO, the data store is updated according to the function f. Multiple different successor states might be possible. The program counter is increased by one. In CBR, the data store remains unchanged and the program counter is set according to the evaluation of the predicate b. The transition of `comm` is labeled with the communicated event ev. In COMM, the set of possible events is given by the function f_{ds}. The data store is updated according to the function f_{ds} which depends on the communicated event ev. The program counter is increased by one.

The concurrent steps (SYNC, INTERLEAVING-LEFT, INTERLEAVING-RIGHT in Definition 5.10) are similar to those of CSP. In SYNC, both components offer the same event ev if the event ev is in both their communication interfaces. In INTERLEAVING-LEFT σ_1 offers the event ev, but it is allowed to make the transition independently of σ_2, as ev is not in the communication interface of σ_2. INTERLEAVING-RIGHT is analogous.

Definition 5.10: Operational Semantics of CUC *

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

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

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

$$\frac{\sigma_1 \xrightarrow{ev}_{cp_1} \sigma'_1 \qquad \sigma_2 \xrightarrow{ev}_{cp_2} \sigma'_2 \qquad ev \in \alpha_1 \cap \alpha_2}{\sigma_1 \parallel \sigma_2 \xrightarrow{ev}_{\left(cp_1 {}_{\alpha_1}\|_{\alpha_2} cp_2\right)} \sigma'_1 \parallel \sigma'_2} \text{ SYNC}$$

INTERLEAVING-LEFT

$$\frac{\sigma_1 \xrightarrow{ev}_{cp_1} \sigma'_1 \qquad ev \in (\alpha_1 \cup \{\tau\}) \setminus \alpha_2}{\sigma_1 \parallel \sigma_2 \xrightarrow{ev}_{\left(cp_1 {}_{\alpha_1}\|_{\alpha_2} cp_2\right)} \sigma'_1 \parallel \sigma_2}$$

INTERLEAVING-RIGHT

$$\frac{\sigma_2 \xrightarrow{ev}_{cp_2} \sigma'_2 \qquad ev \in (\alpha_2 \cup \{\tau\}) \setminus \alpha_1}{\sigma_1 \parallel \sigma_2 \xrightarrow{ev}_{\left(cp_1 {}_{\alpha_1}\|_{\alpha_2} cp_2\right)} \sigma_1 \parallel \sigma'_2}$$

EXEC-0

$$\frac{}{\sigma \overset{\langle\rangle}{\Longrightarrow}_{cp} \sigma}$$

EXEC-EV

$$\frac{\sigma \overset{tr'}{\Longrightarrow}_{cp} \sigma'' \quad \sigma'' \xrightarrow{ev}_{cp} \sigma' \quad tr' \frown \langle ev \rangle = tr}{\sigma \overset{tr}{\Longrightarrow}_{cp} \sigma'}$$

EXEC-τ

$$\frac{\sigma \overset{tr}{\Longrightarrow}_{cp} \sigma'' \qquad \sigma'' \xrightarrow{\tau}_{cp} \sigma'}{\sigma \overset{tr}{\Longrightarrow}_{cp} \sigma'}$$

The execution semantics (EXEC-0, EXEC-EV, EXEC-τ in Definition 5.10) is defined starting with the trivial execution, i. e., the empty trace (EXEC-0), and then prepending either visible steps and the communicated event to the trace (EXEC-EV), or prepending invisible steps and leaving the trace unchanged (EXEC-τ). Execution relations can be defined either by prepending or by appending single steps. As we only consider finite executions (as in CSP[2]), they are equivalent. We choose to append single steps, as this eases the proofs of correspondence with the denotational semantics. The execution semantics describes possibly partial executions. If we want to talk about a terminating execution, we use an additional condition that the *pc* of the final state does not point into the code:

Definition 5.11: Terminating Execution

An execution *tr* is **final** if and only if $\sigma \overset{tr}{\Rightarrow}_{cp} \sigma' \wedge \sigma'_{pc} \notin_{pc} cp$.

In Example 5.9 we illustrate the operational semantics of CUC using the program cp_{ex} from Example 5.6. In this subsection, we have defined the operational semantics for CUC. It gives the formal intuition of CUC. The operational semantics of CUC is very similar to the operational semantics of CSP in that it is labeled, concurrent composition requires communication interfaces, and it has interleaving steps and synchronous steps. In the next subsection, we give a brief introduction on how to construct denotational semantics for non-terminating programs using fixpoints. We use fixpoints for the definition of the denotational semantics of CUC in the subsequent subsections.

5.3.2 Defining Denotational Semantics with Fixpoints

In this subsection, we describe how to use fixpoints to construct denotational semantics for non-terminating programs.

In contrast to operational semantics, which describe how single instructions manipulate the state, in denotational semantics each program is assigned a function (its *denotation*) which maps an initial state to the meaning of the program. This can be, e. g., a final state or a set of traces as in the case of CSP or a set of state-trace pairs as in the semantics of CUC we define in the next subsection.

To be able to assign a denotation to programs containing loops, a function *extend* from *denotation* to *denotation* is considered, which extends the denotation with the semantics for one execution of the loop. The least fixpoint of this function *extend* is then the denotation of the program containing the loop. The initial denotation, i. e., the bottom element of the complete partial order of denotations, is the function that maps every input to the empty set.

To define a least fixpoint (and ensure its existence[3]), we use a chain-complete partial order on the denotations. A partial order is chain-complete, if every chain has a least upper bound (in the carrier set of the partial order). The chain-complete partial order on the denotations is usually obtained by lifting the chain-complete partial order on the result of the computations,

[2]In the semantical models of CSP we consider, i. e., traces and stable failures, the infinite executions are modeled with all finite prefixes of infinite executions.

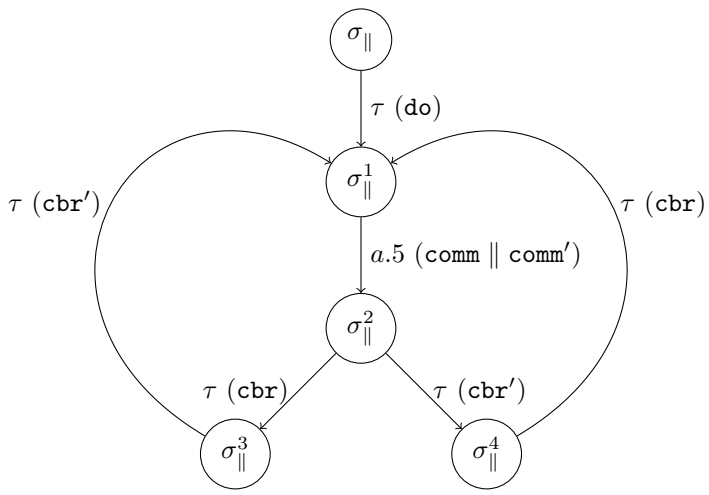
[3]The existence of fixpoints for chain-complete partial orders is ensured by the Kleene fixpoint theorem, which can be found, e. g., in [Win93].

Example 5.9: Operational Semantics of cp_{ex}

We illustrate the operational semantics of CUC by giving the labeled state graph for program cp_{ex}. We first repeat the definition of cp_{ex} for convenience.

$$
\begin{aligned}
lp_{ex} &:= \Big\{ \big(1, \texttt{do}\ (\lambda ds.\ \{ds[x := 5]\})\big), \\
&\quad \big(2, \texttt{comm}\ (\lambda ds.\ \{a.ds(x)\})\ (\lambda ds\ ev.\ ds)\big), \\
&\quad \big(3, \texttt{cbr}\ (\lambda ds.\ true)\ 2\ 2\big) \Big\} \\
lp'_{ex} &:= \Big\{ \big(1, \texttt{comm}\ (\lambda ds.\ \{a.v | v \in \mathbb{T}\})\ (\lambda ds\ ev.\ ds[y := val(ev)])\big), \\
&\quad \big(2, \texttt{cbr}\ (\lambda ds.\ true)\ 1\ 1\big) \Big\} \\
cp_{ex} &:= \quad lp_{ex}\ {}_{\{a.v|v\in\mathbb{T}\}}\|_{\{a.v|v\in\mathbb{T}\}}\ lp'_{ex}
\end{aligned}
$$

We indicate the executed instruction in parenthesis after the event, e. g., $\tau(\texttt{do})$ stands for the event τ that was caused by the instruction do being executed. Let both the program counters of the concurrent initial state $\sigma_{\|} = \sigma \parallel \sigma'$ point to the first instruction ($\sigma_{pc} = \sigma'_{pc} = 1$). From there, lp_{ex} executes `do`, reaching $\sigma^1_{\|} = \sigma^1 \parallel \sigma^{1\prime}$, where $\sigma^1_{pc} = 2$ and $\sigma^1_{ds} = \sigma_{ds}[x := 5]$. Then both components synchronously communicate $a.5$ and reach $\sigma^2_{\|}$. Afterwards, the `cbr` instructions of lp_{ex} and lp'_{ex} (indicated as $\texttt{cbr}'$) are executed interleaved and passing via $\sigma^3_{\|}$ or $\sigma^4_{\|}$, respectively, reach again $\sigma^1_{\|}$.



i.e., the co-domain of the denotations. In general, the resulting denotation may not be computable in finite time, but fixpoint induction can be used to reason about properties of denotations of programs containing loops. Fixpoint induction (or *Scott induction*, e.g., in [Win93]) requires the function *extend* to be continuous, and the property of interest I to be admissible. A function is continuous if the application of the function and the application of the supremum of a chain are interchangeable. A property is admissible, if the property holding for all elements of a chain implies that the property also holds for the supremum of the chain. The fixpoint induction then consists of showing the base case and the step case, like a regular induction.

To define the denotational semantics of CUC in Section 5.3 we lift the subset relation on sets (of trace-state pairs or failures) to functions and use the point wise subset relation to define a partial order on functions on sets (of trace-state pairs or failures). For any two functions f and g that map from sets to sets, we define the partial order as follows, where S ranges over all possible sets (of trace-state pairs or failures)

$$f \leq g \coloneqq \forall S.\ f(S) \subseteq g(S)$$

Both partial orders are chain-complete: The subset relation is chain-complete and the point-wise subset relation, too, as it disjointly combines chain-complete partial orders.

In this subsection, we have described how to use fixpoints to construct denotational semantics for non-terminating programs. We use fixpoints to construct the traces semantics and the stable failures semantics of CUC. In the next subsection, we define the traces semantics for CUC, which allows us to consider all behaviors of a (concurrent) program at once. It allows for the formulation of safety properties.

5.3.3 Traces Semantics

In this subsection, we define the traces semantics for CUC. We first give the operational characterization of the traces semantics and then the traces semantics itself. In Theorem 5.1, we show that they both describe the same behaviors.

Definition 5.13: 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.

$$tr \in \mathcal{T}_{cuc}(\sigma) \coloneqq \exists\, \sigma'.\ \sigma \overset{tr}{\Rightarrow}_{cuc} \sigma'$$

The (denotational) traces semantics differs in three important aspects from the operational semantics: 1) It allows us to determine first the semantics of single components and then combine them to a concurrent semantics, making it concurrently compositional. 2) It also allows us to compose the semantics of parts of a program, making it sequentially compositional. 3) It captures non-terminating behaviors, by considering all finite (partial) behaviors.

The traces semantics is depicted in Definition 5.12. Similarly to the operational semantics, it contains the same three kinds of rules, albeit in different order: 1) The local single steps ($\mathcal{T}$-DO, $\mathcal{T}$-CBR, $\mathcal{T}$-COMM), 2) the (local) semantics for multiple steps ($\mathcal{T}$-SEQ, $\mathcal{T}$-EXT),

Definition 5.12: Traces Semantics of CUC

$\mathcal{T}$-DO

$$[\![\ell, \texttt{do}\ f]\!]^{\mathcal{T}}(S) := S \cup \big\{(tr, \sigma') \big| (tr, \sigma) \in S \wedge \sigma_{pc} = \ell \wedge \sigma'_{ds} \in f(\sigma_{ds}) \wedge \sigma'_{pc} = \sigma_{pc} + 1\big\}$$

$\mathcal{T}$-CBR

$$[\![\ell, \texttt{cbr}\ b\ m\ n]\!]^{\mathcal{T}}(S) := S \cup \Big\{(tr, \sigma') \Big| (tr, \sigma) \in S \wedge \sigma_{pc} = \ell \wedge \sigma'_{ds} = \sigma_{ds} \wedge \big(b\ \sigma \wedge \sigma'_{pc} = m \vee \neg b\ \sigma \wedge \sigma'_{pc} = n\big)\Big\}$$

$\mathcal{T}$-COMM

$$[\![\ell, \texttt{comm}\ f_{ev}\ f_{ds}]\!]^{\mathcal{T}}(S) := S \cup \big\{(tr', \sigma') \big| (tr, \sigma) \in S \wedge \sigma_{pc} = \ell \wedge ev \in f_{ev}(\sigma_{ds}) \wedge tr' = tr \frown \langle ev \rangle \wedge \sigma'_{ds} = f_{ds}(\sigma_{ds}, ev) \wedge \sigma'_{pc} = \sigma_{pc} + 1\big\}$$

$\mathcal{T}$-SEQ

$$[\![sp_1 \oplus sp_2]\!]^{\mathcal{T}} := \big(\mu d.\ extend(sp_1, sp_2)\ (d)\big)$$

$\mathcal{T}$-EXT

$$extend(sp_1, sp_2)\ (d) := \lambda S.\ S \cup d\big([\![sp_1]\!]^{\mathcal{T}}(S)\big) \cup d\big([\![sp_2]\!]^{\mathcal{T}}(S)\big)$$

$\mathcal{T}$-PAR

$$[\![scp_1\ {}_{\alpha_1}\|_{\alpha_2}\ scp_2]\!]^{\mathcal{T}}(S) = \big\{(tr, \sigma'_1 \parallel \sigma'_2) \mid (\langle\rangle, \sigma_1 \parallel \sigma_2) \in S \wedge (tr \upharpoonright \alpha_1, \sigma'_1) \in [\![scp_1]\!]^{\mathcal{T}}\big(\{(\langle\rangle, \sigma_1)\}\big) \wedge (tr \upharpoonright \alpha_2, \sigma'_2) \in [\![scp_2]\!]^{\mathcal{T}}\big(\{(\langle\rangle, \sigma_2)\}\big) \wedge set(tr) \subseteq (\alpha_1 \cup \alpha_2)\big\}$$

and 3) the concurrent composition ($\mathcal{T}$-PAR). It is defined for programs in *SP* and *SCP*, respectively.

The idea of the traces semantics is to describe all behaviors of a CUC program at once, both in terms of partial executions of possibly non-terminating executions and in terms of alternative behaviors due to different interactions between components or non-determinism. Similarly to CSP, to account for the multitude of behaviors, we express the traces semantics as a set of pairs of traces and states. We require a state with every trace, as CUC is stateful in contrast to the stateless CSP. The intuition of the pair is that the trace leads to the state.

The traces semantics is defined as a semantic function $[\![\cdot]\!]^{\mathcal{T}}$ that maps a structured (concurrent) CUC program to its *denotation*. The denotation is a function mapping from a set of trace-state pairs to a set of trace-state pairs. Intuitively, $[\![\cdot]\!]^{\mathcal{T}}$ assigns every program a function that takes a set of trace-state pairs and computes (the reflexive, transitive hull of) the successor pairs.

$$[\![\cdot]\!]^{\mathcal{T}} \colon SCP \to \big(\mathscr{P}(\Sigma^* \times States) \to \mathscr{P}(\Sigma^* \times States)\big)$$

Requiring sets also as inputs to the denotations allows us to chain denotations directly and facilitates an elegant formulation of the sequential composition ($\mathcal{T}$-EXT). For compatibility with CSP in Section 5.3.5, we require the inclusion of the empty trace and *prefix closure* of traces. To this end, we assume that the traces of all initial trace-state pairs are empty, i. e., when we look at the semantics of a whole program, we require the initial traces to be empty.

Assumption 5.5: Empty Initial Traces

When considering the denotational semantics of an entire program, we assume that all traces in the input set are empty.

$$\forall\, s \in S_{init}.\ \exists\, \sigma.\, s = (\langle\rangle, \sigma)$$

The semantics of the (local) single step rules $\mathcal{T}$-DO, $\mathcal{T}$-CBR, and $\mathcal{T}$-COMM (cf. Definition 5.12) are very similar to their operational semantics counterparts. For all states where the *pc* points to the label ℓ of the respective instruction, all successor states are computed from the respective instruction as defined in the operational semantics and are added to the resulting set. In $\mathcal{T}$-COMM, the trace is extended with the communicated event.

The semantics of the sequential composition ($\mathcal{T}$-SEQ and $\mathcal{T}$-EXT in Definition 5.12) are different from the execution semantics. We use (parts of) a structured program instead of the whole program as a labeled instruction set. The function *extend*, defined in $\mathcal{T}$-EXT, extends a given denotation d with the executions of the two components sp_1 and sp_2 respectively. Recall that a denotation is a function mapping input trace-state pairs to output trace-state pairs. Thus, a denotation is of the same type as, e. g., $[\![\ell : \texttt{do}\ f]\!]^{\mathcal{T}}$. With the help of *extend*, we describe in $\mathcal{T}$-SEQ the sequential composition $\oplus$ which is modeled as fixpoint of *extend* and thus as the repeated, possibly alternating application of the two components sp_1 and sp_2. We use the μ operator to denote the *least fixpoint* of a denotation. The underlying

chain-complete partial order is the point-wise subset relation. Note that $\oplus$ is also the looping construct as every part of code can contain branch instructions.

Example 5.10: $\mathcal{T}$-SEQ and $\mathcal{T}$-EXT

We illustrate the rule $\mathcal{T}$-SEQ and $\mathcal{T}$-EXT with an example, which is a simplified version of sp'_{ex}. Consider the initial trace-state pair $(\langle\rangle, \sigma)$ with $\sigma_{pc} = 1$ and the program

$$\big(1, \texttt{comm}\ (\lambda ds.\ \{a\})\ (\lambda ds\ ev.\ \{ds\})\big) \oplus \big(2, \texttt{cbr}\ (\lambda ds.\ \mathit{true}\ 1\ 1)\big)$$

It is a non-terminating program communicating a repeatedly with its environment. According to $\mathcal{T}$-EXTEND, both instructions are evaluated separately, where initially `comm` modifies the set accordingly (e. g., append a to the trace) and `cbr` does nothing, as σ_{pc} does not point to it.

$$\begin{aligned}
[\![1, \texttt{comm}...]\!]^{\mathcal{T}}\big(\{(\langle\rangle, \sigma)\}\big) &= \{(\langle\rangle, \sigma), (\langle a\rangle, \sigma[pc := 2])\} \\
[\![2, \texttt{cbr}...]\!]^{\mathcal{T}}\big(\{(\langle\rangle, \sigma)\}\big) &= \{(\langle\rangle, \sigma)\}
\end{aligned}$$

In the next iteration of the fixpoint iteration, both instructions are again executed. This time `comm` does nothing (new) but `cbr` will now generate trace-state pairs whose program counter points to `comm`, so in the next iteration the loop will be executed from the beginning.

$$\begin{aligned}
[\![1, \texttt{comm}...]\!]^{\mathcal{T}}\big(\{(\langle\rangle, \sigma), (\langle a\rangle, \sigma[pc := 2])\}\big) &= \{(\langle\rangle, \sigma), (\langle a\rangle, \sigma[pc := 2])\} \\
[\![2, \texttt{cbr}...]\!]^{\mathcal{T}}\big(\{(\langle\rangle, \sigma), (\langle a\rangle, \sigma[pc := 2])\}\big) &= \{(\langle\rangle, \sigma), (\langle a\rangle, \sigma[pc := 2]), (\langle a\rangle, \sigma)\}
\end{aligned}$$

As a global fixpoint we get the following set of trace-state pairs, where $\langle a\rangle^*$ denotes the trace of 0 or more repetitions of the event a.

$$[\![(1, \texttt{comm}...) \oplus (2, \texttt{cbr}...)]\!]^{\mathcal{T}}\big(\{(\langle\rangle, \sigma)\}\big) = \{(\langle a\rangle^*, \sigma), (\langle a\rangle^*, \sigma[pc := 2])\}$$

We define the concurrent semantics as close a possible to the concurrent CSP semantics. The purpose is to inherit the (concurrent) compositionality of the parallel composition of CSP and thus the compositionality of its refinement relation. This enables us to refine each component separately. It is important to notice that we only define *top-level* parallel composition. So, components can be composed in parallel, but may themselves not contain parallel components. Remember that the nesting structure of a concurrent state matches the nesting structure of a parallel program (Assumption 5.3). We choose *alphabetized parallel* as the most general parallel combination in CSP that can be used to represent the other two, namely interface parallel and interleaving. We closely follow the CSP definition of alphabetized parallel and adjust it to CUC by adding the states. As we only allow top-level composition, we consider only entire local programs and thus assume initial traces to be empty (Assumption 5.5). In the same way as in CSP, the concurrent composition of two components considers all interleavings of the traces of the components, synchronizing on the given alphabets.

Having defined the denotational traces semantics, we show its correspondence to the

operational characterization. When defining a denotational semantics, it is important to relate it to the operational characterization to show its adequacy. We show correspondence between the denotational traces semantics and its operational characterization by showing that the traces semantics is *sound* and *complete* with respect to the operational characterization. Informally, we show that, starting in an arbitrary state, the traces observable through the execution semantics and the traces semantics of a given unstructured program and a structured version of it are the same:

Theorem 5.1: Correspondence Between Operational Characterization and Traces Semantics

$$tr \in \mathcal{T}_{\mathcal{U}(scp)}(\sigma) \Leftrightarrow \exists\, \sigma'.\ (tr, \sigma') \in [\![scp]\!]^{\mathcal{T}}\big(\{(\langle\rangle, \sigma)\}\big)$$ *

The proof for the concurrent case can be found in Appendix A.1.1. From the correspondence we can directly conclude that the specific structure we chose for a program does not change its semantics. This also implies commutativity and associativity of $\oplus$.

Corollary 5.1: Invariance Under Structure

$$\mathcal{U}(sp_1) = \mathcal{U}(sp_2) \Longrightarrow [\![sp_1]\!]^{\mathcal{T}}(S) = [\![sp_2]\!]^{\mathcal{T}}(S)$$

The traces semantics defined in this subsection allows us to use the traces refinement of CSP, which preserves safety properties. In contrast to the operational semantics, it is compositional, both sequentially and concurrently. The safety properties ensure that "nothing bad" happens. However, doing nothing ensures doing "nothing bad", and, thus, preserves all safety properties. This is not what we expect from an implementation. We expect that "something good" happens, which is captured by liveness properties. Therefore, we require also that liveness properties are preserved. Especially reactive systems are required to react to user inputs or environment changes. In the next subsection, we define a stable failures semantics. The corresponding stable failures refinement preserves also liveness properties.

5.3.4 Stable Failures Semantics

In this subsection, we define the stable failures semantics for CUC. The main advantage of using the stable failures semantics is that it captures information about the communication capabilities in addition to the observed behavior. Recording communication capabilities allows for the formulation of liveness properties. As internal transitions could change the offered communication, we consider only states where internal transitions are not possible. Hence, they are called *stable*. To enable the use of a subset relation as refinement relation, we do not record which events are offered, but which events *cannot* be offered. Hence, they are called *failures*.

Our definition of the stable failures semantics of CUC is similar to the definition of the traces semantics of CUC. The three important differences are *1)* that we add distinct states where the component is ready to communicate, *2)* that we include refusal sets to capture which communication events can be refused, and *3)* that we only consider stable states for

semantics. First, we motivate and define the necessary notions, and then we define the stable failures semantics for CUC.

A *failure* is a triplet (tr, σ, X) consisting of a trace tr leading to a state σ where the events in X can be refused. We explain and define the stable failures of CUC in the following. As in CSP, we only include *stable* failures. A failure is stable, if it does not change unnoticed, thus, if it cannot perform an internal action τ. Let *cuc* be a CUC program.

Definition 5.14: CSP-like Stable States of CUC

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

Refusal sets are defined similarly to their CSP counterparts.

Definition 5.15: Refusal Set of CUC

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

$$\sigma \text{ ref}_{cuc}\ X := \forall\, a \in X.\ \nexists\, \sigma'.\ \sigma \xrightarrow{a}_{cuc} \sigma'$$

Having defined stable states and refusal sets, we use them to define the operational characterization of the stable failures of a CUC program *cuc*.

Definition 5.16: Operational Characterization of the Stable Failures of CUC

A **stable failure** of a CUC program *cuc* is a pair of a trace tr and a refusal set X. It denotes that there is a stable state σ' which can be reached from the initial state σ via the trace tr and refuses X.

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

We distinguish between two types of stable failures: Terminal failures, where the program has ended, and communication failures, where the program is ready to communicate. However, we cannot distinguish those two settings in the traces semantics of CUC, as we cannot distinguish between "the *pc* points to `comm`" and "`comm` is ready to communicate and offers certain communication events". To facilitate the distinction, we introduce special *communication states* (*CStates*). They have the same type (*LStates*) as the previously defined local states, which we call now *normal states* for distinction. To distinguish the types of normal states and communication states we extend the type of the normal states with "$\times\{c\}$" to obtain communication states. We define the sum-type of normal and communication states (*NCStates*) as well as its concurrent composition (*CNCStates*) as usual. Let σ be a (possibly concurrent) normal or communication state.

Definition 5.17: Communication States

$$\begin{aligned} \mathit{CStates} &\coloneqq (\mathit{Labels} \times \mathit{DS}) \times \{c\} \\ \mathit{NCStates} &\coloneqq \mathit{LStates} \mid \mathit{CStates} \\ \mathit{CNCStates} &\coloneqq \mathit{NCStates} \mid \mathit{CNCStates} \parallel \mathit{CNCStates} \end{aligned}$$

In Definition 5.18, we introduce a predicate $N(\cdot)$ to test if a local state is normal, and a function $\cdot^C$ which converts a normal state to a communication state.

Definition 5.18: Test for Normal State and Conversions to Communication State, $N(\cdot)$, $\cdot^C$

$$\begin{aligned} N(\sigma) &\coloneqq \sigma \in \mathit{LStates} \\ \cdot^C &: \mathit{LStates} \to \mathit{CStates} \\ \sigma^C &\coloneqq (\sigma, c) \end{aligned}$$

Let a CUC failure be a triple consisting of a trace, a state, and a refusal set. We can now define $[\![\cdot]\!]^{\mathcal{SF}}$, which maps structured (concurrent) CUC programs to functions from sets of failures to sets of failures. Intuitively, $[\![\cdot]\!]^{\mathcal{SF}}$ assigns every program a function that takes a set of failures and computes (the transitive hull of) the successor failures.

$$[\![\cdot]\!]^{\mathcal{SF}} : \mathit{SCP} \to \big(\mathscr{P}(\Sigma^* \times \mathit{CNCStates} \times \mathscr{P}(\Sigma)) \to \mathscr{P}(\Sigma^* \times \mathit{CNCStates} \times \mathscr{P}(\Sigma))\big)$$

To ensure compatibility to the CSP stable failures refinement (discussed in Subsection 5.3.5), we need to account for initial failures, whose state does not point into the program, and, thus, is not "processed" by the semantics. We require all initial states to be normal so that we do not introduce unrelated communication behavior. Furthermore, we require all initial refusals to be maximal and subset closed to ensure that also "unprocessed" failures comply to Properties SF2 and SF3 of stable failures semantics. Also, we require again all initial traces to be empty.

Assumption 5.6: Initial Failures

$$\forall\, s \in S_{init}.\ \exists\, \sigma\ X.\ s = (\langle\rangle, \sigma, X) \wedge N(\sigma) \wedge \big(\forall\, Y \subseteq \Sigma.\ (\langle\rangle, \sigma, Y) \in S_{init}\big)$$

When applying the sequential composition of the stable failures semantics, former terminal failures, which did not point into the program and now point into the program, are no longer stable and need to be removed. To this end, we define the following operator $\setminus_{(\cdot)}$ for sets of local failures. It will be used with the sets of labels of a (sub-)program as a parameter. We will give an example of its use when explaining the rule of the stable failures semantics (Example 5.11).

Definition 5.19: Removal of Former Terminal Failures, $\backslash_{(\cdot)}$

$$S\backslash_{labels} \coloneqq S \setminus \{(tr, \sigma, X) \mid \sigma_{pc} \in labels \wedge N(\sigma)\}$$

Definition 5.20: Stable Failures Semantics of CUC

$\mathcal{SF}$-DO

$$\begin{aligned}[\![\ell : \texttt{do}\ f]\!]^{\mathcal{SF}}(S) \quad &\coloneqq S\backslash_{\ell} \cup \big\{(tr, \sigma', X) \big|\ (tr, \sigma, \_) \in S \wedge \sigma_{pc} = \ell \wedge N(\sigma) \wedge \\ &\qquad \sigma'_{ds} \in f(\sigma_{ds}) \wedge \sigma'_{pc} = \ell + 1 \wedge X \subseteq \Sigma\big\}\end{aligned}$$

$\mathcal{SF}$-CBR

$$\begin{aligned}[\![\ell : \texttt{cbr}\ b\ m\ n]\!]^{\mathcal{SF}}(S) \ &\coloneqq S\backslash_{\ell} \cup \big\{(tr, \sigma', X) \big|\ (tr, \sigma, \_) \in S \wedge \sigma_{pc} = \ell \wedge N(\sigma) \wedge \\ &\qquad \big(b(\sigma_{ds}) \wedge \sigma'_{pc} = m\ \vee \neg b(\sigma_{ds}) \wedge \sigma'_{pc} = n\big) \wedge \\ &\qquad \sigma_{ds} = \sigma'_{ds} \wedge X \subseteq \Sigma\big\}\end{aligned}$$

$\mathcal{SF}$-COMM

$$\begin{aligned}[\![\ell : \texttt{comm}\ f_{ev}\ f_{ds}]\!]^{\mathcal{SF}}(S) \coloneqq\ & S\backslash_{\ell} \cup \big\{(tr, \sigma^C, X) \big| (tr, \sigma, \_) \in S \wedge \sigma_{pc} = \ell \wedge N(\sigma) \wedge \\ &\qquad X \subseteq \Sigma \setminus f_{ev}(\sigma_{ds})\big\} \\ &\cup \big\{(tr', \sigma', X) \big| (tr, \sigma, \_) \in S \wedge \sigma_{pc} = \ell \wedge N(\sigma) \wedge \\ &\qquad ev \in f_{ev}\big(\sigma_{ds}\big) \wedge tr' = tr^{\frown}\langle ev\rangle \wedge \\ &\qquad \sigma'_{ds} = f_{ds}\big(\sigma_{ds}, ev\big) \wedge \sigma'_{pc} = \ell + 1\ \wedge X \subseteq \Sigma\big\}\end{aligned}$$

$\mathcal{SF}$-SEQ

$$[\![sp_1 \oplus sp_2]\!]^{\mathcal{SF}}(S) \coloneqq \Big(\big(\mu d.\ extend(sp_1, sp_2)\ (d)\big)(S)\Big)\backslash_{labels(sp_1 \oplus sp_2)}$$

$\mathcal{SF}$-EXT

$$extend(sp_1, sp_2)\ (d) \coloneqq \lambda S.\ S \cup d\big([\![sp_1]\!]^{\mathcal{SF}}(S)\big) \cup d\big([\![sp_2]\!]^{\mathcal{SF}}(S)\big)$$

$\mathcal{SF}$-PAR

$$\begin{aligned}[\![scp_1\ {}_{\alpha_1}\|_{\alpha_2}\ scp_2]\!]^{\mathcal{SF}}(S) = \big\{(tr, \sigma'_1 \parallel \sigma'_2, X) \ \big|\ &\exists X_1\ X_2.\ (\langle\rangle, \sigma_1 \parallel \sigma_2, \_) \in S \wedge \\ &X \cap (\alpha_1 \cup \alpha_2) = (X_1 \cap \alpha_1) \cup (X_2 \cap \alpha_2) \wedge \\ &(tr \upharpoonright\ \alpha_1, \sigma'_1, X_1) \in [\![scp_1]\!]^{\mathcal{SF}}\big(\{(\langle\rangle, \sigma_1, \_)\}\big) \wedge \\ &(tr \upharpoonright\ \alpha_2, \sigma'_2, X_2) \in [\![scp_2]\!]^{\mathcal{SF}}\big(\{(\langle\rangle, \sigma_2, \_)\}\big) \wedge \\ &set(tr) \subseteq (\alpha_1 \cup \alpha_2)\big\}\end{aligned}$$

The $\mathcal{SF}$ semantics is depicted in Definition 5.20. Similar to the traces semantics, it contains 1) the local single steps ($\mathcal{SF}$-DO, $\mathcal{SF}$-CBR, $\mathcal{SF}$-COMM), 2) the semantics for multiple local steps ($\mathcal{SF}$-SEQ, $\mathcal{SF}$-EXT), and 3) the concurrent composition ($\mathcal{SF}$-PAR). As $\mathcal{T}$, it is defined for programs in SP and SCP, respectively.

The semantics of the local single step rules $\mathcal{SF}$-DO and $\mathcal{SF}$-CBR (cf. Definition 5.20), are similar to $\mathcal{T}$-DO and $\mathcal{T}$-CBR (cf. Definition 5.12). The differences are that the former terminal failures of the input set are removed, only successors to normal states are included, and all possible refusal sets are added.

Example 5.11: $\mathcal{T}$-DO and $\backslash_{(\cdot)}$

Consider the initial input set $\{(\langle\rangle, \sigma, X), (\langle\rangle, \sigma', Y)\}$, where X and Y are arbitrary refusal sets, $\sigma_{pc} = 5$, $\sigma'_{pc} = 1$, both normal, and the instruction $(1, \texttt{do}\ (\lambda ds.\ \{ds\}))$, which does nothing but increment the program counter. As the program counter of σ is not pointing to this instruction, the failure is still terminal. Thus, the failure $(\langle\rangle, \sigma, X)$ is not removed. Furthermore, there are no successor failures of $(\langle\rangle, \sigma, X)$. The program counter σ'_{pc} points to the instruction, so there are successor failures $\{(\langle\rangle, \sigma'[pc := 2], Z) \mid Z \subseteq \Sigma\}$. As the initial failure $(\langle\rangle, \sigma', Y)$ points into the program, it is not terminal anymore, thus, no longer stable and needs to be removed. The resulting failures are:

$$
\begin{aligned}
&[\![1, \texttt{do}\ ;\ \lambda ds.\ \{ds\}]\!]^{\mathcal{SF}}\big(\{(\langle\rangle, \sigma, X), (\langle\rangle, \sigma', Y)\}\big) \\
&= \{(\langle\rangle, \sigma, X), (\langle\rangle, \sigma', Y)\} \backslash_{\{1\}} \cup \{(\langle\rangle, \sigma'[pc := 2], Z) \mid Z \subseteq \Sigma\} \\
&= \{(\langle\rangle, \sigma, X)\} \qquad\qquad\qquad\quad \cup \{(\langle\rangle, \sigma'[pc := 2], Z) \mid Z \subseteq \Sigma\}
\end{aligned}
$$

The rule $\mathcal{SF}$-COMM in Definition 5.20 is the most different from its $\mathcal{T}$ counterpart. It adds two types of failures instead of one: The communication failures and the new terminal failures. The new terminal failures are similar to those added in $\mathcal{SF}$-DO and $\mathcal{SF}$-CBR, i. e., with a possibly updated data store and an incremented program counter. However, the communicated event is appended to the trace. The communication failures have a communication state instead of a normal state and only add refusal sets that do not include events offered by f_{ev}. The communication failures only serve the purpose to keep track of the communication possibilities during the execution of the `comm` instruction. It is never removed by a subsequent application of the semantics (cf. Definition 5.19 of $\backslash_{(\cdot)}$). Successor failures are only calculated based on the normal states. The normal states are also likely to be removed by a subsequent application of the semantics.

The semantics of the sequential composition ($\mathcal{SF}$-SEQ and $\mathcal{SF}$-EXT in Definition 5.20) only differ in the removal of the former terminal failures *after* the application of the fix point operator in $\mathcal{SF}$-SEQ.

Example 5.12: $\mathcal{SF}$-COMM, $\mathcal{SF}$-SEQ and $\mathcal{SF}$-EXT

We illustrate the rules $\mathcal{SF}$-COMM, $\mathcal{SF}$-SEQ and $\mathcal{SF}$-EXT with an example. Consider the initial failure $(\langle\rangle, \sigma, X)$ with $\sigma_{pc} = 1$, X arbitrary and the same program as in Example 5.10

$$\big(1, \texttt{comm}\ (\lambda ds.\ \{a\})\ (\lambda ds\ ev.\ \{ds\})\big) \oplus \big(2, \texttt{cbr}\ (\lambda ds.\ \mathit{true}\ 1\ 1)\big)$$

According to $\mathcal{SF}$-EXTEND, both instructions are evaluated separately, where initially `comm` modifies the set accordingly (e. g., append the event a to the trace and add all appropriate refusal sets). In contrast to Example 5.10, the stable failures semantics does not retain the initial failures, as they are no longer terminal. They are removed by $\setminus_{\{1\}}$. Also, the communication failure is added. The instruction `cbr` does nothing, as σ_{pc} does not point to it.

$$\begin{aligned}
[\![1, \texttt{comm}...]\!]^{\mathcal{SF}}\big(\{(\langle\rangle, \sigma, X)\}\big) &= \big\{(\langle\rangle, \sigma^C, Y) \mid Y \subseteq \Sigma \setminus \{a\}\big\} \cup \big\{(\langle a\rangle, \sigma[pc := 2], Y) \mid Y \subseteq \Sigma\big\} \\
[\![2, \texttt{cbr}...]\!]^{\mathcal{SF}}\big(\{(\langle\rangle, \sigma, X)\}\big) &= \big\{(\langle\rangle, \sigma, X)\big\}
\end{aligned}$$

In the next iteration of the fixpoint iteration, both instructions are again executed, with combined initial failures. This time, the instruction `comm` does nothing (new) but `cbr` will now generate trace-state pairs whose program counter points to `comm`, so that in the next iteration the loop will be executed from the beginning with a longer trace. Both instructions remove former terminal failures with $\setminus_{(\cdot)}$.

$$\begin{aligned}
[\![1, \texttt{comm}...]\!]^{\mathcal{SF}}\Big(\big\{(\langle\rangle, \sigma, X)\big\} \cup \big\{(\langle\rangle, \sigma^C, Y) \mid Y \subseteq \Sigma \setminus \{a\}\big\} \cup \big\{(\langle a\rangle, \sigma[pc := 2], Y) \mid Y \subseteq \Sigma\big\}\Big) \\
= \quad \big\{(\langle\rangle, \sigma^C, Y) \mid Y \subseteq \Sigma \setminus \{a\}\big\} \cup \big\{(\langle a\rangle, \sigma[pc := 2], Y) \mid Y \subseteq \Sigma\big\} \\
[\![2, \texttt{cbr}...]\!]^{\mathcal{SF}}\Big(\big\{(\langle\rangle, \sigma, X)\big\} \cup \big\{(\langle\rangle, \sigma^C, Y) \mid Y \subseteq \Sigma \setminus \{a\}\big\} \cup \big\{(\langle a\rangle, \sigma[pc := 2], Y) \mid Y \subseteq \Sigma\big\}\Big) \\
= \ \big\{(\langle\rangle, \sigma, X)\big\} \cup \big\{(\langle\rangle, \sigma^C, Y) \mid Y \subseteq \Sigma \setminus \{a\}\big\} \cup \big\{(\langle a\rangle, \sigma, Y) \mid Y \subseteq \Sigma\big\}
\end{aligned}$$

As a global fixpoint we get the following set of trace-state pairs. The stable failures semantics contains only terminal failures and communication failures. As this program does not terminate, there are no terminal failures in its semantics, only communication failures. The former additional failures, which are retained by the "$S \cup$" part in $\mathcal{SF}$-EXTEND are removed by the $\setminus_{\{1,2\}}$ in $\mathcal{SF}$-SEQ.

$$[\![(1, \texttt{comm}...) \oplus (2, \texttt{cbr}...)]\!]^{\mathcal{SF}}\big(\{(\langle\rangle, \sigma, X)\}\big) = \big\{(\langle a\rangle^*, \sigma^C, Y) \mid Y \subseteq \Sigma \setminus \{a\}\big\}$$

We modeled the rules for the concurrent semantics ($\mathcal{SF}$-PAR in Definition 5.20) again closely after their CSP counterpart. If one of the multiple (concurrent) states can refuse an event of its communication interface, then the combined states can refuse it, too. Again, as we only allow top-level concurrency, we only consider entire local programs and thus only consider empty traces of the initial failures (Assumption 5.6).

Similar to the correspondence of the traces semantics, we can also show that the denotational stable failures semantics and its operational characterization describe the same

behaviors.

Theorem 5.2: Correspondence Between Operational Characterization and Stable Failures Semantics *

$$(tr, X) \in \mathcal{SF}_{\mathcal{U}(scp)}(\sigma) \Leftrightarrow \exists\, \sigma'.\ (tr, \sigma', X) \in [\![scp]\!]^{\mathcal{SF}}\big(\{(\langle\rangle, \sigma, Y) \mid Y \subseteq \Sigma\}\big)$$

The proof can be found in Appendix A.1.2.

In this subsection, we have defined the stable failures semantics. In contrast to the traces semantics of the last section, the stable failures semantics allows for the formulation of liveness properties. In the next subsection, we show that the traces and stable failures semantics are compatible to their CSP counterparts.

5.3.5 Compatibility to CSP

In this section, we show that our CUC semantics enjoys the fundamental properties of the denotational CSP semantics. This allows us to show that CUC is compatible with CSP, which finally allows us to prove a stable failures refinement between a CUC implementation and its CSP specification. As mentioned in the CSP section, the refinement relation is basically a subset relation. However, not every subset of behaviors is coherent and corresponds to a process or program. The compared sets of behaviors themselves (the semantics) need to fulfill certain properties. We introduce and explain adapted versions for CUC of the properties of the CSP denotational semantics (see Section 2.3.5). The adaption are due to the different semantics of CSP and CUC and consist mostly of taking into account programs and states, which are not present in the CSP semantics. For each of the following properties, we require that it holds for the initial set of trace-state pairs or failures, respectively (Assumptions 5.5 and 5.6). The proofs can be found in Appendix A.2. The first two properties (T1 and T2) require the empty behavior and all partial behaviors to be observable by the traces semantics. The latter four properties (SF1 to SF4) require the stable failures semantics to comply with the traces semantics and to contain all refusal sets that are possible for a given program.

Property T1

$$\langle\rangle \in \mathcal{T}_{cuc}(\sigma)$$

The empty trace must always be contained. Any program can be observed to do nothing.

Property T2

$$\forall\, tr\ tr'.\ tr' \leq tr \wedge tr \in \mathcal{T}_{cuc}(\sigma) \Longrightarrow tr' \in \mathcal{T}_{cuc}(\sigma)$$

$\mathcal{T}$ is prefix closed. All partial behaviors can be observed.

Property SF1

$$(tr, X) \in \mathcal{SF}_{cuc}(\sigma) \Longrightarrow tr \in \mathcal{T}_{cuc}(\sigma)$$

All trace-state pairs are included in the traces semantics. This property ensures that the stable failures semantics "acts" within the boundaries of the traces semantics.

Property SF2

$$(tr, X) \in \mathcal{SF}_{cuc}(\sigma) \wedge X' \subseteq X \Longrightarrow (tr, X') \in \mathcal{SF}_{cuc}(\sigma)$$

Refusal sets are subset closed.

Property SF3

$$(tr, X) \in \mathcal{SF}_{cuc}(\sigma) \wedge \forall\, a \in X'.\ tr^\frown\langle a\rangle \notin \mathcal{T}_{cuc}(\sigma) \Longrightarrow (tr, X \cup X') \in \mathcal{SF}_{cuc}(\sigma)$$

All events that can be refused occur in a refusal set. More specifically, the refusal sets can be augmented with refused events. This is the crucial property ensuring that the stable failures are "enough" refusals to show liveness properties, because it ensures that all events that can be refused are contained in the stable failures semantics.

Property SF4

$$\sigma \overset{tr}{\Longrightarrow}_{cuc} \sigma' \wedge \sigma'_{pc} \notin_{pc} code \Longrightarrow \exists\, X.\ (tr, X) \in \mathcal{SF}_{cuc}(\sigma)$$

Terminal failures are stable.
Although the equivalence $\exists\, \sigma'.\, \sigma \overset{tr}{\Longrightarrow}_{cuc} \sigma' \Leftrightarrow tr \in \mathcal{T}_{cuc}(\sigma)$ holds, we choose not to use $\mathcal{T}$ as in the CSP version of SF4, as we need to talk about the reached state σ' explicitly to model terminal behavior.

The fact that these properties hold for the denotational semantics of CUC allow us to extend the refinement notion from CSP to CUC, both for the traces and for the stable failures model. Additionally, the Properties SF1 and SF4 relate the traces and stable failures semantics of CUC.

In this subsection, we have shown that the traces and the stable failures semantics of CUC share the fundamental properties of the denotational semantics of CSP. This allows us to extend the notion of refinement from only relating traces or stable failures of CSP to relating traces or stable failures, respectively, with those of CUC. The extended notion of CSP refinement to include also CUC is the basis for our workflow to relate CSP specifications and CUC programs in Section 5.5.

In this section, we have defined three different semantics for CUC: 1) The operational semantics to define the intuition of a CUC program, 2) the traces semantics to describe all

behaviors of a CUC program in a compositional way, and 3) the stable failures semantics to additionally describe future communication capabilities. We have shown the correspondence between the denotational semantics (traces and stable failures) and their operational characterizations. We have shown compatibility of the traces and stable failures semantics to their CSP counterparts, and with that also between the traces and stable failures semantics of CUC itself. The compatibility between the CSP and the CUC semantics allows us to extend the respective refinement notions of CSP to CUC.

In the next section, we define a Hoare calculus for the stable failures semantics of CUC, which enables to show refinements between CSP specifications and CUC programs.

5.4 Hoare Calculus

To facilitate compositional reasoning about (non-concurrent) structured programs, we introduce a Hoare calculus for our stable failures semantics of CUC. We use the Hoare calculus in the next section to show properties for the stable failures of a CUC program. This enables us to relate them to the stable failures of a CSP process in order to show a stable failures refinement. Therefore, we require our Hoare calculus to enable reasoning about non-terminating and communicating programs. As the notion of stable failures refinement is concurrently compositional, single components can be refined and the resulting composition is again a refinement. Therefore, we consider only the refinement of single components explicitly and define our Hoare calculus for (non-concurrent) structured programs.

Our assertions are predicates in higher order logic (HOL) on single CUC failures.

Definition 5.21: Hoare Triple for CUC

$$\{P\}\ sp\ \{Q\} \coloneqq \forall\, s.\ \big(P(s) \longrightarrow \forall\, s' \in [\![sp]\!]^{\mathcal{SF}}(\{s\}).\ Q(s')\big)$$

Starting with the precondition P, a structured CUC program sp satisfies the postcondition Q, if all stable failures in the stable failures semantics of sp satisfy Q, when the initial failures satisfy P.

Observe that our semantics yields a set that includes *all intermediate* stable failures, i. e., all stable failures that occur during the execution of a program. Thus, postconditions in our Hoare calculus are also *invariants for communication failures*. We still can construct usual postconditions Q' which hold only after terminated executions, e. g., by setting

$$Q(tr, \sigma, X) \coloneqq \sigma_{pc} \notin_{pc} sp \longrightarrow Q'(tr, \sigma, X)$$

We present our Hoare calculus for stable failures of CUC in Definition 5.22. It contains three kinds of rules, of which the first two match the kinds of the $\mathcal{SF}$ semantics: 1) Single steps (H-DO, H-CBR, H-COMM), 2) sequential composition (H-SEQ), and 3) the rule of consequence (H-CONS).

Definition 5.22: Hoare Calculus for CUC

H-DO

$$\frac{\begin{aligned} P(tr,\sigma,X) \equiv \neg(N(\sigma) \wedge \sigma_{pc} = \ell) &\longrightarrow Q(tr,\sigma,X) \wedge \\ N(\sigma) \wedge \sigma_{pc} = \ell \ &\longrightarrow \big(\forall \sigma'.\ N(\sigma') \wedge \sigma'_{pc} = \ell + 1 \wedge \sigma'_{ds} \in f(\sigma_{ds}) \\ &\qquad \longrightarrow \forall Y \subseteq \Sigma.\ Q(tr,\sigma',Y)\big) \end{aligned}}{\{P\}\ \ell : \texttt{do}\ \ f\ \{Q\}}$$

H-CBR

$$\frac{\begin{aligned} P(tr,\sigma,X) \equiv \neg(N(\sigma) \wedge \sigma_{pc} = \ell) &\longrightarrow Q(tr,\sigma,X) \wedge \\ N(\sigma) \wedge \sigma_{pc} = \ell \ &\longrightarrow \big(\forall \sigma'.\ (b\ \sigma_{ds} \wedge \sigma'_{pc} = m \vee \neg(b\ \sigma_{ds}) \wedge \sigma'_{pc} = n) \wedge \\ &\qquad N(\sigma') \wedge \sigma_{ds} = \sigma'_{ds} \longrightarrow \forall Y \subseteq \Sigma.\ Q(tr,\sigma',Y)\big) \end{aligned}}{\{P\}\ \ell : \texttt{cbr}\ \ b\ m\ n\ \{Q\}}$$

H-COMM

$$\frac{\begin{aligned} P(tr,\sigma,X) \equiv \neg(N(\sigma) \wedge \sigma_{pc} = \ell) &\longrightarrow Q(tr,\sigma,X) \wedge \\ N(\sigma) \wedge \sigma_{pc} = \ell \ &\longrightarrow \Big(\big(\forall Y \subseteq \Sigma \setminus f_{ev}(\sigma_{ds}).\ Q(tr,\sigma^C,Y)\big) \wedge \\ &\qquad \Big(\forall \sigma'.\ N(\sigma') \wedge \sigma'_{pc} = \ell + 1 \wedge \\ &\qquad\qquad ev \in f_{ev}(\sigma_{ds}) \wedge \sigma'_{ds} = f_{ds}(\sigma_{ds}, ev) \\ &\qquad\qquad \longrightarrow \big(\forall Y \subseteq \Sigma.\ Q(tr^\frown\langle ev\rangle,\sigma',Y)\big)\Big)\Big) \end{aligned}}{\{P\}\ \ell : \texttt{comm}\ \ f_{ev}\ f_{ds}\ \{Q\}}$$

H-SEQ

$$\frac{\begin{array}{c} \{\lambda(tr,\sigma,X).\ I(tr,\sigma,X) \wedge \sigma_{pc} \in_{pc} sp_1 \wedge N(\sigma)\}\ \ sp_1\ \ \{I\} \\ \{\lambda(tr,\sigma,X).\ I(tr,\sigma,X) \wedge \sigma_{pc} \in_{pc} sp_2 \wedge N(\sigma)\}\ \ sp_2\ \ \{I\} \end{array}}{\{I\}\ \ sp_1 \oplus sp_2\ \ \{\lambda(tr,\sigma,X).\ I(tr,\sigma,X) \wedge (N(\sigma) \longrightarrow \sigma_{pc} \notin_{pc} sp_1 \oplus sp_2)\}}$$

H-CONS

$$\frac{P \longrightarrow P' \qquad \{P'\}\ \ sp\ \ \{Q'\} \qquad Q' \longrightarrow Q}{\{P\}\ \ sp\ \ \{Q\}}$$

The rules H-DO, H-CBR and H-COMM describe how pre- and postconditions can be connected for the basic instructions. We define the rules in the "weakest precondition" style, where the postcondition is general and the precondition is defined as a function of the postcondition, i. e., the weakest precondition that ensures the postcondition. In contrast to a traditional Hoare calculus for a high-level language, we do not know, if the considered instruction is actually to be executed, e. g., if the program counter does not point to the instruction. Therefore, the preconditions of all three rules consist of a case distinction capturing whether the state of the considered failure is normal ($N(\sigma)$) and points to the respective instruction ($\sigma_{pc} = \ell$) or not. If this is not the case, then the instruction will not affect the failure and the post condition is required to hold immediately. This is expressed by the first line in each rule:

$$\neg(N(\sigma) \wedge \sigma_{pc} = \ell) \longrightarrow Q(tr, \sigma, X)$$

Otherwise the post condition needs to hold for the successor-failures, which are calculated according to the stable failures semantics of CUC of the respective instruction (cf. Definition 5.20).

H-SEQ relates the pre- and postconditions for the sequential composition $\oplus$. As the sequential composition potentially introduces loops, H-SEQ is based on an invariant I as in [SU05]. "Hiding" our invariant in the postcondition allows us to reuse the sequential composition rule from [SU05] in spite of adding an invariant. The invariant I usually consists of disjunctions which explicitly reference the program counter. Thus, it describes an invariance property for each label. The invariant I in H-SEQ is restricted by different program counters at the different positions in the rule. We use λ-functions to restrict the invariant and define new assertions (which are functions from single CUC failures to true or false). In the assumptions of the rule, only the part relevant for the respective partial program needs to be assumed for the precondition of the Hoare triples. In the postcondition of the Hoare triple in the conclusion of the rule, the invariant is stripped of all possibly former terminal failures. This enables compositional reasoning without keeping track of all normal states all the time. For a partial program, the invariant only needs to specify the normal states of the entry and exit points.

H-CONS is the usual rule of consequence. It allows for strengthening of the precondition and weakening of the postcondition. Apart from its usual application within a proof, we use the weakening of the postcondition (which is also the invariant) in the next section to "forget" the states of a CUC program and relate its traces and refusals to those of a CSP specification.

All rules of the calculus are correct with respect to Definition 5.21 of the Hoare triple.

Theorem 5.3: Soundness of Our Hoare Calculus
Our Hoare calculus is sound with respect to Definition 5.21.

We have shown this by structural induction over the structure of an arbitrary CUC program. The correctness of the Hoare calculus with respect to Definition 5.21 corresponds to partial correctness for failures with normal states. However, for failures with communication states, the postcondition holds universally and can, thus, be used as invariant about trace-refusal pairs (CSP failures). This is important as this enables us to show properties for reactive systems, i. e., communicating, non-terminating systems.

The Hoare calculus we have presented in this section is sound and facilitates sequentially compositional reasoning. Although it keeps an invariant for the communication behavior, it does not require a huge global invariant of all normal states. In the next section, we show how our Hoare calculus can be applied to relate the stable failures of a CSP specification and a CUC program.

5.5 Relating CSP and CUC

In this section, we present a workflow for establishing the relation between a specification in CSP and a low-level implementation in CUC. We assume that a CSP specification *Spec* is given, as well as an implementation *Impl* thereof in CUC. To verify the preservation of liveness and safety properties from *Spec* to *Impl*, we show that $Spec \sqsubseteq_{\mathcal{SF}} Impl$ holds in the stable failures model.

Before presenting the workflow, we show that for CUC the notion of stable failures refinement implies the notion of traces refinement, if we assume divergence freedom of the CUC program (cf. Assumption 5.7). To this end, we show that every trace in the traces semantics of a CUC program is also captured by its stable failures semantics.

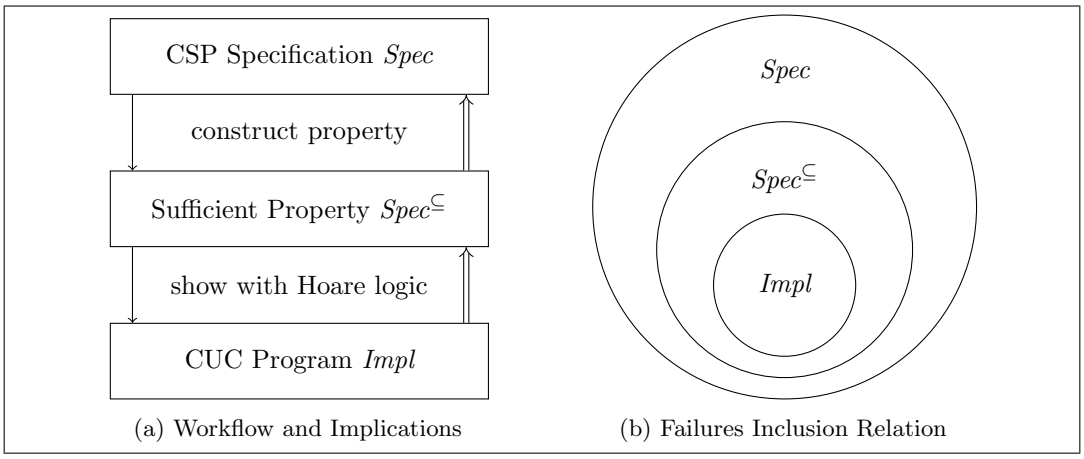
Lemma 5.1: Traces Imply Stable Failures in CUC

$$(tr, \_) \in \mathcal{T}_{cuc} \Longrightarrow \exists\, X.\ (tr, \_, X) \in \mathcal{SF}_{cuc}$$

Proof: Lemma 5.1 (Traces Imply Stable Failures in CUC)

In the traces semantics of CUC, only the `comm` instruction extends the trace. The `comm` instruction also creates stable failures (communication failures). Thus, traces are only extended directly after a stable state. If we assume divergence freedom, i. e., no infinite τ-loops, then every program either terminates at some point or communicates events forever. Thus, there is a stable failures after every extension of the trace (either a communication failure caused the next `comm` instruction or a terminal failure if the program ends). It follows that every trace in traces semantics of a CUC program is also captured by its stable failures semantics. □

Lemma 5.1 can be thought of as the inverse of Property SF1, i. e., ensuring that all behaviors captured by the traces semantics are also captured by the stable failures semantics. We can now show that the stable failures refinement from a CSP process *csp* to a CUC program *cuc* implies the traces refinement from *csp* to *cuc*.



(a) Workflow and Implications (b) Failures Inclusion Relation

Figure 5.1: Overview of the Workflow

Theorem 5.4: Traces Refinement Implies Stable Failures Refinement for CUC

$$csp \sqsubseteq_{\mathcal{SF}} cuc \Longrightarrow csp \sqsubseteq_{\mathcal{T}} cuc$$

Proof

Want to show: $csp \sqsubseteq_{\mathcal{T}} cuc :\Longleftrightarrow (tr, \_) \in \mathcal{T}_{cuc} \Longrightarrow tr \in \mathcal{T}_{csp}$

$$(tr, \_) \in \mathcal{T}_{cuc} \xRightarrow{\text{Lemma 5.1}} \exists X.\ (tr, \_, X) \in \mathcal{SF}_{cuc} \xRightarrow{\sqsubseteq_{\mathcal{SF}}} (tr, X) \in \mathcal{SF}_{csp} \xRightarrow{\text{SF1}} tr \in \mathcal{T}_{csp} \quad \square$$

Theorem 5.4 enables us to consider only the stable failures semantics to show a stable failures refinement, which usually requires us to also show a traces refinement. However, we need to assume divergence freedom of CUC programs. Divergence freedom can be shown by syntax analysis of CUC programs (every possible loop needs to contain a `comm` instruction). An alternative to the assumption of divergence freedom is to show the traces refinement in addition to the stable failures refinement. In this thesis, we assume divergence freedom. In future work, we envision a failures-divergences semantics for CUC (see Section 8.3).

Assumption 5.7: Divergence Freedom of CUC Programs

All considered CUC programs are assumed to be divergence free, i. e., do not have τ-loops.

Having established that we only need to consider the stable failures semantics to show a stable failures refinement, we proceed to present our workflow, which is depicted in Figure 5.1a. It is hard to establish a refinement between *Spec* and *Impl* directly, as they are structurally very different: CSP is structured and unstructured languages (such as CUC) are not. In structured languages, the control flow is visible in the syntactic structure. This is not the case for languages with unrestricted jumps. Furthermore, the internal computations of CUC are

not reflected in CSP. We use an intermediate *sufficient property* to connect the specification and the implementation. The workflow consists of three steps:

1) Manually constructing a *sufficient property* $Spec^{\subseteq}$ from *Spec*,

2) showing that $Spec^{\subseteq}$ is sufficient for *Spec*, and

3) showing that $Spec^{\subseteq}$ holds for *Impl*.

The property $Spec^{\subseteq}$ that is constructed from *Spec* in step *1)* is a predicate on CSP failures and needs to be at least as strong as *Spec*, which is shown in step *2)*. It is formally captured by

$$Spec^{\subseteq}(tr, X) \Longrightarrow (tr, X) \in \mathcal{SF}(Spec)$$

Thus, a sufficient property $Spec^{\subseteq}$ has to be sufficiently strong to contain *only* the behaviors of the CSP specification *Spec*. At the same time, it has to be weak enough to contain the behaviors of a concrete CUC program *Impl*, which is shown in step *3)*. The inclusion relation is visualized in Figure 5.1b. The inclusion relation also shows that the weaker $Spec^{\subseteq}$ is, the more CUC implementations can be shown to be a refinement. The ideal property $Spec^{\subseteq}$ is describing exactly the failures of *Spec*. The complexity of finding an ideal property $Spec^{\subseteq}$ is similar to finding an invariant, and as such, there is no automatic way of finding it in general. For our framework, we just require a proof showing that the failures captured by $Spec^{\subseteq}$ are failures of *Spec*. We use our formalization in Isabelle/HOL to assure that this manual process does not introduce errors. We outline alternative approaches in 5.5.2.

In step *3)*, it needs to be shown that the property $Spec^{\subseteq}$ holds for the CUC program *Impl*. The Hoare logic presented in the previous section is well-suited for this task. In the next subsection, we will conduct such a proof for an example consisting of a simple buffer and its parallel combination. The result of step *3)* is formally captured by

$$(tr, \sigma, X) \in [\![Impl]\!]^{\mathcal{SF}} \Longrightarrow Spec^{\subseteq}(tr, X)$$

i. e., we show that the failures of *Impl* fulfill $Spec^{\subseteq}$, ignoring the internal state information of CUC.

After completing all three steps, we get by transitivity that

$$(tr, \sigma, X) \in [\![Impl]\!]^{\mathcal{SF}} \Longrightarrow (tr, X) \in \mathcal{SF}(Spec)$$

holds, which is equivalent to our goal $Spec \sqsubseteq_{\mathcal{SF}} Impl$.

5.5.1 Example

We demonstrate the workflow as presented above and show that a given CSP specification *Spec* for a one place buffer is refined by a given CUC implementation *Impl*. We have formalized this example in the Isabelle/HOL formalization in [BBD+19]. The CSP specification and its CUC implementation are shown in Figure 5.2. The elements that can be stored in the buffer are of type $\mathbb{T}$. *Spec* waits for an input on channel *in*, i. e., synchronizes on any event $\{in.x \mid x \in \mathbb{T}\}$, outputs the received value x on channel *out*, and then starts over. Next, we

$$
\begin{array}{l}
\mathit{Spec} = \mathit{in}?x : \mathbb{T} \to \mathit{out}!x \to \mathit{Spec} \\
\\
\mathit{Impl} := \\
\quad\quad 1\colon\ \textbf{do}\ \ (\lambda ds.\ \{ds[\mathit{free} := \textsc{True}]\}) \\
\oplus \quad 2\colon\ \textbf{comm}\ \ f_{ev}\ \ f_{ds}\ \ \text{where} \\
\quad\quad\quad f_{ev}\ =\ (\lambda ds.\ \{\mathit{in}.x\ \mid ds(\mathit{free}) = \textsc{True} \wedge x \in \mathbb{T}\} \\
\quad\quad\quad\quad\quad\quad \cup \{\mathit{out}.x \mid ds(\mathit{free}) = \textsc{False} \wedge x = ds(\mathit{buffer})\}) \\
\quad\quad\quad f_{ds}\ =\ (\lambda ds\ ev.\ \textbf{case}\ ev\ \textbf{of} \\
\quad\quad\quad\quad\quad\quad \mid \mathit{in}.x\ \ \Rightarrow ds[\mathit{buffer} := x, \mathit{free} := \textsc{False}] \\
\quad\quad\quad\quad\quad\quad \mid \mathit{out}.x \Rightarrow ds[\mathit{free} := \textsc{True}]) \\
\oplus \quad 3\colon\ \textbf{cbr}\ \ (\lambda ds.\ \textsc{True})\ \ 2\ \ 2
\end{array}
$$

Figure 5.2: CSP Specification and CUC Implementation of a One Place Buffer

explain *Impl* instruction by instruction:
`(1:do)` – This is the initialization. The boolean *free* indicates that the *buffer* is ready to store data.
`(2:comm)` – The `comm`-instruction offers events and changes the state after the communication happened. The events offered by f_{ev} are all values of type $\mathbb{T}$ on channel *in* if the buffer is free, else only the output event with the value stored in the buffer is offered. According to the event communicated, it either stores the input value and sets the buffer to not free, or it frees the buffer.
`(3:cbr)` – The conditional branch is used in this case to model an unconditional branch and always jumps back to the `comm`-instruction at label 2.

Step 1: Manual Extraction of $\mathit{Spec}^{\subseteq}$ from *Spec*

First, we extract a sufficient property $\mathit{Spec}^{\subseteq}$, which is only true for the failures of *Spec*. Let tr^* mean tr zero or more times concatenated, where the variable $\widetilde{x} \in \mathbb{T}$ is fresh in every occurrence of tr. Let F be a single failure and $\mathbb{F}_{empty}$ and $\mathbb{F}_{full}$ sets of failures. We define

$$
\begin{aligned}
\mathit{Spec}^{\subseteq}(F) &:= F \in \mathbb{F}_{empty} \vee F \in \mathbb{F}_{full} \qquad\qquad \text{where} \\
\mathbb{F}_{empty} &:= \left\{ \left(\langle \mathit{in}.\widetilde{x}, \mathit{out}.\widetilde{x}\rangle^*, X\right) \;\middle|\; X \subseteq \Sigma \setminus \{\mathit{in}.y \mid y \in \mathbb{T}\} \right\} \\
\mathbb{F}_{full} &:= \left\{ \left(\langle \mathit{in}.\widetilde{x}, \mathit{out}.\widetilde{x}\rangle^* \frown \mathit{in}.y, X\right) \;\middle|\; y \in \mathbb{T} \wedge X \subseteq \Sigma \setminus \{\mathit{out}.y\} \right\}
\end{aligned}
$$

This means, we choose pairs of matching inputs and outputs and at most one "free" input at the end. Initially and after an output, only inputs are possible. After an input only the matching output is possible.

Step 2: $\mathit{Spec}^{\subseteq} \Longrightarrow \mathit{Spec}$

We prove that $\mathit{Spec}^{\subseteq}$ implies *Spec*, i. e., that $\mathit{Spec}^{\subseteq}$ is *only* satisfied for stable failures of *Spec*. In this simple case it is easy to see, as $\mathit{Spec}^{\subseteq}$ describes exactly the failures of *Spec*. We have

$$\mathit{Spec}^{\subseteq}(F) \Longrightarrow F \in \mathcal{SF}(\mathit{Spec})$$

Step 3: *Impl* $\Longrightarrow$ *Spec*$^{\subseteq}$

In the next step, we show that *Spec*$^{\subseteq}$ holds for all failures of the program *Impl*, or more exactly for all elements of the projection of the failures of *Impl* onto traces-refusal pairs. To this end, we define an invariant *Inv*, which implies *Spec*$^{\subseteq}$ but is effectively stronger as we need state information such as the current program counter. We also specify the initial failures with a precondition *Pre*. As precondition *Pre*, we assume that the trace is empty and the state is normal from Assumption 5.6 and, additionally that the state points to the first instruction.[4] In the following, we show $\{Pre\}\ Impl\ \{Inv\}$. First, we define *Pre* and *Inv*:

$$
\begin{aligned}
Pre(tr, \sigma, X) :=& \sigma_{pc} = 1 \wedge tr = \langle\rangle \wedge N(\sigma) \\
Inv :=& Pre \vee I_{2,3} \\
I_{2,3}(tr, \sigma, X) :=& \sigma_{pc} \in \{2, 3\} \wedge \\
& ((tr, X) \in \mathbb{F}_{empty} \wedge \sigma_{ds}(free) = \textsc{True} \vee \\
& (tr, X) \in \mathbb{F}_{full} \wedge \sigma_{ds}(free) = \textsc{False} \\
& \wedge \exists\, x.\ \sigma_{ds}(buffer) = x \wedge last(tr) = in.x)
\end{aligned}
$$

We show

$$(tr, \sigma, X) \in [\![Impl]\!]^{\mathcal{SF}}(\{(tr', \sigma', X') \mid Pre(tr', \sigma', X')\}) \Longrightarrow Inv(tr, \sigma, X) \wedge \neg N(\sigma)$$

with our Hoare calculus, i. e., $\{Pre\}\ Impl\ \{Inv \wedge \neg N\}$ holds. The formula part $\neg N(\sigma)$ ensures that the initial failures are no longer present in the stable failures of the program and we only consider communication failures. For brevity, we denote the instruction by their label and instruction name, e. g., 1: `do`. The idea of the Hoare calculus proof is that starting in *Pre*, 1: `do` leads to the loop (2: `comm` $\oplus$ 3: `cbr`) and $I_{2,3}$ holds. During execution of the loop, the invariant $I_{2,3}$ is preserved, thus overall the invariant $Inv \equiv Pre \vee I_{2,3}$ holds. As it is an infinite loop, all normal states, i. e., terminal failures, are removed. As the Invariant *Inv* without normal states implies either $\mathbb{F}_{empty}$ or $\mathbb{F}_{full}$, it implies the sufficient property *Spec*$^{\subseteq}$:

$$Inv(tr, s, X) \wedge \neg N(\sigma) \Longrightarrow Spec^{\subseteq}(tr, X)$$

We conclude

$$(tr, \sigma, X) \in [\![Impl]\!]^{\mathcal{SF}}(\{(tr', \sigma', X') \mid Pre(tr', \sigma', X')\}) \Longrightarrow Spec^{\subseteq}(tr, X)$$

Conclusion

From step *2)* and step *3)* we conclude that all trace-refusal pairs of *Impl* are failures of *Spec*, i.e., $Spec \sqsubseteq_{\mathcal{SF}} Impl$ holds. The relation $\sqsubseteq_{\mathcal{SF}}$ corresponds to the CSP (stable) failures refinement. Thus, *Impl* enjoys all liveness and safety properties of *Spec*.

[4] We do not need to assume that the initial state points to the first instruction, but it allows us to not consider the other cases in the invariant.

Concurrency

Thanks to the compositionality of refinement and parallel composition of CSP, we are able to model, e. g., a two place buffer by connecting two buffers. Still, all safety and liveness properties are preserved. Consider the following CSP processes:

$$\begin{aligned} \mathit{Spec}_1 &= \mathit{in}?x : \mathbb{T} \rightarrow \mathit{mid}!x \rightarrow \mathit{Spec}_1 \\ \mathit{Spec}_2 &= \mathit{mid}?x : \mathbb{T} \rightarrow \mathit{out}!x \rightarrow \mathit{Spec}_2 \\ \mathit{Spec}_{\|} &= \mathit{Spec}_1 \ {}_{\{mid\}}\|_{\{mid\}} \ \mathit{Spec}_2 \end{aligned}$$

We can show that two programs Impl_1 and Impl_2 (similar to the code in Figure 5.2, also injecting *mid* as common channel) refine Spec_1 and Spec_2 respectively. Let

$$\mathit{Impl}_{\|} = \mathit{Impl}_1 \ {}_{\{mid\}}\|_{\{mid\}} \ \mathit{Impl}_2$$

We can immediately follow that $\mathit{Spec}_{\|} \sqsubseteq_{\mathcal{SF}} \mathit{Impl}_{\|}$, which demonstrates the compositionality of our approach. Observe that it scales well with the number of components: a system with N components requires only N separate refinement proofs. For homogeneous systems (as this example), we even can reuse the refinement proof.

5.5.2 Automation Approaches

Currently, the formulation of $\mathit{Spec}^{\subseteq}$ and its proof with respect to the CSP specification are assumed to be performed manually with mechanical assistance in Isabelle/HOL. In a current Bachelor thesis, we are working on a safe under-approximation of the CSP process to generate $\mathit{Spec}^{\subseteq}$ automatically, while still being weak enough to contain the behaviors of the CUC implementation.

In a Diploma thesis [MA17], we have extracted CSP processes from a given CUC program and have used the refinement checker FDR4 to relate the CUC program and the CSP process. A related approach [KBG+11] uses manual annotations for the extraction of CSP processes, which are then used with FDR4. In contrast, the approach in [MA17] has the advantage that CUC uses the same communication mechanism as CSP, thus, theoretically, allowing an extraction without manual annotation. Practically, we face state space explosion problems typical to model checking: Firstly, the CSP processes are parametrized by all variables of the CUC program. However, we think that only variables local to the component need to be considered. The number of variables required for each (sub-) process could be reduced in future work with analysis techniques from compiler optimizations (such as live-variable analysis [NNH99]). Secondly, the data ranges of the variables have to be explicitly checked in FDR4. Here, another analysis to identify intervals with similar control flow akin to symbolic execution [BCD+18] would be beneficial with respect to scalability.

Although both approaches are still in the early stages of development, they show that the manual step of the construction of $\mathit{Spec}^{\subseteq}$ and its proof can be supported by tools and heuristics in the future.

5.6 Summary

In this chapter, we have defined *Communicating Unstructured Code*, an unstructured, low-level language with an abstract communication instruction. It allows us to connect an abstract specification and a low-level implementation both using abstract communication and, thus, to close the first half of the verification gap. By including the abstract communication mechanism of CSP in a low-level language, we achieve compositionality of communicating low-level components. It also allows us to focus on local low-level programs, for which we have developed a concise formalization thanks to our generic instruction schemes. Using our traces and stable failures semantics, the Hoare calculus, and the verification workflow, we only need to relate single components. For homogeneous components, we only need to show this relation once for each component type. To ensure the rigorousness of our framework, especially also for user supplied proofs, we have formalized the results of this chapter in the theorem prover Isabelle/HOL.[5] This concludes the first part out of two of our framework. In the next chapter, we describe the second part of our framework in which we relate CUC programs to low-level programs that do not provide abstract communication primitives but are based on shared variable communication.

[5] All results apart from the concurrent cases in Theorems 5.1 and 5.2 (Correspondence of Denotational Semantics and Operational Characterizations) are formalized in Isabelle/HOL. All parts that are required for the mechanization of proofs that are supplied by users of our framework are formalized. When we developed the theory for CUC and formalized it in Isabelle/HOL, we focused on single components, as the concurrent denotational semantics of CUC are the same as the concurrent semantics of CSP and therefore we can use the concurrent compositionality of CSP refinement for results about concurrent components. We can apply the concurrent compositionality of CSP refinement also to refinements including CUC, as CUC uses the communication mechanism of CSP and we have shown the compatibility of CUC to CSP in Section 5.3.5. This approach still manifests itself in the Hoare calculus in Section 5.4, which we have also only defined for single components. However, to relate CUC to a low-level language with shared variables in the next chapter, we require an *operational* semantics with concurrent execution for CUC. For a uniform representation of the operational semantics of CUC we extend it with rules for concurrent execution. However, extending the proof in Isabelle/HOL would be tedious and very time-consuming.

Chapter 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 ($\sigma \in \mathit{States}$) 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

$$\mathit{GStates} := \mathit{DS} \times \mathit{States}$$

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

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

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 γ reads the value of a shared variable γ into a local register x.

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

`cas` r γ 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 5.3) and a concurrent program *cp* to be a tree of local programs (Definition 5.5). We also require the uniqueness of labels (Assumption 5.2) and that the tree structure of a concurrent state matches the tree structure of a program (Assumption 5.3). 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.

Definition 6.3: Operational Semantics of SV

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

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

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

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

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

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

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

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

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

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 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

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

Figure 6.1: Implementation of the Handshake Protocol: *send* and *receive*

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 6.1. 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 6.2 on page 91 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 5.3). 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 Section 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 5.3), 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.

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

$\texttt{comm}_s$ 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 $s \in ids(cp_i) \vee r \in ids(cp_i)$ 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 5.14.

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:

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

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*.

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

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 communication 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.

$$EL\colon GStates \times ID \times Instructions \to \Sigma \cup \{\tau, \tau_c\}$$

$$\begin{aligned} EL\big((\Gamma, \_), id, \texttt{read}\ \_\ \gamma_c\big) &:= c.s.r.v && \text{where } s = \Gamma(m_c), r = id, v = \Gamma(\gamma_c) \\ EL(\_, \_, ins) &:= \tau_c && \text{if } ins \text{ is part of } send \text{ or } receive \text{ (see Fig. 6.1)} \\ EL(\_, \_, \_) &:= \tau && \text{otherwise} \end{aligned}$$

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

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

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

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

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

Definition 6.13: Operational Traces Semantics of SV

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

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

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

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

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

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:

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

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 $X \subseteq \Sigma$.

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

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.

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

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 <u>in</u> the channel, and after the communication happened, the channel will be eventually <u>un</u>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 6.2 in the rectangular boxes in the middle column. Figure 6.2 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 <u>n</u>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 channel-states.

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

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

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

[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.

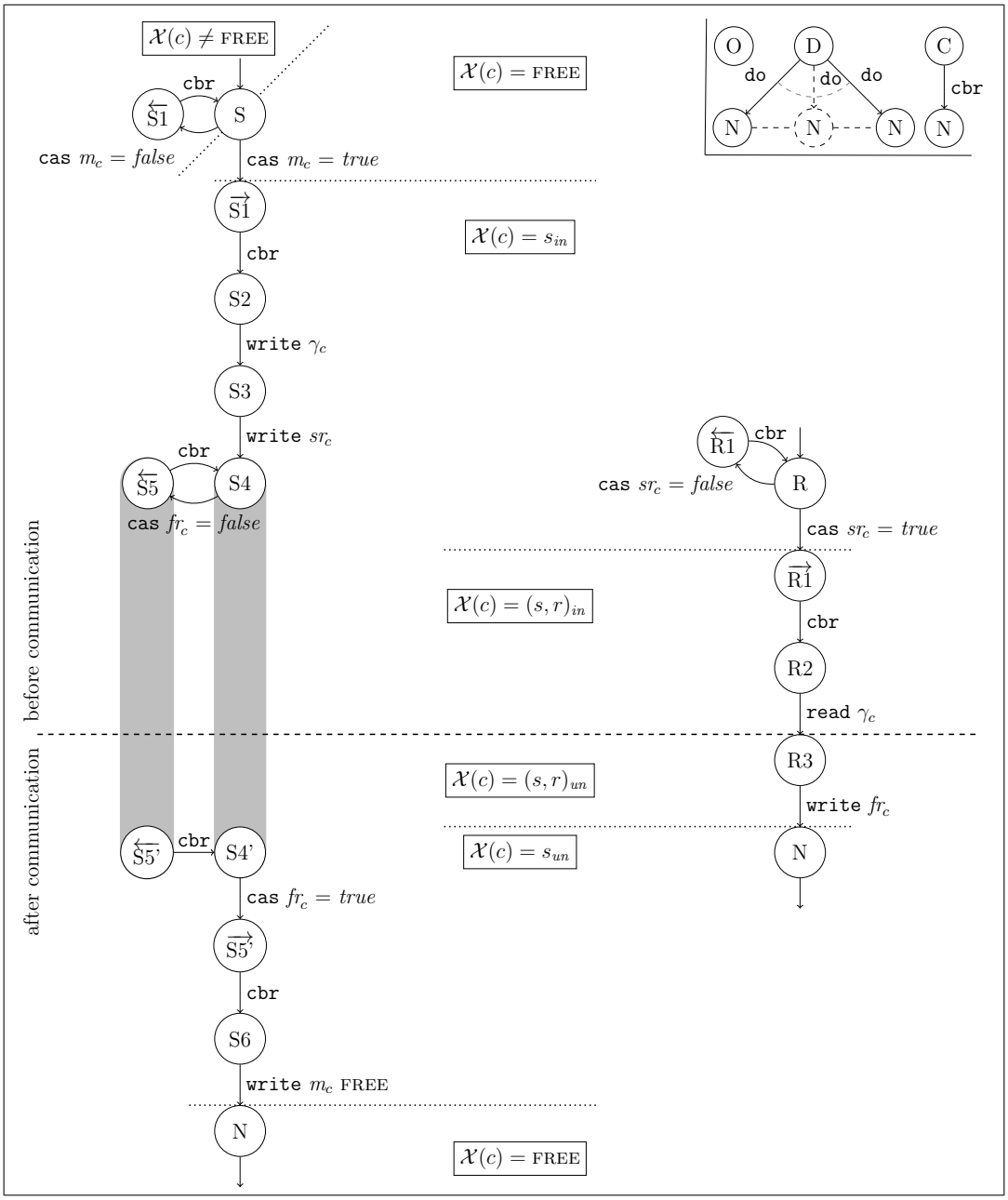


Figure 6.2: The Flow of the Handshake Protocol

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.

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:

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

Up-simulation:

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

Unlocking-simulation:

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

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) correspond to the states with the same names in the protocol flow in Figure 6.2. The disjuncts describe triplets $(cuc, \mathcal{X}, sv)$,

Definition 6.20: Protocol Restrictions

$$\mathcal{P}_{cuc,sv,\psi}\big(\sigma,\mathcal{X},(\Gamma,\hat{\sigma})\big) := \big(\forall\, c.\, \mathcal{X}(c) = \textsc{free} \Longrightarrow \neg\Gamma(sr_c) \wedge \neg\Gamma(fr_c)\big) \\ \wedge\ \forall\, id. \mathcal{P}^{id}_{cuc,sv,\psi}\big(\sigma,\mathcal{X},(\Gamma,\hat{\sigma})\big)$$

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

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.

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

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`.

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

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.

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.4 in Definition A.1.

Although we have presented our method for a concrete (handshake) protocol, it provides the foundation for a more generalized notion of relations between abstract synchronous and concrete asynchronous communication based on other communication/synchronization protocols. The presented protocol can be divided into four phases (which match with the four non-FREE 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

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

Proof

Using the Definitions 5.13 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 $\mathcal{X} = \emptyset$

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

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*

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

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.6. □

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

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 6.3 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.

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

Figure 6.3: Alternative Implementation of the Handshake Protocol Without Sender Identifier in the Mutex

Corollary 6.1: Liveness Properties Without Sender Identifier

An adaption of the handshake protocol given in Figure 6.3, 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.

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

Proof: Idea

The proof can be found in Appendix A.5 and is similar to bisimilarity proofs: all possible transitions of one part can be answered by its counterpart. An important difference is

Proof: Idea

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.

Chapter 7

Evaluation & Case Study

In this chapter, we illustrate the application of our framework. The main advantages of our framework are its scalability with respect to the number of components, the verification of not only safety, but also liveness properties, the verification of non-terminating systems, and the rigorousness of the verification from abstract specification down to low-level code. To demonstrate this, especially the scalability with the number of components, we consider the example of redundant m-servers-n-clients. It is a system with n client processes, which delegate complex computations to m[1] server processes. They request the same computation/information from m server processes for redundancy. Redundant sensors or computations are usual in safety critical systems to decrease the reliability on individual sensors or results. Our focus in this example is on the communication between the components. The example is suitable to demonstrate the application of our approach, as it is concurrent, communicating, and non-terminating. Due to the compositionality of our approach, we do not need to consider all processes at once. We can refine a single client and a single server separately, and obtain directly the preservation of safety and liveness properties for systems with arbitrarily many clients.

First, we specify the system and its components in CSP in Section 7.1. We implement the specification in CUC in Section 7.4 and show that it is a stable failures refinement of the specification, using the first part of our framework: In Section 7.2, we define the connecting property and in Section 7.5 we prove the refinement from CSP to CUC using the Hoare calculus. Finally, we replace the abstract communication with the handshake protocol and obtain an implementation in SV in Section 7.6, using the second part of our framework. In Theorem 6.4 we have shown that this implies also a stable failures refinement between CUC and SV. By transitivity, we obtain that all safety and liveness properties from the abstract specification in CSP are preserved in the low-level implementation with communication over shared variables in SV.

[1] The degree of redundancy, i. e., the number of servers, is arbitrary but fixed. For simplicity of this example, and as it is required for most safety critical systems, we choose a degree of redundancy of 3. With an arbitrary number m, we would need to consider processes and programs of variable length, which does not help the understanding of the example. The processes and proofs are easily adaptable to other instantiations of m.

7.1 Specification in CSP

In the following, we present the specification of our system in CSP. We have n homogeneous client components, which send an input for a computation consecutively to three redundant server components, which perform the requested computation (in our example modeled by the function f). The client then waits for the results of the computations. As our example is focused on the communication aspect, the client then finishes. In an actual system, the client would process the data, determine, e. g., via voting, which data to accept, and continue its execution depending on the accepted data. Once the servers have sent back the computed value they are ready for new input. For the simplicity of this example, we choose not to let the client start over when it is finished. Apart from even longer formulas, the proof would not get more complicated.

We first introduce a more abstract version of the specification (called $Client^{simple}$ and $Server^{simple}$), and then we introduce a version, which conforms to the restrictions defined in Section 6.2 in Assumption 6.1, i. e., all components are combined with the alphabetized parallel operator and use communication interface that ensure that every event has distinct senders and receivers. This conformity allows us to later apply Theorem 6.4 that shows that all fitting programs have the same safety and liveness properties. We consider six channels $a_1, a_2, a_3, b_1, b_2, b_3$, one to receive and one to send for each server. The server is parametrized with an id, which determines the channels assigned to the server.

$$
\begin{aligned}
Client^{simple} &= \bigsqcap_{v \in \mathbb{T}} a_1!v \rightarrow a_2!v \rightarrow a_3!v \rightarrow b_1?y_1 \rightarrow b_2?y_2 \rightarrow b_3?y_3 \rightarrow STOP \\
Server_i^{simple} &= a_i?x \rightarrow b_i!f(x) \rightarrow Server_i^{simple}
\end{aligned}
$$

We specify the whole system as follows, where $|||$ is the interleaving operator, which only allows interleaving steps and no synchronization.

$$
\begin{aligned}
Clients^{simple}(n) &= \mathop{|||}_{n} Client^{simple} \\
Servers^{simple} &= \mathop{|||}_{i \in \{1,2,3\}} Server_i^{simple} \\
System^{simple}(n) &= Clients^{simple}(n) \ {}_{\{a_1,a_2,a_3,b_1,b_2,b_3\}}\|_{\{a_1,a_2,a_3,b_1,b_2,b_3\}} \ Servers^{simple}
\end{aligned}
$$

To comply with the restrictions from Assumption 6.1 for our handshake protocol in SV, which we describe in the following, we transform the specification to use the communication as described in 6.2. That means, we embed the sender and receiver *id* into the events to obtain purely unidirectional communication. We also replace all events to be sent with an external choice over all possible receivers. Thus, we make the receivers explicit. Like in CSP, the restriction to send to a certain receiver is ensured by the use of a special event structure,

facilitated by the dot-notation of CSP. In addition to a channel name, we also use the id of the sender and of the receiver. As introduced in Section 6.2, we use events $a.c.id^{out}.v$ that consist of four parts. Let a be the channel name, c the id of the client, ID the set of all ids, and v the value to be sent. We use the following replacement for sending a value ($a!v$):

$$a!v \rightarrow P \rightsquigarrow \underset{id^{out} \in ID \setminus \{c\}}{\Box} a.c.id^{out}.v \rightarrow P$$

Note the use of external choice ($\Box$). The receiver needs to be able to choose its id^{out} from all offered events. We allow all ids but c as the id of the potential receiver. We perform a similar replacement for receiving of event ($a?x$).

$$a?x \rightarrow P(x) \rightsquigarrow \underset{id^{in} \in ID \setminus \{c\}}{\Box} a.id^{in}.c?x \rightarrow P(x)$$

When applied to the server, we get the following result. Let s_i for $i \in \{1, 2, 3\}$ be the ids of the servers. All operators are right-associative.

$$Server_i = \underset{id^{in} \in ID \setminus \{s_i\}}{\Box} a_i.id^{in}.s_i?x \rightarrow \underset{id^{out} \in ID \setminus \{s_i\}}{\Box} b_i.s_i.id^{out}.f(x) \rightarrow Server_i$$

As we can model the internal choice over the value of v with a non-deterministic `do` (think sensor read), we can keep the internal choice for the client and implement it in CUC. Additionally, we would have two options to handle internal choice: Either we refine it to a deterministic choice within CSP, or refine it to a deterministic choice in the refinement to CUC. When applying the replacements to the client, we get the following result. Let c be the id of the client.

$$\begin{aligned}
&Client_c = \\
&\bigsqcap_{v \in \mathbb{T}} \Bigg(\underset{id_1^{out} \in ID \setminus \{c\}}{\Box} a_1.c.id_1^{out}.v \rightarrow \underset{id_2^{out} \in ID \setminus \{c\}}{\Box} a_2.c.id_2^{out}.v \rightarrow \underset{id_3^{out} \in ID \setminus \{c\}}{\Box} a_3.c.id_3^{out}.v \rightarrow \\
&\qquad \underset{id_1^{in} \in ID \setminus \{c\}}{\Box} b_1.id_1^{in}.c?y_1 \rightarrow \underset{id_2^{in} \in ID \setminus \{c\}}{\Box} b_2.id_2^{in}.c?y_2 \rightarrow \underset{id_3^{in} \in ID \setminus \{c\}}{\Box} b_3.id_3^{in}.c?y_3 \rightarrow \\
&\qquad STOP \Bigg)
\end{aligned}$$

The constraints in Section 6.2 do not allow for the interleaving operator ($|||$), as the definition of communication interfaces is not included. We replace it with an alphabetized parallel operator ($\|$) where we do not allow the ids of the other component as senders or receivers in the communication interface.

$$cp_i \,|||\, cp_j \rightsquigarrow cp_i \;{}_{\alpha_i}\|_{\alpha_j}\; cp_j$$
$$\alpha_i = \big\{a.s.r.v \in \Sigma \mid \big(s \in ids(cp_i) \vee r \in ids(cp_i)\big) \wedge r, s \notin ids(cp_j)\big\}.$$

As a result, we get the following overall system:

$$
\begin{aligned}
\mathit{Clients}(n) &:= \Big(\big(\mathit{Client}_{c_1} \;{}_{\beta_{c_1}\setminus\beta_{c_2}}\|_{\beta_{c_2}\setminus\beta_{c_1}}\; \mathit{Client}_{c_2}\big) \;{}_{(\beta_{c_1}\cup\beta_{c_2})\setminus\beta_{c_3}}\|_{\beta_{c_3}\setminus(\beta_{c_1}\cup\beta_{c_2})} \cdots \\
&\qquad \cdots\Big) \;{}_{(\beta_{c_1}\cup\ldots\cup\beta_{c_{n-1}})\setminus\beta_{c_n}}\|_{\beta_{c_n}\setminus(\beta_{c_1}\cup\ldots\cup\beta_{c_{n-1}})}\; \mathit{Client}_{c_n} \\
\mathit{Servers} &:= \big((\mathit{Server}_1 \;{}_{\beta_{s_1}\setminus\beta_{s_2}}\|_{\beta_{s_2}\setminus\beta_{s_1}}\; \mathit{Server}_2) \;{}_{(\beta_{s_1}\cup\beta_{s_2})\setminus\beta_{s_3}}\|_{\beta_{s_3}\setminus(\beta_{s_1}\cup\beta_{s_2})}\; \mathit{Server}_3\big)
\end{aligned}
$$

$$
\mathit{System}(n) := \mathit{Clients} \;{}_{\beta_{c_1}\cup\ldots\cup\beta_{c_n}}\|_{\beta_{s_1}\cup\beta_{s_2}\cup\beta_{s_3}}\; \mathit{Servers}
$$

where we use the following shorthand for the communication interfaces. The set β_{id} describes all events as defined below, where the component with id id is either the sender or the receiver.

$$
\beta_{id} = \{a.s.r.v \in \Sigma \mid (s = id \vee r = id) \wedge s \neq r\}.
$$

Using FDR4, we have validated for concrete instances of n and f that $\mathit{System}^{simple}(n)$ and $\mathit{System}(n)$ are stable failures equivalent (using $n = 3$ and $f(x) = ((x)\%N) + 1$). Having specified the system in CSP, we formulate the sufficient property, which captures the failures of the specification as an assertion. Later, we use the Hoare calculus to show that the sufficient property holds for the CUC implementation of the system.

7.2 Sufficient Property: Specification as an Assertion

The sufficient property is a (possibly stronger) reformulation of the CSP process as an assertion on trace-refusal pairs. It is sufficient for the CSP process in the sense that all behaviors described by the sufficient property are also behaviors of the CSP process. In contrast to the CSP process, the sufficient property can be used as an assertion in our Hoare calculus. When showing later in Section 7.5 the sufficient property for the CUC program, we enrich the sufficient property with information about the states. In this section, we define the sufficient properties for the server and the client. In the next section, we show that they are indeed sufficient, i. e., they only capture stable failures of the respective CSP process

A systematic way to formulate a sufficient property is to identify possible loops, describe their recurring traces, and then formulate failures for each prefix of the loop(s). The server has a loop with a trace length of 2, and we describe the two prefixes separately. The failures with an even trace length where the server is waiting for the next input are captured by $\mathbb{F}_i^{idle}$. The failures with an odd trace length where the server is going to perform its calculation are captured by $\mathbb{F}_i^{busy}$. Let $\langle a_i.id_*^{in}.s_i.x_*, b_i.s_i.id_*^{out}.f(x_*)\rangle^*$ be the concatenation of zero or more

times $\langle a_i.id_*^{in}.s_i.x_*, b_i.s_i.id_*^{out}.f(x_*)\rangle$ where $id_*^{in}, id_*^{out}, x_*$ can be different for every repetition.

$$S_i^{\subseteq}(F) := F \in \mathbb{F}_i^{idle} \vee F \in \mathbb{F}_i^{busy} \qquad \text{where}$$

$$\mathbb{F}_i^{idle} := \Big\{ \big(\langle a_i.id_*^{in}.s_i.x_*, b_i.s_i.id_*^{out}.f(x_*)\rangle^*, X\big) \;\Big|\; X \subseteq \Sigma \setminus \{a_i.id^{in}.s_i.v \mid id^{in} \in ID \setminus \{s_i\} \wedge v \in \mathbb{T}\} \Big\}$$

$$\mathbb{F}_i^{busy} := \Big\{ \big(\langle a_i.id_*^{in}.s_i.x_*, b_i.s_i.id_*^{out}.f(x_*)\rangle^* \frown \langle a_i.id^{in}.s_i.v\rangle, X\big) \;\Big|\; X \subseteq \Sigma \setminus \big\{b_i.s_i.id^{out}.f(v) \,\big|\, id^{out} \in ID \setminus \{s_i\}\big\} \wedge id^{in} \in ID \setminus \{s_i\} \wedge v \in \mathbb{T} \Big\}$$

Note the difference of the set comprehension ("quantification") over id^{in}/id^{out} and v. In the definition of $\mathbb{F}_i^{idle}$ they appear *in* the complement of the maximal refusal set. In the definition of $\mathbb{F}_i^{busy}$ they appear *after* the maximal refusal set, thus, the suffix $\langle a_i.id^{in}.s_i.v\rangle$ can occur for any sender id^{in} and any value v.

We formulate the sufficient property for the client in Figure 7.1. It does not have a loop. We describe the possible prefixes grouped by length.

7.3 Correctness Proof of the Sufficient Property

We prove for both sufficient properties that they describe at most the failures of the respective CSP processes. In both cases, it is a simple case analysis. We give the full proof for the server in Appendix A.7.1.

Lemma 7.1: Both Sufficient Properties are Correct

$$C_c^{\subseteq}(F) \Longrightarrow F \in \mathcal{SF}(Client_c)$$
$$S_i^{\subseteq}(F) \Longrightarrow F \in \mathcal{SF}(Server_i)$$

7.4 Implementation in CUC

We define the system in CUC. To this end, we define a client and a server. Both are depicted in Figure 7.2. The client non-deterministically generates a value, which it sends over three channels. It collects the three results and then terminates. The server receives a value and performs its computation modeled as the application of the function f. Afterwards, the server sends the computed values and is ready to accept new input. For these short programs it is easy to see that both programs do not diverge and, thus, fulfill Assumption 5.7 (Divergence Freedom of CUC Programs).

$$
\begin{aligned}
&C_c^{\subseteq}(F) \coloneqq F \in \\
&\Big\{ (\langle\rangle, X) \,\Big|\, X \subseteq \Sigma \setminus \big\{ a_1.c.id_1^{out}.v \,\big|\, id_1^{out} \in ID \setminus \{c\} \big\} \wedge v \in \mathbb{T} \Big\} \cup \\
&\Big\{ \big(\langle a_1.c.id_1^{out}.v \rangle, X\big) \\
&\quad \Big| \, X \subseteq \Sigma \setminus \big\{ a_2.c.id_2^{out}.v \,\big|\, id_2^{out} \in ID \setminus \{c\} \big\} \wedge id_1^{out} \in ID \setminus \{c\} \wedge v \in \mathbb{T} \Big\} \cup \\
&\Big\{ \big(\langle a_1.c.id_1^{out}.v, a_2.c.id_2^{out}.v \rangle, X\big) \\
&\quad \Big| \, X \subseteq \Sigma \setminus \big\{ a_3.c.id_3^{out}.v \,\big|\, id_3^{out} \in ID \setminus \{c\} \big\} \wedge id_1^{out}, id_2^{out} \in ID \setminus \{c\} \wedge v \in \mathbb{T} \Big\} \cup \\
&\Big\{ \big(\langle a_1.c.id_1^{out}.v, a_2.c.id_2^{out}.v, a_3.c.id_3^{out}.v \rangle, X\big) \\
&\quad \Big| \, X \subseteq \Sigma \setminus \big\{ b_1.id_1^{in}.c.y_1 \,\big|\, id_1^{in} \in ID \setminus \{c\} \wedge y_1 \in \mathbb{T} \big\} \wedge \\
&\quad id_1^{out}, id_2^{out}, id_3^{out} \in ID \setminus \{c\} \wedge v \in \mathbb{T} \Big\} \cup \\
&\Big\{ \big(\langle a_1.c.id_1^{out}.v, a_2.c.id_2^{out}.v, a_3.c.id_3^{out}.v, b_1.id_1^{in}.c.y_1 \rangle, X\big) \\
&\quad \Big| \, X \subseteq \Sigma \setminus \big\{ b_2.id_2^{in}.c.y_2 \,\big|\, id_2^{in} \in ID \setminus \{c\} \wedge y_2 \in \mathbb{T} \big\} \wedge \\
&\quad id_1^{out}, id_2^{out}, id_3^{out}, id_1^{in} \in ID \setminus \{c\} \wedge v, y_1 \in \mathbb{T} \Big\} \cup \\
&\Big\{ \big(\langle a_1.c.id_1^{out}.v, a_2.c.id_2^{out}.v, a_3.c.id_3^{out}.v, b_1.id_1^{in}.c.y_1, b_2.id_2^{in}.c.y_2 \rangle, X\big) \\
&\quad \Big| \, X \subseteq \Sigma \setminus \big\{ b_3.id_3^{in}.c.y_3 \,\big|\, id_3^{in} \in ID \setminus \{c\} \wedge y_3 \in \mathbb{T} \big\} \wedge \\
&\quad id_1^{out}, id_2^{out}, id_3^{out}, id_1^{in}, id_2^{in} \in ID \setminus \{c\} \wedge v, y_1, y_2 \in \mathbb{T} \Big\} \cup \\
&\Big\{ \big(\langle a_1.c.id_1^{out}.v, a_2.c.id_2^{out}.v, a_3.c.id_3^{out}.v, b_1.id_1^{in}.c.y_1, b_2.id_2^{in}.c.y_2, b_3.id_3^{in}.c.y_3 \rangle, X\big) \\
&\quad \Big| \, X \subseteq \Sigma \wedge id_1^{out}, id_2^{out}, id_3^{out}, id_1^{in}, id_2^{in}, id_3^{in} \in ID \setminus \{c\} \wedge v, y_1, y_2, y_3 \in \mathbb{T} \Big\}
\end{aligned}
$$

Figure 7.1: Sufficient Property for the Client

Client $\coloneqq$
1: **do** $(\lambda ds.\ \{ds[x \coloneqq v] \mid v \in \mathbb{T}\})$
2: $\text{comm}_s\ \ a_1\ \ x$
3: $\text{comm}_s\ \ a_2\ \ x$
4: $\text{comm}_s\ \ a_3\ \ x$
5: $\text{comm}_r\ \ b_1\ \ y_1$
6: $\text{comm}_r\ \ b_2\ \ y_2$
7: $\text{comm}_r\ \ b_3\ \ y_3$

Server$_i$ $\coloneqq$
1: $\text{comm}_r\ \ a_i\ \ x$
2: **do** $(\lambda ds.\ \{ds[y \coloneqq f(ds(x))]\})$
3: $\text{comm}_s\ \ b_i\ \ y$
4: **cbr** $(\lambda ds.\ \textsc{True})\ \ 1\ \ 1$

Figure 7.2: Client and Server in CUC

7.5 The CUC Programs fulfills the Sufficient Property

We use the Hoare calculus to show that the failures of the CUC programs fulfill the sufficient properties. We require a structured version of the program as the Hoare calculus is defined for the stable failures semantics of CUC, which is defined on structural programs. Due to the correspondence of the operational and denotational semantics (Theorems 5.1 and 5.2), we can consider structured versions of the programs. We sketch the proof for the server. We consider a structured version $s_i^{\oplus}$ of the server.[2] We show that all failures of the program $s_i^{\oplus}$ fulfill $S_i^{\subseteq}$. The initial precondition requires a normal state ($N(\sigma)$) at the first instruction ($\sigma_{pc} = 1$) and no previously observed behavior ($tr = \langle\rangle$). We do not specify the refusal sets, as everything can be refused in a normal state[3] as also captured by Assumption 5.6 about the initial failures of a CUC program. We show the following Hoare triple:

Lemma 7.2: The CUC Server Fulfills its Sufficient Property

$$\{\sigma_{pc} = 1 \wedge N(\sigma) \wedge tr = \langle\rangle\}\ s_i^{\oplus}\ \left\{\lambda tr\ \sigma\ X.\ S_i^{\subseteq}(tr, X)\right\}$$

In Figure 7.3, we show the outline of the proof using the Hoare calculus. We use circled line numbers as in, e. g., $(((1 \oplus 2) \oplus 3) \oplus 4)$ as a shorthand to describe (parts of) the considered program. We use Pre_i and $Post_i$ as shorthand for the pre- and postcondition of the individual instruction i. The full proof for the server with definitions and detailed explanations is given in the Appendix A.7.2. The proof for the client is similar.

Using the Hoare calculus, we have shown for both the client and the server that the failures of the CUC programs fulfill the sufficient properties, which in turn imply that the failures also belong to the CSP specification. Thus, we have shown a stable failures refinement for the single components. Due to the compositionality of the stable failures refinement of CSP, which we have extended to CUC, we can combine the refinement results for the individual components and obtain the refinement results for the entire system.

7.6 Implementation in SV

The application of the second part of the framework is automatic, if we have a CUC program that complies to the constraints for the application of the handshake protocol. As we constructed a concurrent CUC program that complies to the constraints, we generate fitting SV programs and then apply Theorem 6.4 to obtain that the SV program preserves all safety and liveness properties of the CUC program. By transitivity and the first part of this example, the SV program also preserves all safety and liveness properties of the CSP specification.

[2] According to Corollary 5.1 (Invariance Under Structure), we can pick any structure on the $Server_i$ program we like. They all have the same semantics.

[3] In CUC, communication is only offered *while* the `comm` instruction is executed. Thus, *before* any instruction, everything can be refused. This enables the distinction of states whose program counter only points to a `comm` instruction and states where the `comm` instruction is actually executed. This distinction is captured by the normal states and the communication states defined in Definition 5.17 (Communication States).

$$
\begin{array}{lll}
\text{H-COMM} & \text{H-DO} & \\
\dfrac{}{\{Pre_1\}\ 1 : \texttt{comm}_r\ \{Post_1\}} & \dfrac{}{\{Pre_2\}\ 2 : \texttt{do}\ \{Post_2\}} & \\
\vdots & \vdots & \\
\multicolumn{2}{c}{\dfrac{\qquad\qquad\qquad\qquad}{\{\cdots\}\ ① \oplus ②\ \{\cdots\}}} & \text{H-SEQ} \\
\vdots & \text{H-COMM} & \\
 & \dfrac{}{\{Pre_3\}\ 3 : \texttt{comm}_s\ \{Post_3\}} & \\
 & \vdots & \\
\multicolumn{2}{c}{\dfrac{\qquad\qquad\qquad\qquad}{\{\cdots\}\ (① \oplus ②) \oplus ③)\ \{\cdots\}}} & \text{H-SEQ} \\
\vdots & \text{H-CBR} & \\
 & \dfrac{}{\{Pre_4\}\ 4 : \texttt{cbr}\ \{Post_4\}} & \\
 & \vdots & \\
\multicolumn{2}{c}{\dfrac{\qquad\qquad\qquad\qquad}{\{\cdots\}\ ((① \oplus ②) \oplus ③) \oplus ④\ \left\{\begin{array}{l} \sigma_{pc} = 1 \wedge \neg N(\sigma) \wedge (tr, X) \in \mathbb{F}_i^{even} \vee \\ \sigma_{pc} = 3 \wedge \neg N(\sigma) \wedge (tr, X) \in \mathbb{F}_i^{odd} \end{array}\right\}}} & \text{H-SEQ} \\
\multicolumn{2}{c}{\dfrac{\qquad\qquad\qquad\qquad}{\{\sigma_{pc} = 1 \wedge N(\sigma) \wedge tr = \langle\rangle\}\ ((① \oplus ②) \oplus ③) \oplus ④\ \left\{\lambda tr\ \sigma\ X.\ S_i^{\subseteq}(tr, X)\right\}}} & \text{H-CONS}
\end{array}
$$

Figure 7.3: Proof Outline of the Hoare Calculus Proof for the Server

To generate SV programs from CUC programs, which only use the instantiation $\texttt{comm}_s$ and $\texttt{comm}_r$ of $\texttt{comm}$, we replace $\texttt{comm}_s$ and $\texttt{comm}_r$ with *send* and *receive*, respectively. Figure 7.4 shows the client and Figure 7.5 shows the server. In both figures, the CUC program is on the left side and the resulting SV program on the right side. In the middle, the program label map ψ is given. We only consider the sequential programs as components. The concurrent structure is the same in SV as in CUC. By construction, the SV programs of the client and the server are fitting with the given ψ to their CUC counterparts. This enables the application of Theorem 6.4.

We can now combine the single steps and obtain by transitivity that the SV implementation is a stable failures refinement of the CSP specification.

- Every stable failure of the SV implementation is a stable failure of the CUC implementation as shown in this section.
- All stable failures of the components of the CUC implementation satisfy their respective sufficient properties as shown in Section 7.5.
- All stable failures satisfying the sufficient properties are stable failures of the respective CSP specifications as shown in Section 7.3. This implies that all CUC components are stable failures refinements of their CSP specifications.
- Because the stable failures semantics of both CSP and CUC are compositional with respect to concurrent composition, the stable failures refinement is compositional as well. We have shown the compatibility of CUC to CSP in Section 5.5. Thus, the entire CUC implementation consisting of all components is a stable failures refinement of the CSP specification.

1: do $(\lambda ds.\ \{ds[x := v] \mid v \in \mathbb{T}\})$	$\psi(1) = 1$	1: do $(\lambda ds.\ \{ds[x := v] \mid v \in \mathbb{T}\})$
		2: cas hl_{a_1} m_{a_1} FREE TAKEN
		3: cbr hl_{a_1} 2 4
		4: write γ_{a_1} x
2: $\mathtt{comm}_s$ a_1 x	$\psi(2) = 2$	5: write sr_{a_1} $\top$
		6: cas ss_{a_1} fr_{a_1} $\top$ $\bot$
		7: cbr ss_{a_1} 6 8
		8: write m_{a_1} FREE
		9: cas hl_{a_2} m_{a_2} FREE TAKEN
		10: cbr hl_{a_2} 9 11
		11: write γ_{a_2} x
3: $\mathtt{comm}_s$ a_2 x	$\psi(3) = 9$	12: write sr_{a_2} $\top$
		13: cas ss_{a_2} fr_{a_2} $\top$ $\bot$
		14: cbr ss_{a_2} 13 15
		15: write m_{a_2} FREE
		16: cas hl_{a_3} m_{a_3} FREE TAKEN
		17: cbr hl_{a_3} 16 18
		18: write γ_{a_3} x
4: $\mathtt{comm}_s$ a_3 x	$\psi(4) = 16$	19: write sr_{a_3} $\top$
		20: cas ss_{a_3} fr_{a_3} $\top$ $\bot$
		21: cbr ss_{a_3} 20 22
		22: write m_{a_3} FREE
		23: cas ss_{b_1} sr_{b_1} $\top$ $\bot$
		24: cbr ss_{b_1} 23 25
5: $\mathtt{comm}_r$ b_1 y_1	$\psi(5) = 23$	25: read y_1 sr_{b_1}
		26: write fr_{b_1} $\top$
		27: cas ss_{b_2} sr_{b_2} $\top$ $\bot$
		28: cbr ss_{b_2} 27 29
6: $\mathtt{comm}_r$ b_2 y_2	$\psi(6) = 27$	29: read y_2 sr_{b_2}
		30: write fr_{b_2} $\top$
		31: cas ss_{b_3} sr_{b_3} $\top$ $\bot$
		32: cbr ss_{b_3} 31 33
7: $\mathtt{comm}_r$ b_3 y_3	$\psi(7) = 31$	33: read y_3 sr_{b_3}
		34: write fr_{b_3} $\top$

Figure 7.4: The Client in CUC and in SV

		1: $\texttt{cas}\ ss_{a_i}\ sr_{a_i}\ \top\ \bot$
		2: $\texttt{cbr}\ ss_{a_i}\ 1\ 3$
1: $\texttt{comm}_r\ a_i\ x$	$\psi(1) = 1$	3: $\texttt{read}\ x\ sr_{a_i}$
		4: $\texttt{write}\ fr_{a_i}\ \top$
2: $\texttt{do}\ (\lambda ds.\ \{ds[y := f(x)]\})$	$\psi(2) = 5$	5: $\texttt{do}\ (\lambda ds.\ \{ds[y := f(x)]\})$
		6: $\texttt{cas}\ hl_{b_i}\ m_{b_i}$ FREE TAKEN
		7: $\texttt{cbr}\ hl_{b_i}\ 6\ 8$
		8: $\texttt{write}\ \gamma_{b_i}\ y$
3: $\texttt{comm}_s\ b_i\ y$	$\psi(3) = 6$	9: $\texttt{write}\ sn_{b_i}\ \top$
		10: $\texttt{cas}\ ss_{b_i}\ fr_{b_i}\ \top\ \bot$
		11: $\texttt{cbr}\ ss_{b_i}\ 10\ 12$
		12: $\texttt{write}\ m_{b_i}$ FREE
4: $\texttt{cbr}\ (\lambda ds.\ \text{TRUE})\ 1\ 1$	$\psi(4) = 13$	13: $\texttt{cbr}\ (\lambda ds.\ \text{TRUE})\ 1\ 1$

Figure 7.5: The Server in CUC and in SV

7.7 Summary

In this chapter, we have shown that the SV implementation of our system of n clients and three servers preserves all safety and liveness properties of the specification of the system in CSP. Our example system is concurrent, communicating and non-terminating. In particular, our approach scales well with the number of homogeneous or parametrizable components. We can refine each component individually, and only need to refine every component scheme once. The independence of the number of similar components allows for a parametric formulation of the overall system with an arbitrary number n of clients. Thanks to the use of the verified handshake protocol (using Theorem 6.4), we do not need to consider the non-compositionality of communication via shared variables. The application of the second part of our approach was fully automatic. We handle non-termination gracefully with our Hoare calculus on top of the denotational semantics for the low-level language CUC. In conclusion, we have demonstrated with a small example that our approach can handle parametrized systems, i. e., systems with an arbitrary number of components, and that it enables the rigorous verification of the preservation of properties from abstract specifications down to executable low-level code. In the next chapter, we give pointers for future work and summarize and discuss the presented approach and the contributions.

Chapter 8

Conclusion

In this chapter, we summarize and discuss our results. In Section 8.1, we summarize our contributions on the formal relation of an abstract specification and low-level code while preserving safety and liveness properties and point out the main advantages of our framework. In Section 8.2, we discuss our results with respect to the objectives given in the Introduction (Chapter 1). We close this chapter with outlines of ideas for future work in Section 8.3.

8.1 Results

In this thesis, we have presented a rigorous framework for the verification of low-level code. With our approach, it is possible to verify that a program in low-level code conforms to its *Communicating Sequential Processes* (CSP) specification, i. e., all safety and liveness properties are preserved. To this end, we have transferred the notion of CSP refinement to low-level code and thereby have extended the CSP refinement from CSP-only to also cover low-level code. Our formalization in a theorem prover enables user proofs to be mechanized and reusable. Together with the compositionality of our approach, this makes it possible to provide rigorous guarantees of concurrent low-level programs in a way that scales with the number of components.

In the first part of our framework, the calculation refinement part, the abstract specification in CSP is related to our intermediate low-level language *Communicating Unstructured Code* (CUC). CUC combines local low-level computing with abstract communication. The language CUC consists of three instruction schemes (`do`, `cbr`, `comm`). The generic state transformation instruction `do` can be instantiated to almost any instructions of a realistic instruction set, e. g., RISC-V [Wat16], ARM [Lim18], MIPS [Kan88] or the Intel family [Cor18]. Only two groups of instructions are not covered by the state transformation `do`: Branches and multi-processor synchronization instructions (e. g., *compare-and-set*, *compare-and-swap* or *load-reserved* and *store-conditional*). The unconditional and conditional branches are covered by CUC's conditional branch instruction `cbr`. Computed jumps are discussed in Future Work (Section 8.3). Communication between different components and synchronization of different components is achieved in CUC by its abstract communication instruction `comm`, which implements synchronous message passing. The multi-processor synchronization is covered in

the second part of our framework. As CUC uses the same communication mechanism as CSP and the same concurrent combination operator as CSP, we can extend the CSP refinement to CUC and inherit its concurrent compositionality. Thus, we can combine refinement results for single components into the refinement of the whole system. To show the relation between a component of the CSP specification and a component of the CUC program we first construct a *sufficient property*, which is a (possibly stronger) reformulation of the behaviors specified by the CSP process for the component. In contrast to the CSP process, the sufficient property is an assertion on failures using set theory. We enrich the sufficient property with state informations and use our Hoare calculus to prove that the sufficient property holds for all failures of the component of the CUC program. With this, we have proven that the CUC program is indeed an implementation of the CSP specification, preserving all safety and liveness properties.

In the second part of our framework, the communication refinement part, we relate the intermediate low-level program with abstract communication in CUC with a low-level program in our language *Shared Variables* (SV). SV is a low-level language that allows for the implementation of communication protocols over shared variables. Similar to CUC, it has a generic state transformation instruction scheme and a conditional branch instruction. In contrast to CUC, SV does not have an abstract communication instruction. To achieve synchronization of multiple components, SV has a synchronization primitive `cas` (compare-and-set). We have formalized and verified a simple handshake protocol using our notion of handshake refinement. From a concurrent CUC program that complies to the handshake protocol, we can generate a concurrent SV program that preserves all safety and liveness properties of the CUC program. As the formalization and the verification of the handshake protocol are once-and-for-all, the application of the second part of our framework reduces to a provably correct code generation for the user.

Our framework has four major advantages: compositionality, support for liveness properties, genericity and mechanized rigorous verification.

Our framework is compositional, i. e., it is sufficient to verify individual components. Due to the comprehensive logical nature of our approach, it is very easy to parametrize components, reducing the verification effort even further. This allows also for infinite or very large state spaces to be formalized and reasoned about. The compositionality and easy parametrization make our approach scalable in the number of concurrent components.

Our framework has the capability to prove the preservation of not only safety, but additionally liveness properties. Especially in embedded systems in safety critical applications, reactivity or responsiveness (which is a liveness property) is often a mission-critical property and, thus, must be verified.

Our framework is generic. As the instruction schemes can be instantiated, it is applicable for a wide range of realistic instruction sets. Similarly, the memory model can be chosen to fit the target architecture and other communication protocols can be easily integrated into the framework. Thus, our approach is applicable to other languages, architectures and protocols and our framework can be adapted to carry out the verification.

Finally, our framework allows for rigorous verification and the user-interactive part is mechanized, which prevents the introduction of manual errors and allows for (semi-) automation.

8.2 Discussion

In this section, we discuss the achievements and limitations of our approach grouped by criteria we have established in Chapter 1.

Model-based design: Our approach allows for a model-based design process. Our framework naturally extends existing design processes using CSP. In CSP, abstract models can be formalized and verified with respect to desired properties. Within CSP, the models can be refined: more abstract models and more concrete models can be related based on formal (traces and stable failures) refinement. Depending on the refinement model, different types of properties, e. g., safety or liveness, are preserved from the more abstract model to the more concrete model. Both the verification of properties on models as well as the proof of the refinement relation between models can be carried out in FDR4 [GABR14], an automatic refinement checker. Traditionally, the refinement notions of CSP can only relate CSP processes and cannot relates CSP processes with constructs in other languages or formalisms. Our framework extends this model-based design approach from CSP-only to also cover low-level code, with the same guarantees of preservation of properties. Thus, the use of our framework enables a model-based design approach from abstract models in CSP down to executable low-level code.

Low-level code: The instruction schemes of our low-level languages CUC and SV can be instantiated to cover almost any instruction of realistic instruction sets such as RISC-V [WA17]. Only one sub-category of instructions is not covered by SV: Computed jumps. While our framework currently does not support computed jumps, we discuss in Future Work (Section 8.3) how computed jumps can be integrated into the framework. The only reasons to not include computed jumps into our framework are *a)* simplicity and *b)* the possibility for static analysis. The semantics of CUC and SV can be easily extended with computed jumps and the Hoare calculus can also handle computed jumps.

Our framework allows for the verification of shared variable communication, which is commonly used for low-level synchronization and data transfer, as part of a communication protocol. Protection of shared resources is realized in SV with the help of a *compare-and-set* instruction `cas`. With respect to multi-processor synchronization in RISC-V specifically, we do not model *load-reserved* and *store-conditional* as those instructions rely on specific scheduling properties to avoid livelocks. However, the way we use `cas` (i. e., to lock a variable) it can be easily replaced by a *load-reserved* and directly followed by a *store-conditional*.

Our generic function for the "register store" can be instantiated with a memory model suitable for the target architecture; usually with appropriate instantiations of `do` to access different parts of the memory (e. g., load and store instructions). An example for such a memory model for an early precursor of CUC/SV without communication is presented in [BG11].

Preservation of safety and liveness properties: Our approach extends the traces and stable failures models and refinements of CSP to CUC and, with the handshake refinement, also to SV. Thereby, it ensures the preservation of all safety and liveness properties. The safety properties are modeled and ensured in the traces model: unwanted behavior is excluded. The

liveness properties are modeled and ensured in the stable failures model: When no internal transitions are possible, the communication capabilities are ensured. While divergences (i. e., infinite chains of internal transitions) are neither covered by the traces nor the stable failures model, we can ensure divergences freedom in CUC programs for special cases with statical analysis (as we only have conditional branches), e. g., that every loop contains a `comm` instruction. Regarding the relation between CUC and SV programs, the handshake refinement is a strong bisimulation outside of the communication protocol implementation. As such, it preserves divergences (or their absence). The unlocking simulation ensures that the implementation of the communication protocol terminates for successful communication. The only additional divergences in SV programs correspond to deadlocks due to unsuccessful abstract communication, which is the busy-wait pendant to a blocking component. In Section 8.3, we discuss the inclusion of the failures-divergences model of CSP into our framework. This would enable us to talk about divergences directly in the semantics and in the refinements.

Verification of concurrent and communicating systems: Our framework allows for the scalable verification of concurrent and communicating systems: The first part of our framework covers the communication and concurrency aspects of CSP. The second part of our framework also covers the same parts of CSP. However, it currently only covers one verified communication protocol. Thanks to our generic and reusable formalization (and proofs), other communication protocols can be formally captured and verified.

Both parts of our framework scale well with respect to the number of concurrent components. The first part of our framework, where we show a refinement from a CSP process to a CUC program, is compositional. We achieve the compositionality by designing CUC and especially its concurrent composition similar to the concurrent composition of CSP. For homogeneous or parametrized components, it is even sufficient to refine only one instance of each type.

In the second part of our framework, we have shown the preservation of all safety and liveness properties for fitting programs once and for all with the general Theorem 6.4 (Fitting Implies Preservation), as the results of the theorem hold for all systems that comply with the restrictions, i. e., use the given communication protocol. The effort involved to formalize and verify another communication protocol is one-time and upfront.

Non-termination and infinite or very large state spaces: Our approach copes with traces of arbitrary lengths and also states that are parametrized over infinite or very large data types, arising, e. g., from counting occurrences of specific events, tracking time, or uncontrollable input such as reading from a sensor. This is in contrast to approaches like model checking, which rely on the automatic exploration of the state space. The entire framework supports this.

Rigorousness: Our entire framework is based on formal semantics and all results are formally proven. Additionally, to preclude the manual introduction of errors by users of the framework, we have formalized the first part, where the user needs to supply proofs, in the theorem prover Isabelle/HOL [NPW02]. In Isabelle/HOL, user supplied proofs

are mechanically checked by the logical inference system of Isabelle. This helps avoiding overlooking of corner cases and other human errors.

Reusability: The generic nature of our framework makes it reusable for various architectures. The main source of reusability are the instruction schemes of both CUC and SV that can be instantiated according to different target architectures. The function modeling the memory (a simple register store in the bare version of the framework) can also be instantiated according to the target architecture.

For the second part of our framework, we have exemplarily verified a simple handshake protocol as a blueprint. Other protocols can be verified and integrated into our framework: Our language SV allows for the implementation of many realistic communication protocols, illustrated by the fact that the instruction schemes of SV can be instantiated to realistic instruction sets such as RISC-V. Our notion of handshake refinement is adaptable to other protocols (see pointers in Section 6.4). In the second part of our framework, the communication mechanism can also be changed for a different communication mechanism other than shared variables, e. g., communication over networks. While this would require modified semantics for the low-level model (SV) and the handshake refinement, the first part of the framework does not need to be modified for this.

Our formalization of the first part of the framework in the theorem prover Isabelle/HOL allows for the formulation of custom theorems and proof procedures (e. g., specific to an instantiation of the instruction schemes for a concrete architecture). These theorems and proof procedures can be reused for different programs for the same architecture. More general theorems and proof procedures can be reused across multiple architectures, as to be expected from a model-based design approach.

8.3 Future Work

In this section, we discuss opportunities for future work based on our approach. We first investigate the inclusion of semantics that can capture divergences into our framework. This would enable our framework to directly reason about divergences instead of assuming the absence of divergences. We then outline how computed jumps can be included into the framework. This would allow us to verify a broader range of low-level programs. We proceed to point out interesting communication protocols to verify and include in the framework. This would allow us to verify low-level programs that use other synchronization or communication mechanisms. Finally, we propose to integrate data refinements into our approach, which would allow for the additional refinement of abstract data structures such as sets into their low-level implementations.

Divergences: In our framework, we focus on liveness in the sense of stable failures, i. e., which events can be offered for communication in *stable* states. Divergences additionally capture livelocks, i. e., "unstable" behaviors. Divergences are captured in the failures-divergences semantics of CSP, which like all semantical models of CSP are used for a model-based design approach. Capturing divergences consists mainly in tracking a set of reachable divergent states. In low-level languages, every block of code may contain a loop.

Thus, reachability in low-level programs is shown by proving termination of loops or total correctness for a Hoare calculus. The techniques we presented previously in [BJ14] can be used to reason about termination of parts in between communication events in a CUC program. As the handshake refinement is a strong bisimulation outside of the implementation of the communication protocol, related CUC and SV programs have the same divergences, apart from the livelocks in an SV program, which correspond to a deadlock due to unsuccessful communication in the CUC program. With the formalization of divergences, we could obtain even stronger guarantees about communication capabilities right within our framework.

Computed jumps: Computed jumps are part of low-level programs, e. g., for functions calls or addressing of continuous memory such as arrays. Currently, our framework supports only *conditional jumps*, as "raw" computed jumps prevent many useful static analyses.

From a theoretical point of view, conditional jumps allow for the same expressiveness as computed jumps (considering that the entire program is known beforehand). From our point of view, jump targets should be known at compile time, especially in the area of safety critical systems. In practice, computed jumps are used, but often with a limited sets of jump targets in mind, e.g. entry points of functions. This intention can be instrumentalized using the concept of *Control Flow Integrity* (CFI) [ABEL05]. CFI originates in the field of computer security. It was designed to counter control flow altering attacks and basically prefixes every computed jump with a check if the computed jump target is in a list of allowed jump targets. CFI is integrated in current compiler suites, such as LLVM [LA04], and is lately even used in consumer products such as the kernel of the Android operating system. For a recent survey of CFI techniques in production compilers, see [BCN+17]. Our semantics can easily be extended with computed jumps. Our results requiring knowledge of jump targets, e. g., preventing jumps into the middle of a communication protocol, can be preserved when computed jumps are combined with CFI techniques. With the integration of computed jumps into our framework, we would expand the applicability of our framework to a wider range of programs.

Communication protocols: For the second part of our framework, we have verified exemplarily a simple handshake protocol. This covers the common communication scheme of two components communication via a channel at a time. Two other communication protocols would be of special interest, which we discuss in the following. In Section 6.2, we have given pointers on how to adapt our handshake refinement to other protocols. The verification of the protocols themselves belongs to the field of distributed systems, and implementations and proofs can be found there (see, e. g., [Pet12] as a starting point).

The first protocol is often called *select*. It enables a component to concurrently monitor multiple channels. Depending on the implementation, it allows for a program to wait to synchronously send on one channel and at the same time to wait to synchronously receive on another channel (which is called mixed choice in process calculi). The *select* protocol is supported in CSP via external choice, and implemented in languages with first class support for communication over channels, such as Ada [TDB+13], which is used in safety-critical areas such as avionics and automotive, or Go [DK15], which is developed by Google and used for large scale server applications.

The second protocol is the multi-way synchronization of CSP. It is a form of barrier synchronization of multiple components, which is used in distributed algorithms using shared resources, e. g., matrix multiplications. As inspiration for the formalization of the multi-way synchronization protocol see [PS92]. In a Bachelor's thesis [Sac15], we have implemented multi-way synchronization for CUC in LLVM using the pthreads library [NBF96] for concurrency and synchronization.

Formalizing the mentioned communication protocols would enable a greater flexibility in implementing high-level specifications while still preserving safety and liveness properties.

Data refinement approaches: Refinements relate abstract and concrete versions of different aspects of a model:

a) They can resolve non-determinisms as in CSP.
b) They can implement abstract actions with more concrete actions. This is often called *action refinement* and overviews of different approaches can be found in [GR01, BvW98].
c) They can refine abstract data structures into the data structures actually needed (bounded integers instead of all integers) or the actual implementation (lists instead of sets). This is often called data refinement and an overview can be found in [RE08].

The resolution of non-determinism (a) is the core idea of the CSP refinements. To split up the abstract communication instructions into the multiple instruction forming the communication protocol, we used an idea similar to action refinement (b) from [RG97]. So far, we have not integrated data refinements (c) into our framework.

The combination or integration of data refinements with our framework would close the final gap between abstract specifications and concrete implementations: In the specification mathematically convenient data types such as integers or sets could be used, and in the implementation their low-level equivalent, while providing formal means to justify the change of data type.

Appendix

A.1 Correspondence Proofs

In this section, we prove the concurrent case of correspondence theorems between the operational semantics and each of the traces and the stable failures semantics. We first show the concurrent case of the theorem concerning the traces semantics of CUC in Subsection A.1.1 and then extend the proof for the theorem concerning the stable failures semantics of CUC in Subsection A.1.2.

A.1.1 Proof: Concurrent Case of Correspondence of Traces

In this section, we prove the concurrent case of correspondence theorems between the operational semantics and each of the traces and the stable failures semantics. For convenience, we recall the Theorem 5.1:

Theorem 5.1: Correspondence Between Operational Characterization and Traces Semantics

$$tr \in \mathcal{T}_{\mathcal{U}(scp)}(\sigma) \Leftrightarrow \exists\, \sigma'.\ (tr, \sigma') \in [\![scp]\!]^{\mathcal{T}}\big(\{(\langle\rangle, \sigma)\}\big)$$

The overall proof of Theorem 5.1 is an induction on the structure of *scp*. The (syntactically simplified version of the) structure of *scp* is depicted in the following.

$$scp = \texttt{do} \mid \texttt{cbr} \mid \texttt{comm} \mid sp_1 \oplus sp_2 \mid scp_1\ {}_{\alpha_1}\|_{\alpha_2}\ scp_2$$

There are five cases for the proof: Three for the single instructions `do`, `cbr`, and `comm`, one for the sequential composition $\oplus$ and one for the concurrent composition $\|$. Here, we consider the concurrent case. We have proven the other cases in Isabelle/HOL. For convenience, we recall the relevant definitions of the concurrent cases in Figure A.1. Definition 5.13 of the operational characterization of the traces semantics of CUC allows us to use the operational semantics and the operational characterization of the trace semantics of CUC interchangeably. We prove both implications (necessary $\Longrightarrow$ and sufficient $\Longleftarrow$) of the equivalence separately and start with the necessary implication.

Proof: $tr \in \mathcal{T}_{\mathcal{U}(scp)}(\sigma) \Longrightarrow \exists\, \sigma'.\ (tr, \sigma') \in [\![scp]\!]^{\mathcal{T}}\big(\{(\langle\rangle, \sigma)\}\big)$

Assumptions:
The concurrent structure of the program is given by the overall induction on the structure of *scp* (I). From the same tree structure of programs and states (Assumption 5.3), we can assume the components of the state (II). From the induction we can assume the hypothesis for the components scp_1 and scp_2, i. e., III) and IV). From the hypothesis to show, we

SYNC

$$\frac{\sigma_1 \xrightarrow{ev}_{cp_1} \sigma_1' \qquad \sigma_2 \xrightarrow{ev}_{cp_2} \sigma_2' \qquad ev \in \alpha_1 \cap \alpha_2}{\sigma_1 \parallel \sigma_2 \xrightarrow{ev}_{\left(cp_1 {}_{\alpha_1}\|_{\alpha_2} cp_2\right)} \sigma_1' \parallel \sigma_2'}$$

EXEC-EV

$$\frac{\sigma \overset{tr'}{\Longrightarrow}_{cp} \sigma'' \quad \sigma'' \xrightarrow{ev}_{cp} \sigma' \quad tr' \frown \langle ev \rangle = tr}{\sigma \overset{tr}{\Longrightarrow}_{cp} \sigma'}$$

INTERLEAVING-LEFT

$$\frac{\sigma_1 \xrightarrow{ev}_{cp_1} \sigma_1' \qquad ev \in (\alpha_1 \cup \{\tau\}) \setminus \alpha_2}{\sigma_1 \parallel \sigma_2 \xrightarrow{ev}_{\left(cp_1 {}_{\alpha_1}\|_{\alpha_2} cp_2\right)} \sigma_1' \parallel \sigma_2}$$

INTERLEAVING-RIGHT

$$\frac{\sigma_2 \xrightarrow{ev}_{cp_2} \sigma_2' \qquad ev \in (\alpha_2 \cup \{\tau\}) \setminus \alpha_1}{\sigma_1 \parallel \sigma_2 \xrightarrow{ev}_{\left(cp_1 {}_{\alpha_1}\|_{\alpha_2} cp_2\right)} \sigma_1 \parallel \sigma_2'}$$

$\mathcal{T}$-PAR

$$\begin{aligned}
[\![scp_1 \ {}_{\alpha_1}\|_{\alpha_2} \ scp_2]\!]^{\mathcal{T}}(S) = \big\{(tr, \sigma_1' \parallel \sigma_2') \mid\ & (\langle\rangle, \sigma_1 \parallel \sigma_2) \in S \wedge \\
& (tr \upharpoonright \alpha_1, \sigma_1') \in [\![scp_1]\!]^{\mathcal{T}}\big(\{(\langle\rangle, \sigma_1)\}\big) \wedge \\
& (tr \upharpoonright \alpha_2, \sigma_2') \in [\![scp_2]\!]^{\mathcal{T}}\big(\{(\langle\rangle, \sigma_2)\}\big) \wedge \\
& set(tr) \subseteq (\alpha_1 \cup \alpha_2)\big\}
\end{aligned}$$

Figure A.1: Relevant Definitions for the Concurrent Cases

Proof: $tr \in \mathcal{T}_{\mathcal{U}(scp)}(\sigma) \Longrightarrow \exists\, \sigma'.\ (tr, \sigma') \in [\![scp]\!]^{\mathcal{T}}\big(\{(\langle\rangle, \sigma)\}\big)$

can assume the premise (V).

I) $scp = scp_1 \ {}_{\alpha_1}\|_{\alpha_2} \ scp_2$
II) $\sigma = \sigma_1 \parallel \sigma_2$
III) $\forall\, tr_1.\ tr_1 \in \mathcal{T}_{\mathcal{U}(scp_1)}(\sigma_1) \longrightarrow \exists\, \sigma_1'.\ (tr_1, \sigma_1') \in [\![scp_1]\!]^{\mathcal{T}}\big(\{(\langle\rangle, \sigma_1)\}\big)$
IV) $\forall\, tr_2.\ tr_2 \in \mathcal{T}_{\mathcal{U}(scp_2)}(\sigma_2) \longrightarrow \exists\, \sigma_2'.\ (tr_2, \sigma_2') \in [\![scp_2]\!]^{\mathcal{T}}\big(\{(\langle\rangle, \sigma_2)\}\big)$
V) $tr \in \mathcal{T}_{\mathcal{U}(scp)}(\sigma)$

Want to show (goal):
$\exists\, \sigma'.\ (tr, \sigma') \in [\![scp]\!]^{\mathcal{T}}\big(\{(\langle\rangle, \sigma)\}\big)$

Proof:
Using I) and $\mathcal{T}$-PAR we can reformulate the goal:

$$\begin{aligned}
\exists\, \sigma'.\ (tr, \sigma') \in \big\{(tr, \sigma_1' \parallel \sigma_2') \mid\ & \\
& (\langle\rangle, \sigma_1 \parallel \sigma_2) \in \{(\langle\rangle, \sigma)\} \wedge && (\mathcal{T}.1) \\
& (tr \upharpoonright \alpha_1, \sigma_1') \in [\![scp_1]\!]^{\mathcal{T}}\big(\{(\langle\rangle, \sigma_1)\}\big) \wedge && (\mathcal{T}.2) \\
& (tr \upharpoonright \alpha_2, \sigma_2') \in [\![scp_2]\!]^{\mathcal{T}}\big(\{(\langle\rangle, \sigma_2)\}\big) \wedge && (\mathcal{T}.3) \\
& set(tr) \subseteq (\alpha_1 \cup \alpha_2)\big\} && (\mathcal{T}.4)
\end{aligned}$$

We show the four properties. $\mathcal{T}.1$ is directly satisfied by II).

$\mathcal{T}.2$:
We show $tr \upharpoonright \alpha_1 \in \mathcal{T}_{\mathcal{U}(scp_1)}(\sigma_1)$. That lets us apply III) which yields us $\mathcal{T}.2$.

Proof: $tr \in \mathcal{T}_{\mathcal{U}(scp)}(\sigma) \Longrightarrow \exists\, \sigma'.\ (tr, \sigma') \in [\![scp]\!]^{\mathcal{T}}\big(\{(\langle\rangle, \sigma)\}\big)$

We show this by induction on the trace tr.
$\underline{\mathcal{T}.2.\text{base } tr = \langle\rangle}$: According to Property T1, all traces semantics contain the empty trace.
$\Longrightarrow \langle\rangle \upharpoonright \alpha_1 = \langle\rangle \in \mathcal{T}_{\mathcal{U}(scp_1)}(\sigma_1)$
$\underline{\mathcal{T}.2.\text{step } tr = \widetilde{tr}^\frown\langle ev\rangle}$:
$\widetilde{tr} \upharpoonright \alpha_1 \in \mathcal{T}_{\mathcal{U}(scp_1)}(\sigma_1)$ holds by induction hypothesis.
We consider two cases for ev. $ev \notin \alpha_1 \vee ev \in \alpha_1$.
$\underline{\mathcal{T}.2.\text{step.1 } ev \notin \alpha_1}$: ev is removed by the projection.
$\Longrightarrow \widetilde{tr}^\frown\langle ev\rangle \upharpoonright \alpha_1 = \widetilde{tr} \upharpoonright \alpha_1 \in \mathcal{T}_{\mathcal{U}(scp_1)}(\sigma_1)$
$\underline{\mathcal{T}.2.\text{step.2 } ev \in \alpha_1}$: Either $ev \in \alpha_2 \vee ev \notin \alpha_2$ holds. In both cases, we obtain a semantical step in $\mathcal{U}(scp_1)$ that is labeled with ev (from V) and the rule SYNC or INTERLEAVING-LEFT, respectively. Using rule EXEC-EV, we can append ev to $\widetilde{tr} \upharpoonright \alpha_1$. By the Definition 5.13 of $\mathcal{T}$ that implies $(\widetilde{tr} \upharpoonright \alpha_1)^\frown\langle ev\rangle \in \mathcal{T}_{\mathcal{U}(scp_1)}(\sigma_1)$. As $ev \notin \alpha_1$, we can move ev on either side of the projection.
$(\widetilde{tr} \upharpoonright \alpha_1)^\frown\langle ev\rangle \in \mathcal{T}_{\mathcal{U}(scp_1)}(\sigma_1) \Longrightarrow \widetilde{tr}^\frown\langle ev\rangle \upharpoonright \alpha_1 \in \mathcal{T}_{\mathcal{U}(scp_1)}(\sigma_1)$

$\underline{\mathcal{T}.3}$: Analogous to $\mathcal{T}.2$.

$\underline{\mathcal{T}.4}$: Similar to the proof for $\mathcal{T}.2$ we can show with an induction on the trace tr that all its elements are from $\alpha_1 \cup \alpha_2$.

We can now instantiate $\sigma' = \sigma_1' \parallel \sigma_2'$ with σ_1', σ_2' obtained in the proofs for $\mathcal{T}.2$ and $\mathcal{T}.3$, respectively. Thus, all traces that occur in the operationally characterized traces semantics of the concurrent combination of two components occur also in the denotational traces semantics of the combined components. □

We continue and show the sufficient implication.

Proof: $\exists\, \sigma'.\ (tr, \sigma') \in [\![scp]\!]^{\mathcal{T}}\big(\{(\langle\rangle, \sigma)\}\big) \Longrightarrow tr \in \mathcal{T}_{\mathcal{U}(scp)}(\sigma)$

Assumptions:
The concurrent structure of the program is given by the overall induction on the structure of scp (I). From the same tree structure of programs and states (Assumption 5.3), we can assume the components of the state (II). From the induction we can assume the hypothesis for the components scp_1 and scp_2, i. e., III) and IV). From the hypothesis to show, we can assume the premise, which gives us V)-VII) from $\mathcal{T}$-PAR.

I) $scp = scp_1 \ {}_{\alpha_1}\|_{\alpha_2}\ scp_2$
II) $\sigma = \sigma_1 \parallel \sigma_2$
III) $\forall\, tr_1.\, \exists\, \sigma_1'.\ (tr_1, \sigma_1') \in [\![scp_1]\!]^{\mathcal{T}}\big(\{(\langle\rangle, \sigma_1)\}\big) \Longrightarrow tr_1 \in \mathcal{T}_{\mathcal{U}(scp_1)}(\sigma_1)$
IV) $\forall\, tr_2.\, \exists\, \sigma_2'.\ (tr_2, \sigma_2') \in [\![scp_2]\!]^{\mathcal{T}}\big(\{(\langle\rangle, \sigma_2)\}\big) \Longrightarrow tr_2 \in \mathcal{T}_{\mathcal{U}(scp_2)}(\sigma_2)$
V) $(tr \upharpoonright \alpha_1, \sigma_1') \in [\![scp_1]\!]^{\mathcal{T}}\big(\{(\langle\rangle, \sigma_1)\}\big)$
VI) $(tr \upharpoonright \alpha_2, \sigma_2') \in [\![scp_2]\!]^{\mathcal{T}}\big(\{(\langle\rangle, \sigma_2)\}\big)$
VII) $set(tr) \subseteq (\alpha_1 \cup \alpha_2)$

Proof: $\exists\, \sigma'.\ (tr, \sigma') \in [\![scp]\!]^{\mathcal{T}}\big(\{(\langle\rangle, \sigma)\}\big) \Longrightarrow tr \in \mathcal{T}_{\mathcal{U}(scp)}(\sigma)$

Want to show (goal):
$tr \in \mathcal{T}_{\mathcal{U}(scp)}(\sigma)$

Proof:
We show the goal by induction on the trace tr.

Base $tr = \langle\rangle$: According to Property T1, all traces semantics contain the empty trace.
$\Longrightarrow \langle\rangle \in \mathcal{T}_{\mathcal{U}(scp)}(\sigma)$

Step $tr = \widetilde{tr}^\frown\langle ev\rangle$:
$\widetilde{tr} \in \mathcal{T}_{\mathcal{U}(scp)}(\sigma)$ holds by induction hypothesis.
With VII) we have $set(\widetilde{tr}^\frown\langle ev\rangle) = set(tr) \subseteq (\alpha_1 \cup \alpha_2) \Longrightarrow ev \in (\alpha_1 \cup \alpha_2)$
We consider three cases for ev. $ev \in \alpha_1 \setminus \alpha_2 \vee ev \in \alpha_2 \setminus \alpha_1 \vee ev \in \alpha_1 \cap \alpha_2$.
Step.1 $ev \in \alpha_1 \setminus \alpha_2$:
From $ev \in \alpha_1$ we have $(\widetilde{tr} \upharpoonright \alpha_1)^\frown\langle ev\rangle = (\widetilde{tr}^\frown\langle ev\rangle) \upharpoonright \alpha_1$.
From V) and III) we have $(tr \upharpoonright \alpha_1) \in \mathcal{T}_{\mathcal{U}(scp_1)}(\sigma_1)$.
With both facts, we obtain $(\widetilde{tr} \upharpoonright \alpha_1)^\frown\langle ev\rangle \in \mathcal{T}_{\mathcal{U}(scp_1)}(\sigma_1)$.
That gives us an ev-step in scp_1 after the trace $\widetilde{tr} \upharpoonright \alpha_1$. That lets us apply rule INTERLEAVING-LEFT after the trace $\widetilde{tr}$, which in turn yields $\widetilde{tr}^\frown\langle ev\rangle \in \mathcal{T}_{\mathcal{U}(scp)}(\sigma)$
Step.2 $ev \in \alpha_2 \setminus \alpha_1$: Analogous to Step.1.
Step.3 $ev \in \alpha_1 \cap \alpha_2$: Similar to Step.1 and Step.2. However this time we have ev-steps in both scp_1 and scp_2 and apply rule SYNC.

Thus, all traces that occur in the denotational traces semantics of the concurrent combination of two components occur also in the operationally characterized traces semantics of the combined components. □

Together, the necessary and the sufficient implications show the correspondence between the operationally characterized traces semantics and the denotational traces semantics. In the next subsection, we extend the proof and show the correspondence between the operationally characterized stable failures semantics and the denotational stable failures semantics.

A.1.2 Proof: Concurrent Case of Correspondence of Stable Failures

In this subsection, we prove the concurrent case of Theorem 5.2. For convenience, we recall Theorem 5.2:

Theorem 5.2: Correspondence Between Operational Characterization and Stable Failures Semantics *

$$(tr, X) \in \mathcal{SF}_{\mathcal{U}(scp)}(\sigma) \Leftrightarrow \exists\, \sigma'.\ (tr, \sigma', X) \in [\![scp]\!]^{\mathcal{SF}}\big(\{(\langle\rangle, \sigma, Y) \mid Y \subseteq \Sigma\}\big)$$

The proof extends the proof of the correspondence between the two different traces semantics of CUC from the previous subsection. Here, we consider the concurrent case. We have proven the other cases in Isabelle/HOL. The main difference between the proof for the correspondence of the traces and the correspondence of the stable failures is the treatment of the refusal sets. We refer back to the proof in the last subsection for common proof steps and focus on the new proof obligations. For convenience, we recall Definition 5.16 of the operational characterization of the stable failures semantics and the $\mathcal{SF}$-PAR rule in Figure A.2.

Definition 5.16: Operational Characterization of the Stable Failures of CUC

A **stable failure** of a CUC program *cuc* is a pair of a trace *tr* and a refusal set X. It denotes that there is a stable state σ' which can be reached from the initial state σ via the trace *tr* and refuses X.

$$(tr, X) \in \mathcal{SF}_{cuc}(\sigma) := \exists\, \sigma'.\ \sigma \xRightarrow{tr}_{cuc} \sigma' \wedge \sigma' \downarrow_{cuc} \wedge\ \sigma' \operatorname{ref}_{cuc} X$$

$\mathcal{SF}$-PAR

$$\begin{aligned}[\![scp_1\ {}_{\alpha_1}\|_{\alpha_2}\ scp_2]\!]^{\mathcal{SF}}(S) = \big\{(tr, \sigma_1' \parallel \sigma_2', X) \,\big|\, \exists\, X_1\ X_2.\ (\langle\rangle, \sigma_1 \parallel \sigma_2, \_) \in S \wedge \\ X \cap (\alpha_1 \cup \alpha_2) = (X_1 \cap \alpha_1) \cup (X_2 \cap \alpha_2) \wedge \\ (tr \upharpoonright\ \alpha_1, \sigma_1', X_1) \in [\![scp_1]\!]^{\mathcal{SF}}\big(\{(\langle\rangle, \sigma_1, \_)\}\big) \wedge \\ (tr \upharpoonright\ \alpha_2, \sigma_2', X_2) \in [\![scp_2]\!]^{\mathcal{SF}}\big(\{(\langle\rangle, \sigma_2, \_)\}\big) \wedge \\ set(tr) \subseteq (\alpha_1 \cup \alpha_2)\big\}\end{aligned}$$

Figure A.2: Recall the Concurrent Combination of $\mathcal{SF}$

We prove both implications (necessary $\Longrightarrow$ and sufficient $\Longleftarrow$) of the equivalence separately. Before we start with the necessary implication, we show a lemma about the relation of refusal sets of the components and the refusal sets of the combination of the components, which we need in the following proof.

Lemma A.1: Relation of Refusal Sets of the Components and the Combination

$(\sigma_1 \parallel \sigma_2) \operatorname{ref}_{(cp_1\,{}_{\alpha_1}\|_{\alpha_2}\,cp_2)} X$
$\Leftrightarrow \exists\, X_1\ X_2.\ \sigma_1 \operatorname{ref}_{cp_1} X_1 \wedge \sigma_2 \operatorname{ref}_{cp_2} X_2 \wedge X \cap (\alpha_1 \cup \alpha_2) = (X_1 \cap \alpha_1) \cup (X_2 \cap \alpha_2)$

Proof: Relation of Refusal Sets of the Components and the Combination

We first show a version that is more intuitive to prove and then use set theory to conclude the lemma.

Proof: Relation of Refusal Sets of the Components and the Combination

Want to show (goal):

1) $(\sigma_1 \parallel \sigma_2)\ \mathrm{ref}_{(cp_{1\alpha_1} \parallel_{\alpha_2} cp_2)}\ X$
$\Leftrightarrow \exists X_1\ X_2.\, \sigma_1\ \mathrm{ref}_{cp_1}\ X_1 \wedge \sigma_2\ \mathrm{ref}_{cp_2}\ X_2 \wedge$
$X \cap (\alpha_1 \cup \alpha_2) = \big(X_1 \setminus \alpha_2 \cup X_2 \setminus \alpha_1 \cup (X_1 \cup X_2) \cap (\alpha_1 \cap \alpha_2)\big) \cap (\alpha_1 \cup \alpha_2)$
2) $\big(X_1 \setminus \alpha_2 \cup X_2 \setminus \alpha_1 \cup (X_1 \cup X_2) \cap (\alpha_1 \cap \alpha_2)\big) \cap (\alpha_1 \cup \alpha_2) = (X_1 \cap \alpha_1) \cup (X_2 \cap \alpha_2)$

1) $\Rightarrow$:
Consider $ev \in X$. We show that we can choose X_1, X_2 with $\sigma_1\ \mathrm{ref}_{cp_1}\ X_1$ and $\sigma_2\ \mathrm{ref}_{cp_2}\ X_2$ satisfying

$$X \subseteq X_1 \setminus \alpha_2 \cup X_2 \setminus \alpha_1 \cup (X_1 \cup X_2) \cap (\alpha_1 \cap \alpha_2)$$

Containment ($\subseteq$) is enough, as refusal sets are subset-closed (Property SF3), so we can always choose smaller X_1, X_2 to obtain equality.
As the combination $\sigma_1 \parallel \sigma_2$ refuses ev, none of the three rules INTERLEAVING-LEFT, INTERLEAVING-RIGHT and SYNC may apply. We use this to show that every $ev \in X$ must also be an element of the right-hand side.
INTERLEAVING-LEFT:
The combination can refuse any event that is not in the communication interface of the right component ($ev \notin \alpha_2$) and refused by the left ($ev \in X_1$).
$\Longrightarrow ev \in X_1 \setminus \alpha_2$
INTERLEAVING-RIGHT:
Analogous to INTERLEAVING-LEFT.
$\Longrightarrow ev \in X_2 \setminus \alpha_1$
SYNC:
The combination can refuse any event that is in both communication interface $ev \in (\alpha_1 \cap \alpha_2)$. The components can refuse events from their respective refusal sets, i. e., $ev \in (X_1 \cup X_2)$.
$\Longrightarrow ev \in (X_1 \cup X_2) \cap (\alpha_1 \cap \alpha_2)$

We can choose X_1, X_2 small enough, so that $\subseteq$ becomes $=$ according to Property SF3 and we can cut both sides with $\alpha_1 \cup \alpha_2$.

---

1) $\Leftarrow$:
Consider X_1, X_2 with $\sigma_1\ \mathrm{ref}_{cp_1}\ X_1$ and $\sigma_2\ \mathrm{ref}_{cp_2}\ X_2$. Let X with

$$X \cap (\alpha_1 \cup \alpha_2) = \big(X_1 \setminus \alpha_2 \cup X_2 \setminus \alpha_1 \cup (X_1 \cup X_2) \cap (\alpha_1 \cap \alpha_2)\big) \cap (\alpha_1 \cup \alpha_2)$$

and $ev \in X$. We use a case distinction on ev to show that none of the rules INTERLEAVING-LEFT, INTERLEAVING-RIGHT and SYNC may be applied, i. e., $(\sigma_1 \parallel \sigma_2)\ \mathrm{ref}_{(cp_{1\alpha_1} \parallel_{\alpha_2} cp_2)}\ X$.

$ev \in (\alpha_1 \cup \alpha_2)$:

Proof: Relation of Refusal Sets of the Components and the Combination

Refusal sets never contain the invisible event τ. Thus, $ev \neq \tau$. None of the concurrent rules apply. $\Longrightarrow (\sigma_1 \parallel \sigma_2)\ \mathrm{ref}_{(cp_{1\alpha_1} \parallel_{\alpha_2} cp_2)}\ \{ev\}$

<u>$ev \in X_1 \setminus \alpha_2$:</u>
Only the rule INTERLEAVING-LEFT could be applied. As the left component refuses ev, the combination refuses ev, too. $\Longrightarrow (\sigma_1 \parallel \sigma_2)\ \mathrm{ref}_{(cp_{1\alpha_1} \parallel_{\alpha_2} cp_2)}\ \{ev\}$

<u>$ev \in X_2 \setminus \alpha_1$:</u>
Analogous to the previous case.

<u>$ev \in (X_1 \cup X_2) \cap (\alpha_1 \cap \alpha_2)$:</u>
Only rule SYNC could be applied. As one of the components refuses ev, the combination refuses ev, too. $\Longrightarrow (\sigma_1 \parallel \sigma_2)\ \mathrm{ref}_{(cp_{1\alpha_1} \parallel_{\alpha_2} cp_2)}\ \{ev\}$

As we have $(\sigma_1 \parallel \sigma_2)\ \mathrm{ref}_{(cp_{1\alpha_1} \parallel_{\alpha_2} cp_2)}\ \{ev\}$ for all $ev \in X$, we can conclude $(\sigma_1 \parallel \sigma_2)\ \mathrm{ref}_{(cp_{1\alpha_1} \parallel_{\alpha_2} cp_2)}\ X$.

<u>2):</u>
We use set equalities to show 2).

$$
\begin{aligned}
&\big(X_1 \setminus \alpha_2 \cup X_2 \setminus \alpha_1 \cup (X_1 \cup X_2) \cap (\alpha_1 \cap \alpha_2)\big) \cap (\alpha_1 \cup \alpha_2) \\
&= (X_1 \setminus \alpha_2) \cap (\alpha_1 \cup \alpha_2) \cup (X_2 \setminus \alpha_1) \cap (\alpha_1 \cup \alpha_2) \cup \big((X_1 \cup X_2) \cap (\alpha_1 \cap \alpha_2)\big) \cap (\alpha_1 \cup \alpha_2) \\
&= (X_1 \setminus \alpha_2) \cap \alpha_1 \cup (X_2 \setminus \alpha_1) \cap \alpha_2 \cup (X_1 \cup X_2) \cap (\alpha_1 \cap \alpha_2) \\
&= (X_1 \setminus \alpha_2) \cap \alpha_1 \cup (X_2 \setminus \alpha_1) \cap \alpha_2 \cup X_1 \cap (\alpha_1 \cap \alpha_2) \cup X_2 \cap (\alpha_1 \cap \alpha_2) \\
&= (X_1 \setminus \alpha_2) \cap \alpha_1 \cup X_1 \cap (\alpha_1 \cap \alpha_2) \cup (X_2 \setminus \alpha_1) \cap \alpha_2 \cup X_2 \cap (\alpha_1 \cap \alpha_2) \\
&= (X_1 \cap \alpha_1) \cup (X_2 \cap \alpha_2)
\end{aligned}
$$

From 1) and 2) we conclude Lemma A.1. □

Having established the relationship of refusal sets of the components and the combination, we can prove the necessary implication of the correspondence proof for the stable failure semantics.

Proof: $(tr, X) \in \mathcal{SF}_{\mathcal{U}(scp)}(\sigma) \Longrightarrow \exists\, \sigma'.\ (tr, \sigma', X) \in [\![scp]\!]^{\mathcal{SF}}\big(\{(\langle\rangle, \sigma, Y) \mid Y \subseteq \Sigma\}\big)$

<u>Assumptions:</u>
The concurrent structure of the program is given by the overall induction on the structure of scp (I). From the same tree structure of programs and states (Assumption 5.3), we can assume the components of the state (II). From the induction we can assume the hypothesis for the components scp_1 and scp_2, i. e., III) and IV). From the hypothesis to show, we

Proof: $(tr, X) \in \mathcal{SF}_{\mathcal{U}(scp)}(\sigma) \Longrightarrow \exists\, \sigma'.\ (tr, \sigma', X) \in [\![scp]\!]^{\mathcal{SF}}\big(\{(\langle\rangle, \sigma, Y) \mid Y \subseteq \Sigma\}\big)$

can assume the premise (V). Unfolding Definition 5.16, we get VI).

I) $scp = scp_1\ {}_{\alpha_1}\|_{\alpha_2}\ scp_2$

II) $\sigma = \sigma_1 \parallel \sigma_2$

III) $\forall\, tr_1\ X_1.\ (tr_1, X_1) \in \mathcal{SF}_{\mathcal{U}(scp_1)}(\sigma_1)$
$\Longrightarrow \exists\, \sigma_1'.\ (tr_1, \sigma_1', X_1) \in [\![scp_1]\!]^{\mathcal{SF}}\big(\{(\langle\rangle, \sigma_1, Y) \mid Y \subseteq \Sigma\}\big)$

IV) $\forall\, tr_2\ X_2.\ (tr_2, X_2) \in \mathcal{SF}_{\mathcal{U}(scp_2)}(\sigma_2)$
$\Longrightarrow \exists\, \sigma_2'.\ (tr_2, \sigma_2', X_2) \in [\![scp_2]\!]^{\mathcal{SF}}\big(\{(\langle\rangle, \sigma_2, Y) \mid Y \subseteq \Sigma\}\big)$

V) $(tr, X) \in \mathcal{SF}_{\mathcal{U}(scp)}(\sigma)$

VI) $\exists\, \sigma'.\ \sigma \overset{tr}{\Longrightarrow}_{\mathcal{U}(scp)} \sigma' \wedge \sigma'\!\downarrow_{\mathcal{U}(scp)} \wedge\ \sigma'\ \mathrm{ref}_{\mathcal{U}(scp)}\ X$

<u>Want to show (goal):</u>
$\exists\, \sigma'.\ (tr, \sigma', X) \in [\![scp]\!]^{\mathcal{SF}}\big(\{(\langle\rangle, \sigma, Y) \mid Y \subseteq \Sigma\}\big)$

<u>Proof:</u>
Using I) and $\mathcal{SF}$-PAR we can reformulate the goal:

$$\begin{aligned}
\exists\, \sigma'.\ (tr, \sigma', X) \in \big\{(tr, \sigma_1' \parallel \sigma_2', X) \,\big|\, \exists\, X_1\ X_2.\ & \\
& (\langle\rangle, \sigma_1 \parallel \sigma_2, _) \in \{(\langle\rangle, \sigma, Y) \mid Y \subseteq \Sigma\} \wedge && (\mathcal{SF}.1)\\
& X \cap (\alpha_1 \cup \alpha_2) = (X_1 \cap \alpha_1) \cup (X_2 \cap \alpha_2) \wedge && (\mathcal{SF}.2)\\
& (tr \upharpoonright \alpha_1, \sigma_1', X_1) \in [\![scp_1]\!]^{\mathcal{SF}}\big(\{(\langle\rangle, \sigma_1, _)\}\big) \wedge && (\mathcal{SF}.3)\\
& (tr \upharpoonright \alpha_2, \sigma_2', X_2) \in [\![scp_2]\!]^{\mathcal{SF}}\big(\{(\langle\rangle, \sigma_2, _)\}\big) \wedge && (\mathcal{SF}.4)\\
& set(tr) \subseteq (\alpha_1 \cup \alpha_2)\big\} && (\mathcal{SF}.5)
\end{aligned}$$

We show the five properties.

<u>$\mathcal{SF}$.1:</u>
is directly satisfied by II) as refusal sets are always a subset of Σ.

<u>$\mathcal{SF}$.2:</u>
From VI) we have $(\sigma_1' \parallel \sigma_2')\ \mathrm{ref}_{(scp_1\,{}_{\alpha_1}\|_{\alpha_2}\,scp_2)}\ X$. Using Lemma A.1, we show $\mathcal{SF}$.2 and get $\sigma_1'\ \mathrm{ref}_{scp_1}\ X_1$ and $\sigma_2'\ \mathrm{ref}_{scp_2}\ X_2$.
Strictly speaking, we obtain σ_1', σ_2' in the proof of $\mathcal{SF}$.3 and this proof needs to be a part of it. But for the sake of readability and to prove the goals in order, we prepone the argument for $\mathcal{SF}$.2.

<u>$\mathcal{SF}$.3:</u>
We show $(tr \upharpoonright \alpha_1, X_1) \in \mathcal{SF}_{\mathcal{U}(scp_1)}(\sigma_1)$. This yields, together with the Assumption 5.6 about initial failures that lets us apply III), $\mathcal{SF}$.3.
According to Definition 5.16, we have to show:
$\exists\, \sigma_1'.\ \sigma_1 \overset{tr\upharpoonright\alpha_1}{\Longrightarrow}_{\mathcal{U}(scp_1)} \sigma_1' \wedge \sigma_1'\!\downarrow_{\mathcal{U}(scp_1)} \wedge\ \sigma_1'\ \mathrm{ref}_{\mathcal{U}(scp_1)}\ X_1$.

Proof: $(tr, X) \in \mathcal{SF}_{\mathcal{U}(scp)}(\sigma) \Longrightarrow \exists\, \sigma'.\ (tr, \sigma', X) \in [\![scp]\!]^{\mathcal{SF}}\big(\{(\langle\rangle, \sigma, Y) \mid Y \subseteq \Sigma\}\big)$

$\underline{\sigma_1 \overset{tr\restriction\alpha_1}{\Longrightarrow}_{\mathcal{U}(scp_1)} \sigma_1'}$:
Similar to the proof for $\mathcal{T}.2$ in the previous subsection.
$\underline{\sigma_1' \downarrow_{\mathcal{U}(scp_1)}}$:
If the combination is stable, thus, cannot engage in τ events, neither of the component can engage in τ events. Thus, the components are also stable themselves.
$\underline{\sigma_1' \text{ ref}_{\mathcal{U}(scp_1)} X_1}$:
We obtained this in the proof of $\mathcal{SF}.2$ from Lemma A.1.

<u>$\mathcal{SF}.4$:</u> Analogous to $\mathcal{SF}.3$.

<u>$\mathcal{SF}.5$:</u> Similar to the proof for $\mathcal{SF}.3$, we can show with an induction on the trace tr that all its elements are from $(\alpha_1 \cup \alpha_2)$.

We can now instantiate $\sigma' = \sigma_1' \parallel \sigma_2'$ with σ_1', σ_2' obtained in the proofs for $\mathcal{SF}.3$ and $\mathcal{SF}.4$, respectively. Thus, all stable failures that occur in the operationally characterized stable failures semantics of the concurrent combination of two components occur also in the denotational stable failures semantics of the combined components. □

We continue and show the sufficient implication.

Proof: $\exists\, \sigma'.\ (tr, \sigma', X) \in [\![scp]\!]^{\mathcal{SF}}\big(\{(\langle\rangle, \sigma, Y) \mid Y \subseteq \Sigma\}\big) \Longrightarrow (tr, X) \in \mathcal{SF}_{\mathcal{U}(scp)}(\sigma)$

<u>Assumptions:</u>
The concurrent structure of the program is given by the overall induction on the structure of scp (I). From the same tree structure of programs and states (Assumption 5.3), we can assume the components of the state (II). From the induction we can assume the hypothesis for the components scp_1 and scp_2, i. e., III) and IV). From the hypothesis to show, we can assume the premise, which gives us the V)-VIII) from $\mathcal{SF}$-PAR.

I) $scp = scp_1 \ {}_{\alpha_1}\|_{\alpha_2}\ scp_2$
II) $\sigma = \sigma_1 \parallel \sigma_2$
III) $\forall\, tr_1.\, \exists\, \sigma_1'.\ (tr_1, \sigma_1', X_1) \in [\![scp_1]\!]^{\mathcal{SF}}\big(\{(\langle\rangle, \sigma_1, Y) \mid Y \subseteq \Sigma\}\big)$
$\Longrightarrow (tr_1, X_1) \in \mathcal{SF}_{\mathcal{U}(scp_1)}(\sigma_1)$
IV) $\forall\, tr_2.\, \exists\, \sigma_2'.\ (tr_2, \sigma_2', X_2) \in [\![scp_2]\!]^{\mathcal{SF}}\big(\{(\langle\rangle, \sigma_2, Y) \mid Y \subseteq \Sigma\}\big)$
$\Longrightarrow (tr_2, X_2) \in \mathcal{SF}_{\mathcal{U}(scp_2)}(\sigma_2)$
V) $X \cap (\alpha_1 \cup \alpha_2) = (X_1 \cap \alpha_1) \cup (X_2 \cap \alpha_2)$
VI) $(tr \restriction \alpha_1, \sigma_1', X_1) \in [\![scp_1]\!]^{\mathcal{SF}}\big(\{(\langle\rangle, \sigma_1, \_)\}\big)$
VII) $(tr \restriction \alpha_2, \sigma_2', X_2) \in [\![scp_2]\!]^{\mathcal{SF}}\big(\{(\langle\rangle, \sigma_2, \_)\}\big)$
VIII) $set(tr) \subseteq (\alpha_1 \cup \alpha_2)$

<u>Want to show (goal):</u>
$(tr, X) \in \mathcal{SF}_{\mathcal{U}(scp)}(\sigma)$

Proof: $\exists\, \sigma'.\ (tr, \sigma', X) \in [\![scp]\!]^{\mathcal{SF}}\big(\{(\langle\rangle, \sigma, Y) \mid Y \subseteq \Sigma\}\big) \Longrightarrow (tr, X) \in \mathcal{SF}_{\mathcal{U}(scp)}(\sigma)$

<u>Proof:</u>
Using Definition 5.16, we can reformulate the goal:

$$\exists\, \sigma'.\ \sigma \overset{tr}{\Longrightarrow}_{scp} \sigma' \wedge \sigma'{\downarrow}_{scp} \wedge \sigma' \text{ ref}_{scp}\ X$$

We show the three parts separately. Before that, we use VI) and III) as well as VII) and IV) to obtain $(tr_1, X_1) \in \mathcal{SF}_{\mathcal{U}(scp_1)}(\sigma_1)$ and $(tr_2, X_2) \in \mathcal{SF}_{\mathcal{U}(scp_2)}(\sigma_2)$, respectively. Using Definition 5.16, we get the following for the components

IX) $\exists\, \sigma_1'.\ \sigma_1 \overset{tr_1}{\Longrightarrow}_{scp_1} \sigma_1' \wedge \sigma_1'{\downarrow}_{scp_1} \wedge \sigma_1' \text{ ref}_{scp_1}\ X$
X) $\exists\, \sigma_2'.\ \sigma_2 \overset{tr_2}{\Longrightarrow}_{scp_2} \sigma_2' \wedge \sigma_2'{\downarrow}_{scp_2} \wedge \sigma_2' \text{ ref}_{scp_2}\ X$

<u>$\sigma \overset{tr}{\Longrightarrow}_{scp} \sigma'$:</u>
The proof is similar to the sufficient condition of the proof of correspondence of the traces semantics in the previous subsection. It uses all assumptions but V).
<u>$\sigma'{\downarrow}_{scp}$:</u>
From IX) and X) we have $\sigma_1'{\downarrow}_{scp_1}$ and $\sigma_2'{\downarrow}_{scp_2}$.
When the components are stable, the combination is stable, too.

<u>$\sigma' \text{ ref}_{scp}\ X$:</u>
From IX) and X) we have $\sigma_1' \text{ ref}_{scp_1}\ X$ and $\sigma_2' \text{ ref}_{scp_2}\ X$.
Using Lemma A.1 and V), we obtain $\sigma' \text{ ref}_{scp}\ X$

Thus, all stable failures that occur in the denotational stable failures semantics of the concurrent combination of two components occur also in the operationally characterized stable failures semantics of the combined components. □

Together, the necessary and the sufficient implications show the correspondence between the operationally characterized stable failures semantics and the denotational stable failures semantics.

A.2 Proofs of T1 to SF4 for CUC

In this section, we prove that the adapted version of the essential properties for denotational semantics of CSP also hold for the denotational semantics of CUC. Before we prove the properties to hold, we prove a lemma that the input sets to $[\![\cdot]\!]^{\mathcal{T}}$ are also contained in its output sets. One could say that $[\![\cdot]\!]^{\mathcal{T}}$ is "non-destructive".

Lemma A.2: Input Preservation

$$S \subseteq [\![sp]\!]^{\mathcal{T}}(S) \tag{I}$$

$$\big(\forall\, s \in S.\ s = (\langle\rangle, \_)\big) \Longrightarrow S \subseteq [\![scp]\!]^{\mathcal{T}}(S) \tag{II}$$

Proof: Input Preservation

(I) holds, as all non-parallel rules contain "$S \cup$".
(II) holds by induction on the concurrent structure of the states. The base-step $(\langle\rangle, \sigma_i) \in [\![sp_i]\!]^{\mathcal{T}}\big(\{(\langle\rangle, \sigma_i)\}\big)$ holds due to (I). □

We now prove the six properties T1 to SF4. For each property, we first restate the property and then give the proof.

Property T1

$$\langle\rangle \in \mathcal{T}_{cuc}(\sigma)$$

The empty trace must always be contained. Any program can be observed to do nothing.

Proof: T1 holds for CUC

By Assumption 5.5 (Empty Initial Traces), all initial traces are empty, i. e., $\big(\forall\, s \in S.\ s = (\langle\rangle, \_)\big)$.
We use structural induction over SCP, i. e., we go through the rules of $\mathcal{T}_{cuc}$ and use Lemma A.2 to show that those initial traces are preserved. □

Property T2

$$\forall\, tr\ tr'.\ tr' \leq tr \wedge tr \in \mathcal{T}_{cuc}(\sigma) \Longrightarrow tr' \in \mathcal{T}_{cuc}(\sigma)$$

$\mathcal{T}$ is prefix closed. All partial behaviors can be observed.

Proof: T2 holds for CUC

Induction over the length of traces.
The only rule changing the trace is $\mathcal{T}$-COMM. The new traces added are extending existing traces by one event.
By Lemma A.2 (I), the existing traces are also in T.
By Assumption 5.5, we always start with empty traces.
Thus, by induction hypothesis, all prefixes of the new traces are in $\mathcal{T}$. □

Property SF1

$$(tr, X) \in \mathcal{SF}_{cuc}(\sigma) \Longrightarrow tr \in \mathcal{T}_{cuc}(\sigma)$$

All trace-state pairs are included in the traces semantics. This property ensures that the stable failures semantics "acts" within the boundaries of the traces semantics.

Proof: SF1 holds for CUC

It holds as we extended the traces semantics in a safe way: The semantics are very similar, we only discard failures (and with that trace-state pairs) that are not stable. □

Property SF2

$$(tr, X) \in \mathcal{SF}_{cuc}(\sigma) \wedge X' \subseteq X \Longrightarrow (tr, X') \in \mathcal{SF}_{cuc}(\sigma)$$

Refusal sets are subset closed.

Proof: SF2 holds for CUC

This holds by Assumption 5.6 (Initial Failures) and by construction, as we always add *all subsets* of a maximal refusal set. □

Property SF3

$$(tr, X) \in \mathcal{SF}_{cuc}(\sigma) \wedge \forall\, a \in X'.\ tr^\frown\langle a\rangle \notin \mathcal{T}_{cuc}(\sigma) \Longrightarrow (tr, X \cup X') \in \mathcal{SF}_{cuc}(\sigma)$$

All events that can be refused occur in a refusal set. More specifically, the refusal sets can be augmented with refused events. This is the crucial property ensuring that the stable failures are "enough" refusals to show liveness properties, because it ensures that all events that can be refused are contained in the stable failures semantics.

Proof: SF3 holds for CUC

As for SF2, this holds by Assumption 5.6 (Initial Failures) and by construction, as we always add all subsets of a *maximal* refusal set. □

Property SF4

$$\sigma \overset{tr}{\Rightarrow}_{cuc} \sigma' \wedge \sigma'_{pc} \notin_{pc} code \Longrightarrow \exists\, X.\, (tr, X) \in \mathcal{SF}_{cuc}(\sigma)$$

Terminal failures are stable.
Although the equivalence $\exists\, \sigma'.\, \sigma \overset{tr}{\Rightarrow}_{cuc} \sigma' \Leftrightarrow tr \in \mathcal{T}_{cuc}(\sigma)$ holds, we choose not to use $\mathcal{T}$ as in the CSP version of SF4, as we need to talk about the reached state σ' explicitly to model terminal behavior.

Proof: SF4 holds for CUC

This also holds by construction: After the execution of each instruction, a failure is inserted, which is terminal at that moment. The maximal refusal set for those inserted failures is Σ, i. e., all possible (visible) events can be refused. In the sequential composition rule ($\mathcal{SF}$-SEQ), all former terminal failures are removed, i. e., all failures whose state has a program counter that points into the code. The failures whose state's program counter does not point into the code remain. Thus, traces that reach states whose program counter does not point into the code are captured by the stable failures semantics and can refuse anything. □

A.3 Mapping to the Isabelle/HOL Formalization

In this section, we give an overview of how the results from Chapter 5 are formalized in Isabelle/HOL. The Isabelle/HOL formalization accompanying this thesis is published separately in a technical report [BBD+19]. The content of this section corresponds to the content of the technical report. The advantage of reading the technical report in its source form (which is an Isabelle theory file) in a suitable Isabelle IDE is that all referenced entities (definitions, lemmas, theory files, etc.) are links to the respective sections in the Isabelle/HOL formalization. This enables easy navigation within the formalization.

The intention of this section is to give a mapping from Chapter 5 to the Isabelle/HOL formalization. For a tutorial on Isabelle/HOL see [NK14]. Due to technical implementation reasons and because the Isabelle/HOL formalization is a bit older than the formalization in this thesis, which has evolved since, there is not a one-to-one mapping from the formalization in this thesis to the formalization in Isabelle/HOL. Most notably, the operational semantics of CUC is not concurrent. The reasoning (why it is sufficient to consider single components) is explained in Section 5.6. The concurrent cases of the correspondence proofs between the operational and denotational semantics that are missing in the Isabelle/HOL formalization are given in Appendix A.1.

Section 5.2 (Syntax and Semantic States)

Definition 5.1 (Basic Data Types): The local states (LStates) are defined by the record `'data_store definitions.state`. It is parametrized by the type `'data_store` of the data store (formerly register store) R. It contains the data store and a program counter of type `definitions.label`.
Concurrent states are defined in `'state denotational_parallel.parallelState`.
Events can be of any type `'e definitions.event`. The set of all events can be fixed within a locale, for example in the locale `stable_failures.SFsemSigma`.

Definition 5.2 (Instructions of CUC): The instructions of CUC are formalized with the datatype `('state, 'event) instr` and are parametrized over the type of the state and the type of events.

Assumption 5.1 (At Least One Successor State): We assume this assumption implicitly by only using functions f that map to non-empty sets for `do` f.

Definition 5.3 (Local Program lp): The local programs are of the type `('state, 'event) labeled_instruction_set`.

Assumption 5.2 (Uniqueness of Labels): The uniqueness of labels is required in the respective lemmas. The notion of uniqueness of labels for unstructured programs is captured as an assumption of the locale `liSet`. The notion of uniqueness of labels for structured programs is captured by the definition `sc_wellformed`.

Definition 5.4 (Labels of a Program *labels*): To extract the labels of an unstructured program, we use the definition `indom`.

Definition 5.5 (Concurrent Program *cp*): We do not define unstructured concurrent programs in the Isabelle/HOL formalization. This is due to the fact that our main goal for the Isabelle/HOL formalization was to relate *structured* concurrent programs to CSP processes. The type of structured concurrent programs is defined by `('state, 'event) structured_code_parallel`.

Assumption 5.3 (Same Tree Structure): This is defined in the directly in the semantics, e.g., see `denotational_parallel.densem`$_p$: We define successor states only for states whose tree structure matches the tree structure of the code.

Definition 5.6 (Structured Program *sp*): The type used for structured code is `('state, 'event) structured_code` which is actually just a wrapper around the definition `'code structured_code_base` which builds a tree with single lines of labeled instruction as leaves.

Definition 5.7 (Unstructuring Function, $\mathcal{U}$): To obtain the unstructured code from structured code we use the function `sc_inj`.

Definition 5.8 (Labels of a Structured Program *labels*): The labels of a structured program are extracted directly in the formalization via `dom`.

Assumption 5.4 (Uniqueness of Labels for *sp*): As for the uniqueness of labels of unstructured code, we require this property in the locales. For the uniqueness of labels of structured code, we use the function `sc_wellformed`.

Definition 5.9 (Structured Concurrent Program *scp*): The type of structured concurrent programs is defined by `('state, 'event) structured_code_parallel` .

Section 5.3 (Semantics)

Subsection 5.3.1 (Operational Semantics)

Definition 5.10 (Operational Semantics of CUC): The operational semantics of CUC are distributed over several definitions. The smallstep semantics of CUC are defined by `smallstep`, which is an inductively defined set of `('state, 'event) trace_state` pairs, describing trace-state pairs before and after the execution of an instruction, i.e., the transitions. `smallstep` is parametrized by a labeled instruction set. To require the wellformedness assumptions on unstructured programs (uniqueness of labels), we defined a wrapper named Smallstep (observe the capital 'S') `small_step.liSet.Smallstep` inside a locale.

The concurrent operational semantics *are not* formalized. See Section 5.6 for the explanation. The concurrent denotational semantics *are* formalized.

The reflexive transitive hull of small step is defined in `small_step.liSet.multistep`. For historical reasons (see the next definition) `multistep` is defined by prepending to the execution traces. Which is in contrast to Definition 5.10, where it is defined by appending.

Definition 5.11 (Terminating Execution): Terminating executions are defined by `small_step.liSet.Exec`, which is similar to `multistep`. The difference is that the "final empty step" is only allowed in `exec`, if the program counter points outside of the code, i.e., the execution has terminated.

Subsection 5.3.2 (Defining Denotational Semantics with Fixpoints)

We use the existing Isabelle/HOL library `Complete_Partial_Order.ccpo_class`, which provides most notably a fixpoint induction scheme.

Subsection 5.3.3 (Traces Semantics)

Definition 5.12 (Traces Semantics of CUC): The sequential part of the traces semantics is defined in `denotational.densem`. The concurrent part of the traces semantics is defined in `denotational_parallel.densem`$_p$.

Definition 5.13 (Operational Characterization of the Traces of CUC): This is not defined explicitly in the Isabelle/HOL formalization. The operational and the traces semantics are related directly in a lemma. See the correspondence theorem for traces below (Theorem 5.1).

Assumption 5.5 (Empty Initial Traces): The assumption is only implicit in that instantiations only use an empty trace initially. See for example the preconditon in the example `concurrent_buffer_example.SingleBuffer.TPre`.

Theorem 5.1 (Correspondence Between Operational Characterization and Traces Semantics): The correspondence between the operational semantics and the traces semantics is captured by theorems, one for each direction:

- `denotational.liSet.densem_implies_multistep` and
- `denotational.liSet.multistep_implies_densem`

Corollary 5.1 (Invariance Under Structure): In the formalization, the independence of the denotational semantics from a particular structure on the unstructured code is proven as a step *towards* the proof of the previous theorem (in contrast to being a corollary of it).

Definition 5.3.4 (Stable Failures Semantics)

Definition 5.14 (CSP-like Stable States of CUC): The definition of stable states is implicit in the definition of stable failures `stable_failures.SFsemSigma.sfsem`. Only stable states are included in the semantics. That the stable failure semantics works as expected is shown in the theory file `stable_failures_traces_conformant` by proving SF1 to SF4.

Definition 5.15 (Refusal Set of CUC): As for the stable states, the refusal sets are defined implicitly in the definition of stable failures.

Definition 5.16 (Operational Characterization of the Stable Failures of CUC): This is not defined explicitly in the formalization. The operational and the stable failures semantics are related directly in a lemma. See the correspondence theorem for stable failures below (Theorem 5.2).

Definition 5.17 (Communication States): The communication and normal states are defined in `'state NCstate`. In the formalization we use a sum datatype with explicit constructors. Thus, we do not need to append "$\times\{c\}$". The tree of concurrent *NCStates* (named *CNCStates*) is defined in `'state parallelState`.

Definition 5.18 (Test for Normal State and Conversions to Communication State, $N(\cdot)$, $\cdot^C$): The test for normal state is defined by `N`. The conversion from normal state to communication is implemented by `NCC`.

Assumption 5.6 (Initial Failures): We usually generate initial failures from traces using the following function which generates refusal sets according to the assumption: `stable_failures_traces_conformant.SFsemSigma.trace_state_set_to_failure_set`

Definition 5.19 (Removal of Former Terminal Failures, $\setminus_{(\cdot)}$): The former terminal failures are removed by `SClean`.

Definition 5.20 (Stable Failures Semantics of CUC): The sequential part of the traces semantics is defined in `stable_failures.SFsemSigma.sfsem` The concurrent part of the traces semantics is defined in `stable_failures_parallel.sfsem`$_p$.

Theorem 5.2 (Correspondence Between Operational Characterization and Stable Failures Semantics): The correspondence between the operational semantics and the stable failures semantics is captured by theorems, one for each direction.

- `stable_failures.SFsemSigma.sfsem_implies_multistep`
- Observe that due to the implicit definition of stable states and refusal sets, the direction from `multistep` to failures is split in two theorems: One for terminating executions and one for non-terminating executions.
  - `stable_failures.SFsemSigma.Exec_implies_sfsem`
  - `stable_failures.SFsemSigma.multistep_trace_implies_comm_failure`

Subsection 5.3.5 (Compatibility to CSP)

We did not show T1 and T2 explicitly.
T1: If we assume that the initial traces are empty, we can use `denotational.densem_super_id_element` to follow that the traces semantics also contains a trace-state pair with the empty trace.

T2: We have shown that `multistep` is prefix closed in `small_step.liSet.multistep_prefix_closed`.
Using the correspondence between `multistep` and the traces semantics, we conclude that the traces semantics is also prefix closed.

The properties SF1 to SF4 are shown in the following theory files. First for the sequential case, then for the concurrent case.

- `stable_failures_traces_conformance_parallel.thy`
- `stable_failures_traces_conformant.thy`

Definition 5.4 (Hoare Calculus)

Definition 5.21 (Hoare Triple for CUC):
The Hoare triple is defined in `hoare.SFsemSigma.hoare_valid`.

Definition 5.22 (Hoare Calculus for CUC):
The Hoare calculus itself is defined in `hoare.SFsemSigma.hoare`.

Theorem 5.3 (Soundness of Our Hoare Calculus): We have proven the soundness of our Hoare Calculus in `hoare.SFsemSigma.hsem_soundness`

Section 5.5 (Relating CSP and CUC)

Lemma 5.1 (Traces Imply Stable Failures in CUC): When we only consider divergence free CUC programs, then all traces appear also in the stable failures semantics:
`stable_failures_traces_conformant.SFsemSigma.tr_of_T_will_be_in_SF`
The lemma talks about successors of starting states, as we allow arbitrary traces for initial trace-state pairs.

Theorem 5.4 (Traces Refinement Implies Stable Failures Refinement for CUC):
Again, this theorem only holds for divergence free programs.

```
stable_failures_traces_conformant.SFsemSigma.
  sfsem_no_divergencies_imp_sf_refinement_implies_traces_refinement
```

Assumption 5.7 (Divergence Freedom of CUC Programs): We assume divergence freedom directly in lemmas and theorems (see the theorems above). For our (quite simple) examples we used static code analysis, to ensure that every instruction will either be followed by a `comm` instruction (and thus produce a visible event) or will lead to termination. The definitions and lemmas can be found in the theory file `static_code_analysis`.

Subsection 5.5.1 (Example)

Reminder: For the formalization of CSP, we use the CSP Prover library [IR05] by Yoshinao Isobe and Markus Roggenbach.

In the theory file `concurrent_buffer_example` we have formalized the example from Subsection 5.5.1. The major difference in the structure is that we first show that the

code fulfills the sufficient property, and afterwards that the sufficient property implies the specification. In contrast to the example in Subsection 5.5.1, in the Isabelle/HOL formalization we need many more fine grained definitions and lemmas.

The CUC program is defined in `concurrent_buffer_example.SingleBuffer.LIS`. The CSP specification is defined in the assumptions of the lemmas that use it, e.g. `concurrent_buffer_example.SingleBuffer.Conn_Imp_Spec`
It is defined in fixpoint notation

```
"((([[$spec]]Ff [[PN]]Ffix)::(nat × nat) domF) =
        [[(% x .  (TIn, x)) ?  k -> (TOut, k) -> $spec]]Ff [[PN]]Ffix"
```

First, we give the name of the process (`$spec`), and then apply a fixpoint to all process names (`PN`; to handle possible mutual recursion). After the "=" sign, we see the almost CSP like syntax. In CSP syntax, the process would look like the following: $Spec = in?k \rightarrow out!k \rightarrow Spec$

The sufficient property is defined via its two disjuncts, $\mathbb{F}_{empty}$ and $\mathbb{F}_{full}$, which are named `F-even` and `F-odd` in the Isabelle/HOL formalization. The sufficient property is called "connecting property" or "conn".

That the sufficient property implies the CSP specification is proven in the lemma: `concurrent_buffer_example.SingleBuffer.Conn_Imp_Spec`

That the sufficient property holds for all stable failures of the CUC program is proven in the lemma: `concurrent_buffer_example.SingleBuffer.TSeq123_Imp_Conn`

Finally, the conclusion that the CUC program refines the CSP process is shown in: `concurrent_buffer_example.SingleBuffer.Spec_SF_Refinement_Impl`

We have proven the properties about the single buffer inside a locale: `concurrent_buffer_example.SingleBuffer`.
We can now instantiate this locale for each homogeneous component to reason about their parallel composition. The refinement of the concurrent versions is shown in the lemma `concurrent_buffer_example.Buffer_SF_Parallel_refinement`.

A.4 Protocol Constraints

Definition A.1 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.1. 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. $a \veebar b := a \vee b \wedge \neg(a \wedge b)$

Definition A.1: Protocol Constraints (Full)

$$\mathcal{P}_{cuc,sv,\psi}\big(\sigma,\mathcal{X},(\Gamma,\hat{\sigma})\big) := \big(\forall\, c.\, \mathcal{X}(c) = \text{FREE} \Longrightarrow \neg\Gamma(sr_c) \wedge \neg\Gamma(fr_c)\big) \\ \wedge\ \forall\, id. \mathcal{P}^{id}_{cuc,sv,\psi}\big(\sigma,\mathcal{X},(\Gamma,\hat{\sigma})\big)$$

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

Out of code:

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

`do` f:

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

`cbr`:

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

send:

$$\vee\, (\sigma^{id}_{pc}, \texttt{comm}_s\, id\, c\, x_s) \in cuc^{id} \wedge (\hat{\sigma}^{id}_{pc}, \texttt{cas}\, hl_c\, m_c\, \text{FREE}\, id) \in sv^{id} \wedge \psi(\sigma^{id}_{pc}) = \hat{\sigma}^{id}_{pc} \wedge id \notin \mathcal{X} \tag{S}$$

$$\vee\, (\sigma^{id}_{pc}, \texttt{comm}_s\ id\ c\ x_s) \in cuc^{id} \wedge \psi(\sigma^{id}_{pc}) + 1 = \hat{\sigma}^{id}_{pc} \wedge \\ \big(\hat{\sigma}^{id}_{pc}, \texttt{cbr}\ hl_c\ \big(\psi(\sigma^{id}_{pc}) + 2\big)\ \psi(\sigma^{id}_{pc})\big) \in sv^{id} \wedge \big(\neg\hat{\sigma}^{id}_{ds}(hl_c) \wedge id \notin \mathcal{X} \vee \\ \hat{\sigma}^{id}_{ds}(hl_c) \wedge \mathcal{X}(c) = id_{in} \wedge \Gamma(m_c) = id \wedge \neg\Gamma(sr_c) \wedge \neg\Gamma(fr_c)\big) \tag{S1}$$

$$\vee\, (\sigma^{id}_{pc}, \texttt{comm}_s\ id\ c\ x_s) \in cuc^{id} \wedge (\hat{\sigma}^{id}_{pc}, \texttt{write}\ \gamma_c\ x_s) \in sv^{id} \wedge \psi(\sigma^{id}_{pc}) + 2 = \hat{\sigma}^{id}_{pc} \\ \wedge\, \mathcal{X}(c) = id_{in} \wedge \Gamma(m_c) = id \wedge \neg\Gamma(sr_c) \wedge \neg\Gamma(fr_c) \tag{S2}$$

$$\vee\, (\sigma^{id}_{pc}, \texttt{comm}_s\ id\ c\ x_s) \in cuc^{id} \wedge (\hat{\sigma}^{id}_{pc}, \texttt{write}\ sr_c\ \top) \in sv^{id} \wedge \psi(\sigma^{id}_{pc}) + 3 = \hat{\sigma}^{id}_{pc} \\ \wedge\, \mathcal{X}(c) = id_{in} \wedge \Gamma(m_c) = id \wedge \Gamma(\gamma_c) = \hat{\sigma}^{id}_{ds}(x_s)\ \wedge \neg\Gamma(sr_c) \wedge \neg\Gamma(fr_c) \tag{S3}$$

$$\vee\, (\sigma^{id}_{pc}, \texttt{comm}_s\, id\, c\, x_s) \in cuc^{id} \wedge (\hat{\sigma}^{id}_{pc}, \texttt{cas}\, ss_c\, fr_c\, \top\, \bot) \in sv^{id} \wedge \psi(\sigma^{id}_{pc}) + 4 = \hat{\sigma}^{id}_{pc} \\ \wedge\, \Gamma(m_c) = id \wedge \Gamma(\gamma_c) = \hat{\sigma}^{id}_{ds}(x_s) \wedge \neg\Gamma(fr_c) \wedge \\ \big(\Gamma(sr_c) \wedge \mathcal{X}(c) = id_{in} \vee \neg\Gamma(sr_c) \wedge (\exists\, id_r.\, \mathcal{X}(c) = (id, id_r)_{in})\big) \tag{S4}$$

$$\vee\, (\sigma^{id}_{pc}, \texttt{comm}_s\, id\, c\, x_s) \in cuc^{id} \wedge \Big(\hat{\sigma}^{id}_{pc}, \texttt{cbr}\ ss_c\ \big(\psi(\sigma^{id}_{pc}) + 6\big)\ \big(\psi(\sigma^{id}_{pc}) + 4\big)\Big) \in sv^{id} \wedge \\ \psi(\sigma^{id}_{pc}) + 5 = \hat{\sigma}^{id}_{pc} \wedge \Gamma(m_c) = id \wedge \neg\Gamma(fr_c) \wedge \Gamma(\gamma_c) = \hat{\sigma}^{id}_{ds}(x_s) \wedge \neg\hat{\sigma}^{id}_{ds}(ss_c) \wedge \\ \big(\Gamma(sr_c) \wedge \mathcal{X}(c) = id_{in} \vee \neg\Gamma(sr_c) \wedge (\exists\, id_r.\, \mathcal{X}(c) = (id, id_r)_{in})\big) \tag{S5}$$

$$\vee\, (\sigma^{id}_{pc} - 1, \texttt{comm}_s\, id\, c\, x_s) \in cuc^{id} \wedge (\hat{\sigma}^{id}_{pc}, \texttt{cas}\, ss_c\, fr_c\, \top\, \bot) \in sv^{id} \wedge \\ \psi(\sigma^{id}_{pc} - 1) + 4 = \hat{\sigma}^{id}_{pc} \wedge \Gamma(m_c) = id \wedge \neg\Gamma(sr_c) \wedge \\ \big(\Gamma(fr_c) \wedge \mathcal{X}(c) = id_{un} \vee \neg\Gamma(fr_c) \wedge (\exists\, id_r. \mathcal{X}(c) = (id, id_r)_{un})\big) \tag{S4'}$$

$$\vee\, (\sigma^{id}_{pc} - 1, \texttt{comm}_s\, id\, c\, x_s) \in cuc^{id} \wedge \psi(\sigma^{id}_{pc} - 1) + 5 = \hat{\sigma}^{id}_{pc} \wedge \Gamma(m_c) = id \wedge \neg\Gamma(sr_c) \\ \Big(\hat{\sigma}^{id}_{pc}, \texttt{cbr}\ ss_c\big(\psi(\sigma^{id}_{pc} - 1) + 6\big)\big(\psi(\sigma^{id}_{pc} - 1) + 4\big)\Big) \in sv^{id} \wedge \big((\Gamma(fr_c) \veebar \hat{\sigma}^{id}_{ds}(ss_c)) \\ \wedge\, \mathcal{X}(c) = id_{un} \vee \neg\Gamma(fr_c) \wedge \neg\hat{\sigma}^{id}_{ds}(ss_c) \wedge (\exists\, id_r. \mathcal{X}(c) = (id, id_r)_{un})\big) \tag{S5'}$$

Definition A.1: Protocol Constraints (Full)

$$\vee\, (\sigma^{id}_{pc} - 1, \texttt{comm}_s\, id\, c\, x_s) \in cuc^{id} \wedge (\hat{\sigma}^{id}_{pc}, \texttt{write}\, m_c\, \text{FREE}) \in sv^{id} \wedge \psi(\sigma^{id}_{pc} - 1) + 6 = \hat{\sigma}^{id}_{pc}$$
$$\wedge\, \mathcal{X}(c) = id_{un} \wedge \Gamma(m_c) = id \wedge \neg\Gamma(sr_c) \wedge \neg\Gamma(fr_c) \tag{S6}$$

receive:

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

$$\vee\, (\sigma^{id}_{pc}, \texttt{comm}_r\, id\, c\, x_r) \in cuc^{id} \wedge (\hat{\sigma}^{id}_{pc}, \texttt{cbr}\, ss_c\, sr_c\, \top\, \bot) \in sv^{id} \wedge \psi(\sigma^{id}_{pc}) + 1 = \hat{\sigma}^{id}_{pc} \wedge$$
$$\big(\hat{\sigma}^{id}_{ds}(ss_c) \wedge \big(\exists\, id_s.\, \mathcal{X}(c) = (id_s, id)_{in}\big) \vee \neg\hat{\sigma}^{id}_{ds}(ss_c) \wedge id \notin \mathcal{X}\big) \tag{R1}$$

$$\vee\, (\sigma^{id}_{pc}, \texttt{comm}_r\, id\, c\, x_r) \in cuc^{id} \wedge (\hat{\sigma}^{id}_{pc}, \texttt{read}\, x_r\, \gamma_c) \in sv^{id} \wedge \psi(\sigma^{id}_{pc}) + 2 = \hat{\sigma}^{id}_{pc} \wedge$$
$$\big(\exists\, id_s.\, \mathcal{X}(c) = (id_s, id)_{in}\big) \tag{R2}$$

$$\vee\, (\sigma^{id}_{pc} - 1, \texttt{comm}_r\, id\, c\, x_r) \in cuc^{id} \wedge (\hat{\sigma}^{id}_{pc}, \texttt{write}\, fr_c\, \top) \in sv^{id} \wedge \psi(\sigma^{id}_{pc} - 1) + 3 = \hat{\sigma}^{id}_{pc} \wedge$$
$$\big(\exists\, id_s.\, \mathcal{X}(c) = (id_s, id)_{un}\big) \tag{R3}$$

A.5 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.

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

In Figure A.3, we depict the labeled transitions of the protocol. In contrast to Figure 6.2, 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.1). 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).

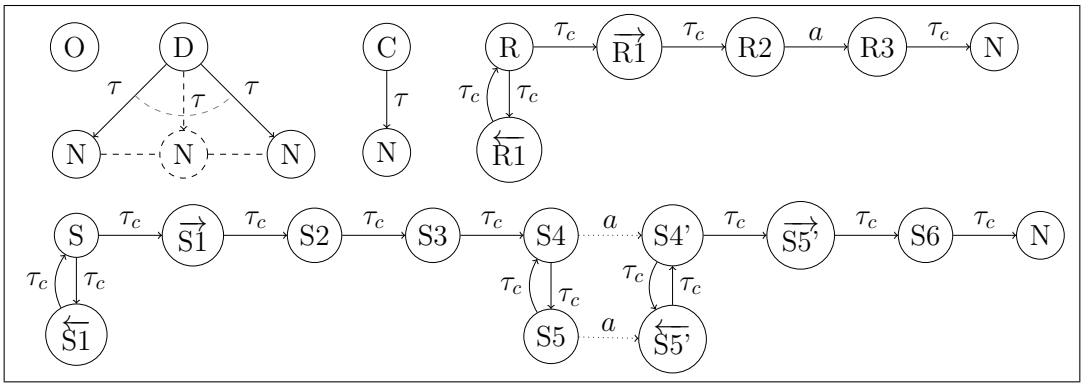


Figure A.3: *sv* Transitions Between the Disjuncts of $\mathcal{P}^{id}_{cuc,sv,\psi}$

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:

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

Up-simulation:

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

Unlocking-simulation:

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

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

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

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

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

Proof: Theorem 6.3 (Fitting Implies Handshake Refinement)

in $\mathcal{B}$.

---

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.1. 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

Proof: Theorem 6.3 (Fitting Implies Handshake Refinement)

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:
We consider the receiver, thus, let $id_r := id$.
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 $\mathcal{X}(c) = \text{FREE}$ from the down-simulation, we conclude that (S) holds for id_s as well as (R) for id_r.
Furthermore, from $\mathcal{X}(c) = \text{FREE}$ 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.6 Refusals imply Refusals

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

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

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

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

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

A.7 Proofs from the Evaluation

A.7.1 Proof of Correctness of the Sufficient Property of the Server

In this subsection, we prove the correctness of the sufficient property of the server component from Chapter 7 (Evaluation & Case Study). We first recall the relevant definitions of the server component i ($Server_i$) and its sufficient property ($S_i^{\subseteq}$). Then, we restate Lemma 7.1 and give the proof for the case of the server component.

$$Server_i = \underset{id^{in} \in ID \setminus \{s_i\}}{\Box} a_i.id^{in}.s_i?x \rightarrow \underset{id^{out} \in ID \setminus \{s_i\}}{\Box} b_i.s_i.id^{out}.f(x) \rightarrow Server_i$$

$$S_i^{\subseteq}(F) := F \in \mathbb{F}_i^{idle} \vee F \in \mathbb{F}_i^{busy} \qquad \text{where}$$

$$\mathbb{F}_i^{idle} := \Big\{ \big(\langle a_i.id_*^{in}.s_i.x_*, b_i.s_i.id_*^{out}.f(x_*)\rangle^*, X\big) \;\Big|\; X \subseteq \Sigma \setminus \{a_i.id^{in}.s_i.v \mid id^{in} \in ID \setminus \{s_i\} \wedge v \in \mathbb{T}\} \Big\}$$

$$\mathbb{F}_i^{busy} := \Big\{ \big(\langle a_i.id_*^{in}.s_i.x_*, b_i.s_i.id_*^{out}.f(x_*)\rangle^* \frown \langle a_i.id^{in}.s_i.v\rangle, X\big) \;\Big|\; X \subseteq \Sigma \setminus \big\{b_i.s_i.id^{out}.f(v) \,\big|\, id^{out} \in ID \setminus \{s_i\}\big\} \wedge id^{in} \in ID \setminus \{s_i\} \wedge v \in \mathbb{T} \Big\}$$

Lemma 7.1: Both Sufficient Properties are Correct

$$C_c^{\subseteq}(F) \Longrightarrow F \in \mathcal{SF}(Client_c)$$
$$S_i^{\subseteq}(F) \Longrightarrow F \in \mathcal{SF}(Server_i)$$

Proof: $S_i^{\subseteq}(tr, X) \Longrightarrow (tr, X) \in \mathcal{SF}(Server_i)$

Consider (tr, X) with $S_i^{\subseteq}(tr, X)$.
Want to show (goal):
$(tr, X) \in \mathcal{SF}(Server_i)$

We show this by case distinction whether the failure (tr, X) is in $\mathbb{F}_i^{idle}$ or $\mathbb{F}_i^{busy}$.

$(tr, X) \in \mathbb{F}_i^{idle}$:
We have $tr = \langle a_i.id_*^{in}.s_i.x_*, b_i.s_i.id_*^{out}.f(x_*)\rangle^*$ and
$X \subseteq \Sigma \setminus \{a_i.id^{in}.s_i.v \mid id^{in} \in ID \setminus \{s_i\} \wedge v \in \mathbb{T}\}$.
We show $(tr, X) \in \mathcal{SF}(Server_i)$.
After any number of times of the trace $\langle a_i.id_*^{in}.s_i.x_*, b_i.s_i.id_*^{out}.f(x_*)\rangle$, the process $Server_i$ behaves again as $Server_i$.
The process $Server_i$ offers the events $\{a_i.id^{in}.s_i.v \mid id^{in} \in ID \setminus \{s_i\} \wedge v \in \mathbb{T}\}$. Thus, it can

Proof: $S_i^{\subseteq}(tr, X) \Longrightarrow (tr, X) \in \mathcal{SF}(Server_i)$

refuse any event not in $\{a_i.id^{in}.s_i.v \mid id^{in} \in ID \setminus \{w_i\} \wedge v \in \mathbb{T}\}$.
Thus, all failures described by $\mathbb{F}_i^{idle}$ are stable failures of the process$Server_i$.
$\Longrightarrow (tr, X) \in \mathcal{SF}(Server_i)$

$\underline{(tr, X) \in \mathbb{F}_i^{busy}}$:
We have $tr = \langle a_i.id_*^{in}.s_i.x_*, b_i.s_i.id_*^{out}.f(x_*)\rangle^{*}\frown\langle a_i.id^{in}.s_i.v\rangle$ and
$X \subseteq \Sigma \setminus \left\{b_i.s_i.id^{out}.f(v) \,\middle|\, id^{out} \in ID \setminus \{s_i\}\right\}$.
We show $(tr, X) \in \mathcal{SF}(Server_i)$.
As in the first case, after any number of times of the trace $\langle a_i.id_*^{in}.s_i.x_*, b_i.s_i.id_*^{out}.f(x_*)\rangle^*$, the process $Server_i$ behaves again as $Server_i$. After the event $a_i.id^{in}.s_i.v$, the process $Server_i$ behaves as $\Box_{id^{out} \in ID \setminus \{s_i\}}\, b_i.s_i.id^{out}.f(v) \rightarrow Server_i$.
The process $\Box_{id^{out} \in ID \setminus \{s_i\}}\, b_i.s_i.id^{out}.f(v) \rightarrow Server_i$ offers the events
$\left\{b_i.s_i.id^{out}.f(v) \,\middle|\, id^{out} \in ID \setminus \{s_i\}\right\}$. Thus, it can refuse any event not in
$\left\{b_i.s_i.id^{out}.f(v) \,\middle|\, id^{out} \in ID \setminus \{s_i\}\right\}$.
Thus, all failures described by $\mathbb{F}_i^{busy}$ are stable failures of the process$Server_i$.
$\Longrightarrow (tr, X) \in \mathcal{SF}(Server_i)$

In both cases, the failures described by $S_i^{\subseteq}$ are stable failures of the process $Server_i$. $\Box$

A.7.2 Proof that the CUC Program for the Server satisfies its Sufficient Property

In this section, we show that the CUC program $Server_i$ satisfies the sufficient property $S_i^{\subseteq}$, i. e., we give the proof for Lemma 7.2.

Lemma 7.2: The CUC Server Fulfills its Sufficient Property

$$\{\sigma_{pc} = 1 \wedge N(\sigma) \wedge tr = \langle\rangle\}\ s_i^{\oplus}\ \Big\{\lambda tr\ \sigma\ X.\ S_i^{\subseteq}(tr, X)\Big\}$$

To this end, we use our Hoare calculus. The sufficient property $S_i^{\subseteq}$ describes only the desired communication behavior. To use it as an invariant, we extend it with information about the state. The state information is only required intermittently to know which instruction is to be executed next (program counter) and to relate processed data (data store).

The general idea of the proof is to formulate pre- and postconditions for each instruction, and then combine the proofs for every component to a proof for an initial precondition and an overall postcondition, which includes an invariant on the communication capabilities, i. e., the sufficient property. We first recall the CUC program for the server and the sufficient property $S_i^{\subseteq}$. Then, we present the overall structure and proof tree, and afterwards consider the different parts of the proof tree in detail.

$$
\begin{array}{l}
Server_i := \\
1:\ \mathrm{comm}_r\ \ a_i\ \ x \\
2:\ \textbf{do}\ \ (\lambda ds.\ \{ds[y := f(ds(x))]\}) \\
3:\ \mathrm{comm}_s\ \ b_i\ \ y \\
4:\ \textbf{cbr}\ \ (\lambda ds.\ \textsc{True})\ \ 1\ \ 1
\end{array}
$$

We use the circled line numbers ①, ②, ③, and ④ as shorthands for the instructions. The structured version $s_i^{\oplus}$ of the server has the structure $((① \oplus ②) \oplus ③) \oplus ④$. The sufficient property for the server is defined as:

$$S_i^{\subseteq}(F) := F \in \mathbb{F}_i^{idle} \vee F \in \mathbb{F}_i^{busy} \qquad \text{where}$$

$$\mathbb{F}_i^{idle} := \Big\{ \big(\langle a_i.id_*^{in}.s_i.x_*, b_i.s_i.id_*^{out}.f(x_*)\rangle^*, X\big) \Big| X \subseteq \Sigma \setminus \{a_i.id^{in}.s_i.v \mid id^{in} \in ID \setminus \{s_i\} \wedge v \in \mathbb{T}\} \Big\}$$

$$\mathbb{F}_i^{busy} := \Big\{ \big(\langle a_i.id_*^{in}.s_i.x_*, b_i.s_i.id_*^{out}.f(x_*)\rangle^* \frown \langle a_i.id^{in}.s_i.v\rangle, X\big) \Big| X \subseteq \Sigma \setminus \{b_i.s_i.id^{out}.f(v) \mid id^{out} \in ID \setminus \{s_i\}\} \wedge id^{in} \in ID \setminus \{s_i\} \wedge v \in \mathbb{T} \Big\}$$

For the proof using the Hoare calculus, we require pre- and postconditions for each instruction. We consider pre- and postconditions, where the broad idea is that, e. g., $Post_1$ implies Pre_2, as we expect linear execution of the program (apart from the loop back from

④ to ①).

$$\{Pre_1\}\ ①\ \{Post_1\} \quad \{Pre_2\}\ ②\ \{Post_2\} \quad \{Pre_3\}\ ③\ \{Post_3\} \quad \{Pre_4\}\ ④\ \{Post_4\}$$

The communication failures, which belong to the invariant, are taken care of in the sequential composition rule. Also, as the program is looping, $Post_4$ implies Pre_1. Figure A.4 shows the complete outline of the proof tree using our Hoare calculus (all shorthands for assertions will be defined in the respective parts). We start at the leaves of the proof tree (the single instructions) and work our way to the root (the assertion to prove). At the beginning of each part, we recall the associated Hoare calculus rule.

Proof: ① – 1 : $\texttt{comm}_r\ a_i\ x$

H-COMM

$$\frac{\begin{array}{l} P(tr,\sigma,X) \equiv \neg(N(\sigma) \wedge \sigma_{pc} = \ell) \longrightarrow Q(tr,\sigma,X) \wedge \\ \qquad N(\sigma) \wedge \sigma_{pc} = \ell \ \longrightarrow \Big(\big(\forall Y \subseteq \Sigma \setminus f_{ev}(\sigma_{ds}).\ Q(tr,\sigma^C,Y)\big) \wedge \\ \qquad\qquad \Big(\forall \sigma'.\ N(\sigma') \wedge \sigma'_{pc} = \ell + 1 \wedge \\ \qquad\qquad\qquad ev \in f_{ev}(\sigma_{ds}) \wedge \sigma'_{ds} = f_{ds}(\sigma_{ds}, ev) \\ \qquad\qquad\qquad \longrightarrow \big(\forall Y \subseteq \Sigma.\ Q(tr^\frown\langle ev\rangle, \sigma', Y)\big)\Big)\Big) \end{array}}{\{P\}\ \ell : \texttt{comm}\ f_{ev}\ f_{ds}\ \{Q\}}$$

We unfold the definition of $\texttt{comm}_r$ to be able to apply H-COMM. We obtain the definitions of f_{ev} and f_{ds}.

$$\begin{aligned} \texttt{comm}_r\ a_i\ x &\coloneqq \texttt{comm}\ f_{ev}\ f_{ds} \qquad \text{with} \\ f_{ev} &\coloneqq \lambda\sigma_{ds}.\ \{a_i.id^{in}.s_i.v \mid id^{in} \in ID \setminus \{s_i\} \wedge v \in \mathbb{T}\} \\ f_{ds} &\coloneqq \lambda\sigma_{ds}\ ev.\ \sigma_{ds}[x \coloneqq val(ev)] \end{aligned}$$

We first define the pre- and postcondition Pre_1 and $Post_1$ that we want to show and then derive P_1 and Q_1, which we will plug into H-COMM. The precondition Pre_1 matches with the overall precondition ($(\langle\rangle, X \in \mathbb{F}_i^{idle}$ for some X), as we are considering the first instruction to be executed. The postcondition $Post_1$ describes the communication failures and terminal failures resulting form the execution. The communication failures have a communication state ($\neg N(\sigma)$) and the same program counter as the initial states ($\sigma_{pc} = 1$). As the program is now offering events, it cannot refuse everything, thus, both the considered trace tr *and* the considered refusal set X are as a pair in $\mathbb{F}_i^{idle}$ ($(tr, X) \in \mathbb{F}_i^{idle}$). The terminal failures have a normal state ($N(\sigma)$), the program counter is increased by one ($\sigma_{pc} = 2$), and the data store is updated to reflect the reception of the event $\Big(\sigma_{ds}(x) = val\big(last(tr)\big)\Big)$. The trace tr was extended by the communicated event and is now in $\mathbb{F}_i^{busy}$ ($(tr, \_) \in \mathbb{F}_i^{busy}$). As we only consider one instruction, the failures after the

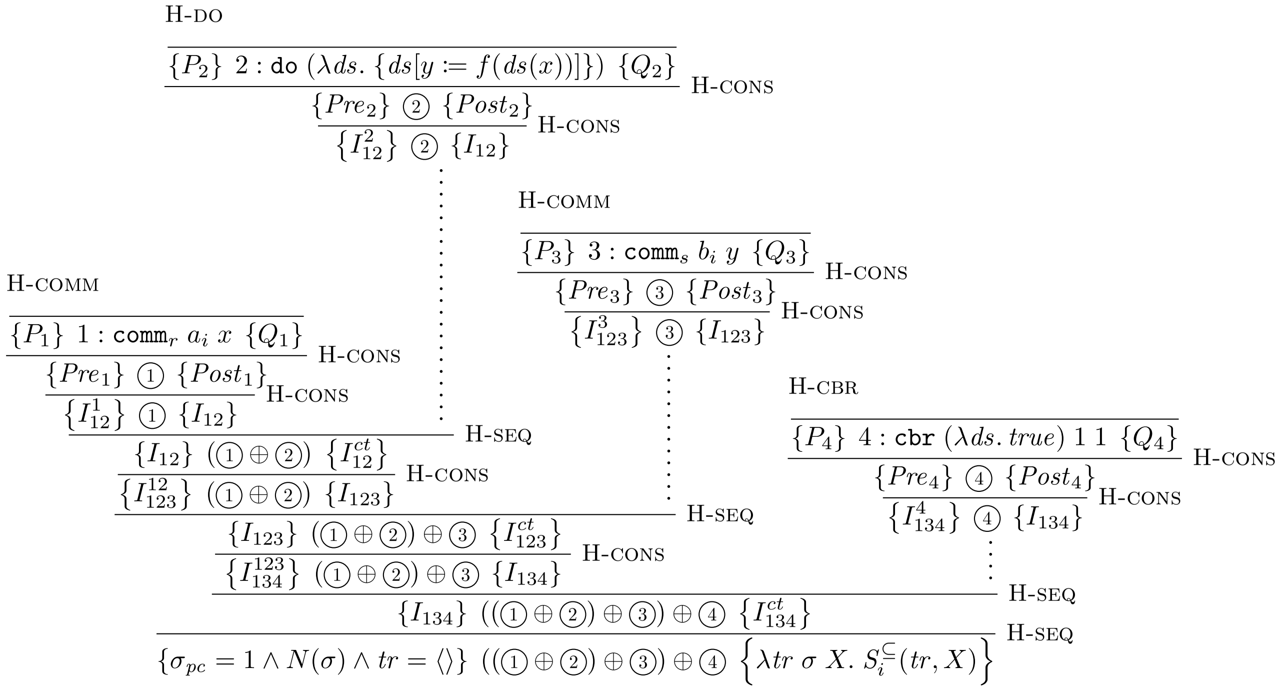


Figure A.4: Complete Proof Outline of the Hoare Calculus Proof for the Server

Proof: ① – $1 : \texttt{comm}_r\ a_i\ x$

execution of the instruction is terminal, so we can refuse all events ($X \subseteq \Sigma$).

$$\begin{aligned} Pre_1(tr, \sigma, X) &:= \quad N(\sigma) \wedge \sigma_{pc} = 1 \wedge (tr, \_) \in \mathbb{F}_i^{idle} \wedge X \subseteq \Sigma \\ Post_1(tr, \sigma, X) &:= \neg N(\sigma) \wedge \sigma_{pc} = 1 \wedge (tr, X) \in \mathbb{F}_i^{idle} \vee \\ &\qquad N(\sigma) \wedge \sigma_{pc} = 2 \wedge (tr, \_) \in \mathbb{F}_i^{busy} \wedge \sigma_{ds}(x) = val\big(last(tr)\big) \wedge X \subseteq \Sigma \end{aligned}$$

We directly use the postcondition $Post_1$ that we have defined as the postcondition Q_1 in the rule H-COMM. We derive P_1 from the rule H-COMM and Q_1. We have designed pre_1 such that $Pre_1 \Longrightarrow P_1$ holds. The main difference between P_1 and Pre_1 is that P_1 additionally accounts for the case that the instruction will not be executed ($\neg(N(\sigma) \wedge \sigma_{pc} = 1)$).

$$\begin{aligned} Q_1(tr, \sigma, X) &:= Post_1(tr, \sigma, X) \\ P_1(tr, \sigma, X) &:= \neg(N(\sigma) \wedge \sigma_{pc} = 1) \longrightarrow Q_1(tr, \sigma, X) \wedge \\ &\qquad N(\sigma) \wedge \sigma_{pc} = 1 \ \longrightarrow \\ &\qquad \Big(\big(\forall Y \subseteq \Sigma \setminus \{a_i.id^{in}.s_i.v \mid id^{in} \in ID \setminus \{s_i\} \wedge v \in \mathbb{T}\}.\ Q_1(tr, \sigma^C, Y)\big) \wedge \\ &\qquad \Big(\forall \sigma'.\ N(\sigma') \wedge \sigma'_{pc} = 2 \wedge \sigma'_{ds} = \sigma_{ds}[x := val(ev)] \wedge \\ &\qquad\qquad ev \in \{a_i.id^{in}.s_i.v \mid id^{in} \in ID \setminus \{s_i\} \wedge v \in \mathbb{T}\} \\ &\qquad\qquad \longrightarrow \big(\forall Y \subseteq \Sigma.\ Q_1(tr^\frown\langle ev\rangle, \sigma', Y)\big)\Big)\Big) \end{aligned}$$

We recall the rule of consequence of our Hoare calculus.

$$\text{H-CONS}\quad \frac{P \longrightarrow P' \qquad \{P'\}\ sp\ \{Q'\} \qquad Q' \longrightarrow Q}{\{P\}\ sp\ \{Q\}}$$

We can now apply the rules H-COMM and H-CONS to prove

$$\{Pre_1\}\ 1 : \texttt{comm}_r\ a_i\ x\ \{Post_1\}\,.$$

Below we show the excerpt we have proven of the overall proof tree.

$$\begin{array}{c} \text{H-COMM} \\ \dfrac{\dfrac{}{\{P_1\}\ 1 : \texttt{comm}_r\ a_i\ x\ \{Q_1\}}}{\{Pre_1\}\ ①\ \{Post_1\}}\ \text{H-CONS} \\ \vdots \end{array}$$

□

Proof: ② – $2 : \texttt{do}\ (\lambda ds.\ \{ds[y \coloneqq f(ds(x))]\})$

$$\frac{\begin{array}{l}\text{H-DO}\\ P(tr,\sigma,X) \equiv \neg(N(\sigma) \wedge \sigma_{pc} = \ell) \longrightarrow Q(tr,\sigma,X) \wedge \\ \qquad N(\sigma) \wedge \sigma_{pc} = \ell \ \longrightarrow \big(\forall \sigma'.\ N(\sigma') \wedge \sigma'_{pc} = \ell + 1 \wedge \sigma'_{ds} \in f(\sigma_{ds}) \\ \qquad\qquad \longrightarrow \forall Y \subseteq \Sigma.\ Q(tr,\sigma',Y)\big)\end{array}}{\{P\}\ \ell : \texttt{do}\ \ f\ \{Q\}}$$

We first define the pre- and postcondition Pre_2 and $Post_2$ that we want to show and then derive P_2 and Q_2, which we will plug into H-DO. The precondition Pre_2 keeps the communication invariant in form of $(tr, \_) \in \mathbb{F}_i^{busy}$ and ensure that the communicated value is stored in x $\Big(\sigma_{ds}(x) = val\big(last(tr)\big)\Big)$. The postcondition $Post_2$ describes only terminal failures resulting form the execution, as `do` does not offer communication. The trace is not changed, so we keep $(tr, \_) \in \mathbb{F}_i^{busy}$. However, we now ensure that the result of the computation is stored in y $\Big(\sigma_{ds}(y) = f\Big(val\big(last(tr)\big)\Big)\Big)$.

$$Pre_2(tr,\sigma,X) \coloneqq N(\sigma) \wedge \sigma_{pc} = 2 \wedge (tr, \_) \in \mathbb{F}_i^{busy} \wedge \sigma_{ds}(x) = val\big(last(tr)\big) \wedge X \subseteq \Sigma$$
$$Post_2(tr,\sigma,X) \coloneqq N(\sigma) \wedge \sigma_{pc} = 3 \wedge (tr, \_) \in \mathbb{F}_i^{busy} \wedge \sigma_{ds}(y) = f\Big(val\big(last(tr)\big)\Big) \wedge X \subseteq \Sigma$$

We directly use the postcondition $Post_2$ that we have defined as the postcondition Q_2 in the rule H-DO. We derive P_2 from the rule H-DO and Q_2. We have designed pre_2 such that $Pre_2 \Longrightarrow P_2$ holds. Again, the main difference between P_2 and Pre_2 is that P_2 additionally accounts for the case that the instruction will not be executed ($\neg(N(\sigma) \wedge \sigma_{pc} = 2)$).

$$\begin{aligned}
&Q_2(tr,\sigma,X) \coloneqq Post_2(tr,\sigma,X)\\
&P_2(tr,\sigma,X) \coloneqq \neg(N(\sigma) \wedge \sigma_{pc} = 2) \longrightarrow Q_2(tr,\sigma,X) \wedge \\
&\qquad N(\sigma) \wedge \sigma_{pc} = 2 \ \longrightarrow \Big(\forall \sigma'.\ N(\sigma') \wedge \sigma'_{pc} = 3 \wedge \sigma'_{ds} \in \big\{\sigma_{ds}[x \coloneqq val(ev)]\big\} \\
&\qquad\qquad \longrightarrow \big(\forall Y \subseteq \Sigma.\ Q_2(tr,\sigma',Y)\big)\Big)
\end{aligned}$$

We can now apply the rules H-DO and H-CONS to prove

$$\{Pre_2\}\ 2 : \texttt{do}\ (\lambda ds.\ \{ds[y \coloneqq f(ds(x))]\})\ \{Post_2\}$$

Below we show the excerpt we have proven of the overall proof tree.

$$\frac{\text{H-DO}\ \dfrac{}{\{P_2\}\ 2 : \texttt{do}\ (\lambda ds.\ \{ds[y \coloneqq f(ds(x))]\})\ \{Q_2\}}}{\{Pre_2\}\ ②\ \{Post_2\}}\ \text{H-CONS}$$
$$\vdots$$

□

Proof: $① \oplus ② - 1 : \texttt{comm}_r\ a_i\ x \oplus 2 : \texttt{do}\ (\lambda ds.\ \{ds[y \coloneqq f(ds(x))]\})$

$$\text{H-SEQ}\ \frac{\begin{array}{c}\{\lambda(tr,\sigma,X).\ I(tr,\sigma,X) \wedge \sigma_{pc} \in_{pc} sp_1 \wedge N(\sigma)\}\ sp_1\ \{I\} \\ \{\lambda(tr,\sigma,X).\ I(tr,\sigma,X) \wedge \sigma_{pc} \in_{pc} sp_2 \wedge N(\sigma)\}\ sp_2\ \{I\}\end{array}}{\{I\}\ sp_1 \oplus sp_2\ \{\lambda(tr,\sigma,X).\ I(tr,\sigma,X) \wedge (N(\sigma) \longrightarrow \sigma_{pc} \notin_{pc} sp_1 \oplus sp_2)\}}$$

We first construct the invariant I_{12} that we will use in the rule H-SEQ. Then, we demonstrate how parts of the invariant I_{12} are related to the pre- and postconditions of the two instructions. The invariant I_{12} is a disjunction of the pre- and postconditions of the two instructions.

$$I_{12}(tr,\sigma,X) \coloneqq Pre_1(tr,\sigma,X) \vee Post_1(tr,\sigma,X) \vee Pre_2(tr,\sigma,X) \vee Post_2(tr,\sigma,X)$$

We select parts of the invariant by combining it with, e. g., an assertion about the program counter. To be able to apply the rule H-CONS in the following, it is crucial that the postcondition of the first instruction agrees with the precondition of the second instruction of normal failures, i. e.,

$$Post_1(tr,\sigma,X) \wedge N(\sigma) \wedge pc = 2 \Longrightarrow Pre_2(tr,\sigma,X) \wedge N(\sigma) \wedge pc = 2$$

We demonstrate the relation of the invariant I_{12} and the pre- and postconditions. The last version of the invariant only contains the communication and terminal failures (all former terminal failures are removed).

$$\begin{aligned}
I_{12}^1 &\coloneqq I_{12}(tr,\sigma,X) \wedge \sigma_{pc} = 1 \wedge N(\sigma) \equiv Pre_1(tr,\sigma,X) \\
I_{12}^2 &\coloneqq I_{12}(tr,\sigma,X) \wedge \sigma_{pc} = 2 \wedge N(\sigma) \equiv Pre_2(tr,\sigma,X) \equiv Post_1(tr,\sigma,X) \wedge \sigma_{pc} = 2 \wedge N(\sigma) \\
I_{12}^{ct} &\coloneqq I_{12}(tr,\sigma,X) \wedge (N(\sigma) \longrightarrow \sigma_{pc} \notin \{1,2\}) \equiv Post_1(tr,\sigma,X) \wedge \neg N(\sigma) \vee Post_2(tr,\sigma,X)
\end{aligned}$$

We can now apply the rule H-CONS to show the premises of the rule H-SEQ and then apply the rule H-SEQ to prove

$$\{I_{12}\}\ ① \oplus ②\ \{I_{12}^{ct}\}\,.$$

Below we show the excerpt we have proven of the overall proof tree.

$$\text{H-SEQ}\ \frac{\text{H-CONS}\ \dfrac{\text{H-CONS}\ \dfrac{\vdots}{\{Pre_1\}\ ①\ \{Post_1\}}}{\{I_{12}^1\}\ ①\ \{I_{12}\}} \qquad \text{H-CONS}\ \dfrac{\text{H-CONS}\ \dfrac{\vdots}{\{Pre_2\}\ ②\ \{Post_2\}}}{\{I_{12}^2\}\ ②\ \{I_{12}\}}}{\{I_{12}\}\ ① \oplus ②\ \{I_{12}^{ct}\}}$$

$$\vdots$$

□

Proof: ③ – 3 : $\texttt{comm}_s\ b_i\ y$

H-COMM

$$\frac{\begin{array}{l} P(tr,\sigma,X) \equiv \neg(N(\sigma) \wedge \sigma_{pc} = \ell) \longrightarrow Q(tr,\sigma,X) \wedge \\ \qquad N(\sigma) \wedge \sigma_{pc} = \ell \ \longrightarrow \Big(\big(\forall Y \subseteq \Sigma \setminus f_{ev}(\sigma_{ds}).\ Q(tr,\sigma^C,Y)\big) \wedge \\ \qquad\qquad \Big(\forall \sigma'.\ N(\sigma') \wedge \sigma'_{pc} = \ell + 1 \wedge \\ \qquad\qquad\qquad ev \in f_{ev}(\sigma_{ds}) \wedge \sigma'_{ds} = f_{ds}(\sigma_{ds}, ev) \\ \qquad\qquad\qquad \longrightarrow \big(\forall Y \subseteq \Sigma.\ Q(tr^\frown\langle ev\rangle, \sigma', Y)\big)\Big)\Big) \end{array}}{\{P\}\ \ell : \texttt{comm}\ f_{ev}\ f_{ds}\ \{Q\}}$$

We unfold the definition of $\texttt{comm}_s$ to be able to apply H-COMM. We obtain the definitions of f_{ev} and f_{ds}.

$$\begin{aligned} \texttt{comm}_s\ a_i\ y &\coloneqq \texttt{comm}\ f_{ev}\ f_{ds} \qquad \text{with} \\ f_{ev} &\coloneqq \lambda\sigma_{ds}.\ \big\{b_i.s_i.id^{out}.\sigma_{ds}(y) \mid id^{out} \in ID \setminus \{s_i\}\big\} \\ f_{ds} &\coloneqq \lambda\sigma_{ds}\ ev.\ \sigma_{ds} \end{aligned}$$

We first define the pre- and postcondition Pre_3 and $Post_3$ that we want to show and then derive P_3 and Q_3, which we will plug into H-COMM. The precondition Pre_3 matches with the normal part of the postcondition $Post_2$ of the previous instruction. As for ①, the postcondition $Post_3$ describes the communication failures and terminal failures resulting form the execution. The communication failures have a communication state ($\neg N(\sigma)$) and the same program counter as the initial states ($\sigma_{pc} = 3$). As the instruction is offering events, it cannot refuse everything and, thus, both the considered trace tr and the considered refusal set X are as a pair in $\mathbb{F}_i^{busy}$ ($(tr, X) \in \mathbb{F}_i^{busy}$). The terminal failures have a normal state ($N(\sigma)$), the program counter is increased by one ($\sigma_{pc} = 4$). The data store remains unchanged. The relation between the received value v and sent value $f(v)$ is now captured in the trace (ensured by $\mathbb{F}_i^{idle}$), so we do not need to additionally record it.

$$\begin{aligned} Pre_3(tr,\sigma,X) &\coloneqq \quad N(\sigma) \wedge \sigma_{pc} = 3 \wedge (tr, \_) \in \mathbb{F}_i^{busy} \wedge \sigma_{ds}(y) = f\Big(val\big(last(tr)\big)\Big) \wedge X \subseteq \Sigma \\ Post_3(tr,\sigma,X) &\coloneqq \neg N(\sigma) \wedge \sigma_{pc} = 3 \wedge (tr, X) \in \mathbb{F}_i^{busy} \vee \\ &\qquad\quad N(\sigma) \wedge \sigma_{pc} = 4 \wedge (tr, \_) \in \mathbb{F}_i^{idle} \wedge X \subseteq \Sigma \end{aligned}$$

We directly use the postcondition $Post_3$ that we have defined as the postcondition Q_3 in the rule H-COMM. We derive P_3 from the rule H-COMM and Q_3. We have designed pre_3

Proof: ③ – $3 : \texttt{comm}_s\ b_i\ y$

such that $Pre_3 \Longrightarrow P_3$ holds.

$$
\begin{aligned}
Q_3(tr, \sigma, X) &\coloneqq Post_3(tr, \sigma, X) \\
P_3(tr, \sigma, X) &\coloneqq \neg(N(\sigma) \wedge \sigma_{pc} = 3) \longrightarrow Q_3(tr, \sigma, X) \wedge \\
&\quad N(\sigma) \wedge \sigma_{pc} = 3 \ \longrightarrow \\
&\quad \Big(\big(\forall Y \subseteq \Sigma \setminus \{b_i.s_i.id^{out}.\sigma_{ds}(y) \mid id^{out} \in ID \setminus \{s_i\}\}.\ Q_3(tr, \sigma^C, Y)\big) \wedge \\
&\quad \Big(\forall \sigma'.\ N(\sigma') \wedge \sigma'_{pc} = 4 \wedge \sigma'_{ds} = \sigma_{ds} \wedge \\
&\qquad ev \in \{b_i.s_i.id^{out}.\sigma_{ds}(y) \mid id^{out} \in ID \setminus \{s_i\}\} \\
&\qquad \longrightarrow \big(\forall Y \subseteq \Sigma.\ Q_3(tr^\frown\langle ev\rangle, \sigma', Y)\big)\Big)\Big)
\end{aligned}
$$

We can now apply the rules H-COMM and H-CONS to prove

$$\{Pre_3\}\ 1 : \texttt{comm}_s\ b_i\ y\ \{Post_3\}\,.$$

Below we show the excerpt we have proven of the overall proof tree.

$$
\dfrac{\dfrac{\text{H-COMM}}{\{P_3\}\ 3 : \texttt{comm}_s\ b_i\ y\ \{Q_3\}}}{\{Pre_3\}\ ③\ \{Post_3\}}\ \text{H-CONS}
$$

$$\vdots$$

□

Proof: $(① \oplus ②) \oplus ③$

$$
\dfrac{\begin{array}{c}\text{H-SEQ} \\ \{\lambda(tr, \sigma, X).\ I(tr, \sigma, X) \wedge \sigma_{pc} \in_{pc} sp_1 \wedge N(\sigma)\}\ sp_1\ \{I\} \\ \{\lambda(tr, \sigma, X).\ I(tr, \sigma, X) \wedge \sigma_{pc} \in_{pc} sp_2 \wedge N(\sigma)\}\ sp_2\ \{I\}\end{array}}{\{I\}\ sp_1 \oplus sp_2\ \{\lambda(tr, \sigma, X).\ I(tr, \sigma, X) \wedge (N(\sigma) \longrightarrow \sigma_{pc} \notin_{pc} sp_1 \oplus sp_2)\}}
$$

Similar to the case $① \oplus ②$, we first construct the invariant I_{123} that we will use in the rule H-SEQ. Then, we demonstrate how parts of the invariant I_{123} are related to the pre- and postconditions of the three instructions. The invariant I_{123} is a disjunction of the pre- and postconditions of the three instructions. The precondition Pre_1 serves as precondition for the component $① \oplus ②$. The communication part of the postcondition $Post_1$ (which we obtain by writing $Post_1(tr, \sigma, X) \wedge \neg N(\sigma)$) describes the communication capabilities of the component $① \oplus ②$. As postcondition for the component $① \oplus ②$ we use only $Post_2$. The pre- and postcondition of instruction ③ are simply the pre- and postcondition of

Proof: $(\textcircled{1} \oplus \textcircled{2}) \oplus \textcircled{3}$

instruction $\textcircled{3}$.

$$I_{123}(tr, \sigma, X) \coloneqq Pre_1(tr, \sigma, X) \lor Post_1(tr, \sigma, X) \land \neg N(\sigma) \lor Post_2(tr, \sigma, X) \lor Pre_3(tr, \sigma, X) \lor Post_3(tr, \sigma, X)$$

We demonstrate the relation of the invariant I_{123} and the individual pre- and postconditions. The last version of the invariant only contains the communication and terminal failures (all former terminal failures are removed).

$$\begin{aligned} I^{12}_{123} &\coloneqq I_{123}(tr, \sigma, X) \land \sigma_{pc} \in \{1, 2\} \land N(\sigma) \equiv Pre_1(tr, \sigma, X) \\ I^{3}_{123} &\coloneqq I_{123}(tr, \sigma, X) \land \sigma_{pc} = 3 \land N(\sigma) \equiv Pre_3(tr, \sigma, X) \equiv Post_2(tr, \sigma, X) \\ I^{ct}_{123} &\coloneqq I_{123}(tr, \sigma, X) \land (N(\sigma) \longrightarrow \sigma_{pc} \notin \{1, 2, 3\}) \\ &\equiv Post_1(tr, \sigma, X) \land \neg N(\sigma) \lor Post_3(tr, \sigma, X) \end{aligned}$$

We can now apply the rule H-CONS to show the premises of the rule H-SEQ and then apply the rule H-SEQ to prove $\{I_{123}\}\ (\textcircled{1} \oplus \textcircled{2}) \oplus \textcircled{3}\ \{I^{ct}_{123}\}$. First, we recall the definitions of I_{12} and I^{ct}_{12}:

$$\begin{aligned} I_{12}(tr, \sigma, X) &\coloneqq Pre_1(tr, \sigma, X) \lor Post_1(tr, \sigma, X) \lor Pre_2(tr, \sigma, X) \lor Post_2(tr, \sigma, X) \\ I^{ct}_{12} &\coloneqq I_{12}(tr, \sigma, X) \land (N(\sigma) \longrightarrow \sigma_{pc} \notin \{1, 2\}) \\ &\equiv Post_1(tr, \sigma, X) \land \neg N(\sigma) \lor Post_2(tr, \sigma, X) \end{aligned}$$

Below we show the excerpt we have proven of the overall proof tree.

$$\dfrac{\dfrac{\dfrac{\vdots}{\{I_{12}\}\ (\textcircled{1} \oplus \textcircled{2})\ \{I^{ct}_{12}\}}\ \text{H-SEQ}}{\{I^{12}_{123}\}\ (\textcircled{1} \oplus \textcircled{2})\ \{I_{123}\}}\ \text{H-CONS} \qquad \dfrac{\dfrac{\vdots}{\{Pre_3\}\ \textcircled{3}\ \{Post_3\}}\ \text{H-CONS}}{\{I^{3}_{123}\}\ \textcircled{3}\ \{I_{123}\}}\ \text{H-CONS}}{\{I_{123}\}\ (\textcircled{1} \oplus \textcircled{2}) \oplus \textcircled{3}\ \{I^{ct}_{123}\}}\ \text{H-SEQ}$$

$\vdots$

□

Proof: $\textcircled{4} - 4 : \texttt{cbr}\ (\lambda ds.\ true)\ 1\ 1$

$$\text{H-CBR}\ \dfrac{\begin{aligned} P(tr, \sigma, X) \equiv &\neg(N(\sigma) \land \sigma_{pc} = \ell) \longrightarrow Q(tr, \sigma, X) \land \\ &N(\sigma) \land \sigma_{pc} = \ell \ \longrightarrow \big(\forall \sigma'.\ (b\ \sigma_{ds} \land \sigma'_{pc} = m \lor \neg(b\ \sigma_{ds}) \land \sigma'_{pc} = n) \land \\ &\qquad N(\sigma') \land \sigma_{ds} = \sigma'_{ds} \longrightarrow \forall Y \subseteq \Sigma.\ Q(tr, \sigma', Y)\big) \end{aligned}}{\{P\}\ \ell : \texttt{cbr}\ \ b\ m\ n\ \{Q\}}$$

Proof: ④ – $4 : \texttt{cbr}\ (\lambda ds.\, true)\ 1\ 1$

We first define the pre- and postcondition Pre_4 and $Post_4$ that we want to show and then derive P_4 and Q_4, which we will plug into H-CBR. As we use instruction ④ to construct a loop, the postcondition $Post_4$ coincides with the precondition Pre_1 of instruction ①.

$$\begin{aligned} Pre_4(tr, \sigma, X) &\coloneqq N(\sigma) \wedge \sigma_{pc} = 4 \wedge (tr, \_) \in \mathbb{F}_i^{idle} \wedge X \subseteq \Sigma \\ Post_4(tr, \sigma, X) &\coloneqq N(\sigma) \wedge \sigma_{pc} = 1 \wedge (tr, \_) \in \mathbb{F}_i^{idle} \wedge X \subseteq \Sigma \equiv Pre_1(tr, \sigma, X) \end{aligned}$$

We directly use the postcondition $Post_4$ that we have defined as the postcondition Q_4 in the rule H-CBR. We derive P_4 from the rule H-CBR and Q_4. We have designed pre_4 such that $Pre_4 \Longrightarrow P_4$ holds.

$$\begin{aligned} Q_4(tr, \sigma, X) &\coloneqq Post_4(tr, \sigma, X) \equiv Pre_1(tr, \sigma, X) \\ P_4(tr, \sigma, X) &\coloneqq \neg(N(\sigma) \wedge \sigma_{pc} = 4) \longrightarrow Q_4(tr, \sigma, X) \wedge \\ &\qquad N(\sigma) \wedge \sigma_{pc} = 4 \ \longrightarrow \Big(\forall \sigma'.\, N(\sigma') \wedge \sigma'_{pc} = 1 \wedge \sigma'_{ds} = \sigma_{ds} \\ &\qquad\qquad \longrightarrow \big(\forall Y \subseteq \Sigma.\, Q_4(tr, \sigma', Y)\big)\Big) \end{aligned}$$

We can now apply the rules H-CBR and H-CONS to prove

$$\{Pre_4\}\ 4 : \texttt{cbr}\ (\lambda ds.\, true)\ 1\ 1\ \{Post_4\}\,.$$

Below we show the excerpt we have proven of the overall proof tree.

$$\dfrac{\dfrac{\text{H-CBR}}{\{P_4\}\ 4 : \texttt{cbr}\ (\lambda ds.\, true)\ 1\ 1\ \{Q_4\}}}{\{Pre_4\}\ ④\ \{Post_4\}}\ \text{H-CONS}$$
$$\vdots$$

□

Proof: $((① \oplus ②) \oplus ③) \oplus ④$

$$\dfrac{\begin{array}{c}\{\lambda(tr, \sigma, X).\, I(tr, \sigma, X) \wedge \sigma_{pc} \in_{pc} sp_1 \wedge N(\sigma)\}\ sp_1\ \{I\} \\ \{\lambda(tr, \sigma, X).\, I(tr, \sigma, X) \wedge \sigma_{pc} \in_{pc} sp_2 \wedge N(\sigma)\}\ sp_2\ \{I\}\end{array}}{\{I\}\ sp_1 \oplus sp_2\ \{\lambda(tr, \sigma, X).\, I(tr, \sigma, X) \wedge (N(\sigma) \longrightarrow \sigma_{pc} \notin_{pc} sp_1 \oplus sp_2)\}}\ \text{H-SEQ}$$

Similar to the previous applications of the rule H-SEQ, we first construct the invariant I_{134}. Then, we demonstrate how parts of the invariant I_{134} are related to the pre- and postconditions of the four instructions. The invariant I_{134} is a disjunction of some of the pre- and postconditions of the four instructions. Note that the invariant does only require the pre- and postcondition of both components (think glue points) and the assertions

Proof: $((① \oplus ②) \oplus ③) \oplus ④$

about the communication behavior. The precondition Pre_1 serves as precondition for the Component $(① \oplus ②) \oplus ③$. The communication part of the postconditions $Post_1$ and $Post_3$ describes the communication capabilities of the Component $(① \oplus ②) \oplus ③$. The normal part of the postcondition $Post_3$ serves as a postcondition for the Component $(① \oplus ②) \oplus ③$. The precondition of the component consisting of Instruction ④ is simply the precondition of Instruction ④. We omit the postcondition $Post_4$ as it is equivalent to the precondition Pre_1 of Instruction ①.

$$\begin{aligned} I_{134}(tr, \sigma, X) \coloneqq\ & Pre_1(tr, \sigma, X) \lor Post_1(tr, \sigma, X) \land \neg N(\sigma) \lor Post_3(tr, \sigma, X) \lor \\ & Pre_4(tr, \sigma, X) \end{aligned}$$

We demonstrate the relation of the invariant I_{134} and the individual pre- and postconditions. The last version of the invariant only contains the communication failures. As the program is looping, there are no terminal failures anymore.

$$\begin{aligned} I_{134}^{123} &\coloneqq I_{134}(tr, \sigma, X) \land \sigma_{pc} \in \{1, 2, 3\} \land N(\sigma) \equiv Pre_1(tr, \sigma, X) \\ I_{134}^{4} &\coloneqq I_{134}(tr, \sigma, X) \land \sigma_{pc} = 4 \land N(\sigma) \equiv Pre_4(tr, \sigma, X) \equiv Post_3(tr, \sigma, X) \land N(\sigma) \\ I_{134}^{ct} &\coloneqq I_{134}(tr, \sigma, X) \land (N(\sigma) \longrightarrow \sigma_{pc} \notin \{1, 2, 3, 4\}) \\ &\equiv \neg N(\sigma) \land \big(Post_1(tr, \sigma, X) \lor Post_3(tr, \sigma, X)\big) \end{aligned}$$

We can now apply the rule H-CONS to show the premises of the rule H-SEQ and then apply the rule H-SEQ to prove $\{I_{134}\}\ \texttt{comm}_r\ \{I_{134}^{ct}\}$. First, we recall the definitions of I_{123} and I_{123}^{ct}:

$$\begin{aligned} I_{123}(tr, \sigma, X) \coloneqq\ & Pre_1(tr, \sigma, X) \lor Post_1(tr, \sigma, X) \land \neg N(\sigma) \lor Post_2(tr, \sigma, X) \lor \\ & Pre_3(tr, \sigma, X) \lor Post_3(tr, \sigma, X) \\ I_{123}^{ct} \coloneqq\ & I_{123}(tr, \sigma, X) \land (N(\sigma) \longrightarrow \sigma_{pc} \notin \{1, 2, 3\}) \\ \equiv\ & Post_1(tr, \sigma, X) \land \neg N(\sigma) \lor Post_3(tr, \sigma, X) \end{aligned}$$

We apply the rule H-CONS one last time to prove the desired Hoare tripel

$$\{\sigma_{pc} = 1 \land N(\sigma) \land tr = \langle\rangle\}\ ((1 \oplus 2) \oplus 3) \oplus 4\ \left\{\lambda tr\ \sigma\ X.\ S_i^{\subseteq}(tr, X)\right\}.$$

$\sigma_{pc} = 1 \land N(\sigma) \land tr = \langle\rangle$ implies I_{134} by definition of Pre_1 and $\mathbb{F}_i^{idle}$.
From I_{134}^{ct} we can conclude $S_i^{\subseteq}$, as the following implication hold by the definition $Post_1$ and $Post_3$:

$$\begin{aligned} \neg N(\sigma) \land Post_1(tr, \sigma, X) &\Longrightarrow \mathbb{F}_i^{idle}(tr, \sigma, X) \\ \neg N(\sigma) \land Post_3(tr, \sigma, X) &\Longrightarrow \mathbb{F}_i^{busy}(tr, \sigma, X) \end{aligned}$$

Proof: $((\textcircled{1} \oplus \textcircled{2}) \oplus \textcircled{3}) \oplus \textcircled{4}$

We obtain the root of the overall proof tree.

$$
\dfrac{
\dfrac{
\dfrac{
\dfrac{\vdots}{\{I_{123}\}\ (\textcircled{1} \oplus \textcircled{2}) \oplus \textcircled{3}\ \{I^{ct}_{123}\}}\ \text{H-SEQ}
}{\{I^{123}_{134}\}\ (\textcircled{1} \oplus \textcircled{2}) \oplus \textcircled{3}\ \{I_{134}\}}\ \text{H-CONS}
\qquad
\dfrac{\vdots}{\{Pre_4\}\ \textcircled{4}\ \{Post_4\}}\ \text{H-CONS}
}{\{I_{134}\}\ ((\textcircled{1} \oplus \textcircled{2}) \oplus \textcircled{3}) \oplus \textcircled{4}\ \{I^{ct}_{134}\}}\ \text{H-SEQ}
}{\{\sigma_{pc} = 1 \wedge N(\sigma) \wedge tr = \langle\rangle\}\ ((\textcircled{1} \oplus \textcircled{2}) \oplus \textcircled{3}) \oplus \textcircled{4}\ \left\{\lambda tr\ \sigma\ X.\ S_i^{\subseteq}(tr, X)\right\}}\ \text{H-SEQ}
$$

□

This concludes the proof for Lemma 7.2.

List of Definitions

List of Theorems and Assumptions

List of Examples

List of Figures

Bibliography

[ABEL05] Martín Abadi, Mihai Budiu, Úlfar Erlingsson, and Jay Ligatti. Control-flow integrity. In Vijay Atluri, Catherine A. Meadows, and Ari Juels, editors, *Proceedings of the 12th ACM Conference on Computer and Communications Security, CCS 2005, Alexandria, VA, USA, November 7-11, 2005*, pages 340–353. ACM, 2005. URL: `http://doi.acm.org/10.1145/1102120.1102165`, `doi:10.1145/1102120.1102165`.

[BBD+19] Nils Berg, Björn Bartels, Armin Danziger, Guilherme Grochau Azzi, and Matthias Bentert. Formal Verification of Low-Level Code in a Model-Based Refinement Process (Technical Report: Isabelle/HOL Formalization). Technical report, Technische Universität Berlin, 2019. `doi:10.14279/depositonce-8636`.

[BBO12] Samik Basu, Tevfik Bultan, and Meriem Ouederni. Synchronizability for verification of asynchronously communicating systems. In Viktor Kuncak and Andrey Rybalchenko, editors, *Verification, Model Checking, and Abstract Interpretation - 13th International Conference, VMCAI 2012, Philadelphia, PA, USA, January 22-24, 2012. Proceedings*, volume 7148 of *Lecture Notes in Computer Science*, pages 56–71. Springer, 2012. URL: `http://dx.doi.org/10.1007/978-3-642-27940-9_5`, `doi:10.1007/978-3-642-27940-9_5`.

[BCD+18] Roberto Baldoni, Emilio Coppa, Daniele Cono D'Elia, Camil Demetrescu, and Irene Finocchi. A survey of symbolic execution techniques. *ACM Comput. Surv.*, 51(3), 2018.

[BCN+17] Nathan Burow, Scott A. Carr, Joseph Nash, Per Larsen, Michael Franz, Stefan Brunthaler, and Mathias Payer. Control-flow integrity: Precision, security, and performance. *ACM Comput. Surv.*, 50(1):16:1–16:33, 2017. URL: `http://doi.acm.org/10.1145/3054924`, `doi:10.1145/3054924`.

[BG11] B. Bartels and S. Glesner. Verification of distributed embedded real-time systems and their low-level implementation using Timed CSP. In T. Dan Thu and K. Leung, editors, *Proceedings of the 18th Asia Pacific Software Engineering Conference (APSEC 2011)*, pages 195–202. IEEE Computer Society, 2011. `doi:10.1109/APSEC.2011.52`.

[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. URL: https://doi.org/10.1007/978-3-030-02450-5_18, doi:10.1007/978-3-030-02450-5_18.

[BJ14] Björn Bartels and Nils Jähnig. Mechanized, compositional verification of low-level code. In Julia M. Badger and Kristin Yvonne Rozier, editors, *NASA Formal Methods*, volume 8430 of *LNCS*, pages 98–112. Springer International Publishing, 2014. doi:10.1007/978-3-319-06200-6_8.

[BO01] Manfred Broy and Ernst-Rüdiger Olderog. Chapter 2 - trace-oriented models of concurrency. In J.A. Bergstra, A. Ponse, and S.A. Smolka, editors, *Handbook of Process Algebra*, pages 101 – 195. Elsevier Science, Amsterdam, 2001. URL: https://www.sciencedirect.com/science/article/pii/B9780444828309500205, doi:https://doi.org/10.1016/B978-044482830-9/50020-5.

[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. URL: https://doi.org/10.1007/BF00263584, doi:10.1007/BF00263584.

[BvW98] Ralph-Johan Back and Joakim von Wright. *Refinement Calculus - A Systematic Introduction*. Graduate Texts in Computer Science. Springer, 1998. URL: https://doi.org/10.1007/978-1-4612-1674-2, doi:10.1007/978-1-4612-1674-2.

[Cho18] Hyungmin Cho. Impact of microarchitectural differences of RISC-V processor cores on soft error effects. *IEEE Access*, 6:41302–41313, 2018. URL: https://doi.org/10.1109/ACCESS.2018.2858773, doi:10.1109/ACCESS.2018.2858773.

[Cor18] Intel Corporation. *Intel 64 and IA-32 Architectures Software Developer Manual*, 2018. URL: https://software.intel.com/en-us/articles/intel-sdm [cited 07.12.2018].

[dFG06] David de Frutos-Escrig and Carlos Gregorio-Rodríguez. Process equivalences as global bisimulations. *J. UCS*, 12(11):1521–1550, 2006. URL: http://dx.doi.org/10.3217/jucs-012-11-1521, doi:10.3217/jucs-012-11-1521.

[DK15] Alan A. A. Donovan and Brian W. Kernighan. *The Go Programming Language (Addison-Wesley Professional Computing Series)*. Addison-Wesley Professional, 2015. URL: http://www.gopl.io.

[FRC+18] Eric Flamand, Davide Rossi, Francesco Conti, Igor Loi, Antonio Pullini, Florent Rotenberg, and Luca Benini. GAP-8: A RISC-V soc for AI at the edge of the iot. In *29th IEEE International Conference on Application-specific Systems, Architectures and Processors, ASAP 2018, Milano, Italy, July 10-12, 2018*, pages 1–4. IEEE Computer Society, 2018. URL: https://doi.org/10.1109/ASAP.2018.8445101, doi:10.1109/ASAP.2018.8445101.

[GABR14] Thomas Gibson-Robinson, Philip J. Armstrong, Alexandre Boulgakov, and A. W. Roscoe. FDR3 - A modern refinement checker for CSP. In *Tools and Algorithms for the Construction and Analysis of Systems - 20th International Conference, TACAS 2014*, pages 187–201, 2014. doi:10.1007/978-3-642-54862-8_13.

[Gar03] William B. Gardner. Bridging CSP and C++ with selective formalism and executable specifications. In *1st ACM & IEEE International Conference on Formal Methods and Models for Co-Design (MEMOCODE 2003), 24-26 June 2003, Mont Saint-Michel, France, Proceedings*, page 237. IEEE Computer Society, 2003. URL: http://dx.doi.org/10.1109/MEMCOD.2003.1210108, doi:10.1109/MEMCOD.2003.1210108.

[GR01] Roberto Gorrieri and Arend Rensink. Action refinement. In J.A. Bergstra, A. Ponse, and S.A. Smolka, editors, *Handbook of Process Algebra*, pages 1047–1147. Elsevier, 2001.

[Hoa78] C. A. R. Hoare. Communicating sequential processes. *Commun. ACM*, 21(8):666–677, August 1978. doi:10.1145/359576.359585.

[HP17] John L. Hennessy and David A. Patterson. *Computer Architecture - A Quantitative Approach, 6th Edition*. Morgan Kaufmann, 2017.

[IR05] Yoshinao Isobe and Markus Roggenbach. A generic theorem prover of csp refinement. In Nicolas Halbwachs and LenoreD. Zuck, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, volume 3440 of *Lecture Notes in Computer Science*, pages 108–123. Springer Berlin Heidelberg, 2005. URL: http://dx.doi.org/10.1007/978-3-540-31980-1_8, doi:10.1007/978-3-540-31980-1_8.

[ISO09] ISO. ISO/DIS 26262 - Road Vehicles - Functional Safety. Technical report, International Organization for Standardization, Geneva, Switzerland, 7 2009.

[JGG15] Nils Jähnig, Thomas Göthel, and Sabine Glesner. A denotational semantics for communicating unstructured code. In Bara Buhnova, Lucia Happe, and Jan Kofron, editors, *Proceedings 12th International Workshop on Formal Engineering approaches to Software Components and Architectures, FESCA 2015, London, United Kingdom, April 12th, 2015.*, volume 178 of *EPTCS*, pages 9–21, 2015. URL: http://dx.doi.org/10.4204/EPTCS.178.2, doi:10.4204/EPTCS.178.2.

[JGG16] Nils Jähnig, Thomas Göthel, and Sabine Glesner. Refinement-based verification of communicating unstructured code. In *Software Engineering and Formal Methods - 14th International Conference, SEFM 2016, Held as Part of STAF 2016, Vienna, Austria, July 4-8, 2016, Proceedings*, pages 61–75, 2016. URL: http://dx.doi.org/10.1007/978-3-319-41591-8_5, doi:10.1007/978-3-319-41591-8_5.

[Jon81] Cliff B. Jones. *Developing methods for computer programs including a notion of interference*. PhD thesis, University of Oxford, UK, 1981. URL: http://ethos.bl.uk/OrderDetails.do?uin=uk.bl.ethos.259064.

[Kan88] Gerry Kane. *MIPS RISC Architecture*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1988.

[KBG+11] Moritz Kleine, Björn Bartels, Thomas Göthel, Steffen Helke, and Dirk Prenzel. *LLVM2CSP*: Extracting CSP models from concurrent programs. In Mihaela Gheorghiu Bobaru, Klaus Havelund, Gerard J. Holzmann, and Rajeev Joshi, editors, *NASA Formal Methods - Third International Symposium, NFM 2011, Pasadena, CA, USA, April 18-20, 2011. Proceedings*, volume 6617 of *Lecture Notes in Computer Science*, pages 500–505. Springer, 2011. URL: `https://doi.org/10.1007/978-3-642-20398-5_39`, `doi:10.1007/978-3-642-20398-5\_39`.

[LA04] Chris Lattner and Vikram S. Adve. LLVM: A compilation framework for lifelong program analysis & transformation. In *2nd IEEE / ACM International Symposium on Code Generation and Optimization (CGO 2004), 20-24 March 2004, San Jose, CA, USA*, pages 75–88. IEEE Computer Society, 2004. URL: `https://doi.org/10.1109/CGO.2004.1281665`, `doi:10.1109/CGO.2004.1281665`.

[Ler09] Xavier Leroy. Formal verification of a realistic compiler. *Communications of the ACM*, 52(7):107–115, 2009. URL: `http://gallium.inria.fr/~xleroy/publi/compcert-CACM.pdf`.

[Lim18] ARM Limited. *ARM Architecture Reference Manual*, 2018. URL: `https://developer.arm.com/docs/ddi0487/latest/` [cited 07.12.2018].

[LNTY17] Julien Lange, Nicholas Ng, Bernardo Toninho, and Nobuko Yoshida. Fencing off go: liveness and safety for channel-based programming. In Giuseppe Castagna and Andrew D. Gordon, editors, *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages, POPL 2017, Paris, France, January 18-20, 2017*, pages 748–761. ACM, 2017. URL: `http://dl.acm.org/citation.cfm?id=3009847`.

[MA17] Nima Mansube Astaneh. Extracting csp processes from communicating unstructured code. Diploma thesis, Technische Universität Berlin, 2017.

[Mil80] Robin Milner. *A Calculus of Communicating Systems*, volume 92 of *Lecture Notes in Computer Science*. Springer, 1980. URL: `https://doi.org/10.1007/3-540-10235-3`, `doi:10.1007/3-540-10235-3`.

[Mil83] Robin Milner. Calculi for synchrony and asynchrony. *Theor. Comput. Sci.*, 25:267–310, 1983. URL: `https://doi.org/10.1016/0304-3975(83)90114-7`, `doi:10.1016/0304-3975(83)90114-7`.

[Mil89] Robin Milner. *Communication and concurrency*. PHI Series in computer science. Prentice Hall, 1989.

[NBF96] Bradford Nichols, Dick Buttlar, and Jacqueline Proulx Farrell. *Pthreads Programming*. O'Reilly & Associates, Inc., Sebastopol, CA, USA, 1996.

[NK14] Tobias Nipkow and Gerwin Klein. *Concrete Semantics - With Isabelle/HOL*. Springer, 2014. URL: `http://www.concrete-semantics.org`, `doi:10.1007/978-3-319-10542-0`.

[NNH99] Flemming Nielson, Hanne Riis Nielson, and Chris Hankin. *Principles of program analysis*. Springer, 1999. URL: `https://doi.org/10.1007/978-3-662-03811-6`, `doi:10.1007/978-3-662-03811-6`.

[NPW02] Tobias Nipkow, Lawrence C. Paulson, and Markus Wenzel. *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*, volume 2283 of *LNCS*. Springer, 2002. `doi:10.1007/3-540-45949-9`.

[Pag09] Daniel Page. *A Practical Introduction to Computer Architecture*. Springer Publishing Company, Incorporated, 1st edition, 2009.

[Pee04] Ad M. G. Peeters. Implementation of handshake components. In Ali E. Abdallah, Cliff B. Jones, and Jeff W. Sanders, editors, *Communicating Sequential Processes: The First 25 Years, Symposium on the Occasion of 25 Years of CSP, London, UK, July 7-8, 2004, Revised Invited Papers*, volume 3525 of *Lecture Notes in Computer Science*, pages 98–132. Springer, 2004. URL: `http://dx.doi.org/10.1007/11423348_7`, `doi:10.1007/11423348_7`.

[Pet12] Kirstin Peters. *Translational Expressiveness. Comparing Process Calculi using Encodings*. PhD thesis, Berlin Institute of Technology, 2012. URL: `http://opus.kobv.de/tuberlin/volltexte/2012/3749/`.

[PKND18] Karyofyllis Patsidis, Dimitris Konstantinou, Chrysostomos Nicopoulos, and Giorgos Dimitrakopoulos. A low-cost synthesizable RISC-V dual-issue processor core leveraging the compressed instruction set extension. *Microprocessors and Microsystems - Embedded Hardware Design*, 61:1–10, 2018. `doi:10.1016/j.micpro.2018.05.007`.

[PS92] Joachim Parrow and Peter Sjödin. Multiway synchrinizaton verified with coupled simulation. In Rance Cleaveland, editor, *CONCUR '92, Third International Conference on Concurrency Theory, Stony Brook, NY, USA, August 24-27, 1992, Proceedings*, volume 630 of *Lecture Notes in Computer Science*, pages 518–533. Springer, 1992. URL: `https://doi.org/10.1007/BFb0084813`, `doi:10.1007/BFb0084813`.

[RE08] Willem-Paul de Roever and Kai Engelhardt. *Data Refinement: Model-Oriented Proof Methods and Their Comparison*. Cambridge University Press, New York, NY, USA, 1st edition, 2008.

[RG97] Arend Rensink and Roberto Gorrieri. Action refinement as an implementation relation. In Michel Bidoit and Max Dauchet, editors, *TAPSOFT'97: Theory and Practice of Software Development, 7th International Joint Conference CAAP/FASE, Lille, France, April 14-18, 1997, Proceedings*, volume 1214 of *Lecture Notes in Computer Science*, pages 772–786. Springer, 1997. URL: `https://doi.org/10.1007/BFb0030640`, `doi:10.1007/BFb0030640`.

[Rid10] Tom Ridge. A rely-guarantee proof system for x86-tso. In Gary T. Leavens, Peter W. O'Hearn, and Sriram K. Rajamani, editors, *Verified Software: Theories, Tools, Experiments, Third International Conference, VSTTE 2010, Edinburgh, UK, August 16-19, 2010. Proceedings*, volume 6217 of *Lecture Notes in Computer Science*, pages 55–70. Springer, 2010. URL: `https://doi.org/10.1007/978-3-642-15057-9_4`, `doi:10.1007/978-3-642-15057-9\_4`.

[Ros10] A. W. Roscoe. *Understanding Concurrent Systems*. Texts in Computer Science. Springer, 2010. URL: `https://doi.org/10.1007/978-1-84882-258-0`, `doi:10.1007/978-1-84882-258-0`.

[Ros18] Jothy Rosenberg. Risc-v: All hype or real hope for the processor market?, September 2018. URL: `https://www.allaboutcircuits.com/industry-articles/risc-v-all-hype-or-real-hope-for-the-processor-market/` [cited 22.11.2018].

[Sac15] Florian Sachse. Implementation of multiway synchronization for communicating unstructured code. Bachelor's thesis, Technische Universität Berlin, 2015.

[Sch99] Steve Schneider. *Concurrent and Real Time Systems: The CSP Approach*. John Wiley & Sons, Inc., New York, NY, USA, 1999. URL: `http://www.computing.surrey.ac.uk/personal/st/S.Schneider/books/CRTS.pdf`.

[SU05] Ando Saabas and Tarmo Uustalu. A compositional natural semantics and hoare logic for low-level languages. In *Proceedings of the Second Workshop on Structured Operational Semantics*, pages 151–168. Elsevier, 2005. `doi:10.1016/j.entcs.2005.09.031`.

[TDB+13] S. Tucker Taft, Robert A. Duff, Randall Brukardt, Erhard Plödereder, Pascal Leroy, and Edmond Schonberg. *Ada 2012 Reference Manual. Language and Standard Libraries - International Standard ISO/IEC 8652/2012 (E)*, volume 8339 of *Lecture Notes in Computer Science*. Springer, 2013. URL: `https://doi.org/10.1007/978-3-642-45419-6`, `doi:10.1007/978-3-642-45419-6`.

[Tew04] Hendrik Tews. Verifying duff's device: A simple compositional denotational semantics for goto and computed jumps. Technical report, Technische Universität Dresden, 2004. URL: `http://askra.de/papers.html.de`.

[WA17] Editors Andrew Waterman and Krste Asanović. *The RISC-V Instruction Set Manual, Volume I: User-Level ISA, Document Version 2.2*. RISC-V Foundation, May 2017.

[Wat16] Andrew Waterman. *Design of the RISC-V Instruction Set Architecture*. PhD thesis, EECS Department, University of California, Berkeley, Jan 2016. URL: `http://www2.eecs.berkeley.edu/Pubs/TechRpts/2016/EECS-2016-1.html`.

[Win93] Glynn Winskel. *The Formal Semantics of Programming Languages: An Introduction*. MIT Press, Cambridge, MA, USA, 1993.

[Zwi89] J. Zwiers. *Compositionality, Concurrency, and Partial Correctness: Proof Theories for Networks of Processes, and Their Relationship*, volume 321 of *LNCS*. Springer, 1989.