Nils Jähnig, Thomas Göthel, Sabine Glesner

Refinement-Based Verification of Communicating Unstructured Code

The final authenticated publication is available online at https://doi.org/10.1007/978-3-319-41591-8_5

Jähnig, N., Göthel, T., & Glesner, S. (2016). Refinement-Based Verification of Communicating Unstructured Code. In Software Engineering and Formal Methods (LNCS, volume 9763). pp. 61–75. Springer International Publishing. https://doi.org/10.1007/978-3-319-41591-8_5

Refinement-based Verification of Communicating Unstructured Code

Nils Jähnig[1], Thomas Göthel[2], and Sabine Glesner[1]

[1] Technische Universität Berlin
nils.jaehnig@tu-berlin.de
[2] Universität Potsdam

Abstract. Formal model refinement aims at preserving safety and liveness properties of models. However, there is usually a verification gap between model and executed code, especially if concurrent processes are involved. The reason for this is that a manual implementation and further code optimizations can introduce implementation errors. In this paper, we present a framework that allows for formally proving a failures refinement between a CSP specification and its low-level implementation. The implementation is given in a generic unstructured language with `gotos` and an abstract communication instruction. We provide a failures-based denotational semantics of it with an appropriate Hoare calculus. Since failures-based refinement is compositional w.r.t. parallel composition of concurrent components and preserves safety and liveness properties, this contributes to reducing the verification gap between high-level specifications and their low-level implementations.

Keywords: gotos, unstructured code, formal semantics, Hoare calculus, CSP, failures refinement

1 Introduction

Verification is usually performed on abstract models, as usually proofs are more manageable than corresponding proofs on an implementation model. However, when the model is transformed to executable (low-level) code, bugs can be introduced. This is especially the case for manual transformations, which are often necessary as an abstract model is strictly more abstract than the implementation model, and as such is missing implementation details. Furthermore, if done automatically, optimizations and their implementations are usually not verified as this is hard to do at a general level.

Still, the verified properties of the abstract model need to be carried over to the implementation. To preserve *safety and liveness* properties when refining the model, the notion of *stable failures refinement* of *Communicating Sequential Processes* (CSP) is suitable. Additionally, CSP is specifically designed for verification of *communicating* and *non-terminating systems*. It allows a refinement from abstract models to concrete models, but only within CSP, not to relate CSP with other executable code.

To overcome the problem described above, we present a framework that allows for formally proving stable failures refinement between CSP specifications and *Communicating Unstructured Code* (CUC) implementations, which preserves *safety and liveness* properties. CUC is a generic low-level language with gotos and an abstract communication instruction. Our contribution includes a stable failure semantics for CUC and a corresponding Hoare calculus. The stable failures refinement implies that all *liveness and safety* properties of the specification also hold for the implementation.

The rest of this paper is structured as follows. We provide necessary background information about CSP in the next section. In Section 3, we present our framework for relating CSP specifications with CUC implementations. In Section 4, we define the stable failures semantics for CUC and in Section 5 the corresponding Hoare calculus. We illustrate the applicability of our framework in Section 6. In Section 7, we discuss related work. We give a conclusion and pointers to future work in Section 8.

2 Communicating Sequential Processes (Background)

Communicating Sequential Processes (CSP) is a process algebra, originally introduced in [Hoa78]. It is designed specifically to model concurrent processes that communicate via events. Communication is synchronous and can thus be used to synchronize processes or exchange data.

Processes can be constructed from the basic processes STOP and SKIP and using operators such as event prefixing, external and internal choice, interrupts, and sequential and parallel composition. In CSP, the *channels* are introduced as syntactic sugar on events. The event $c.v$ is said to communicate the value v over channel c. To describe an input in CSP, $c?x : T$ is used, which denotes an external choice over all events of the form $c.x$ with $x \in T$. An output is denoted as $c!v$ and means simply $c.v$. Note, that there is no actual native concept of sending and receiving in CSP, only synchronization. Therefore CUC needs only a single communication instruction.

There are two important semantic models for CSP with raising complexity and expressiveness: *1)* The trace semantics $\mathcal{T}$, which describes the communication histories of processes and preserves *safety* properties. *2)* The stable failures semantics $\mathcal{SF}$, which additionally captures the events a process can refuse after a trace, and thereby preserves safety and *liveness* properties. In this paper, we focus on the stable failures semantics.

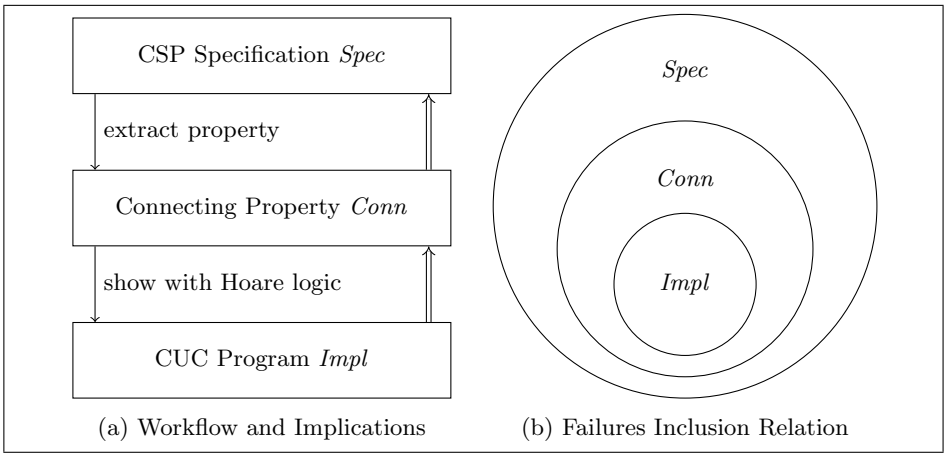
A *failure* is a pair of a trace and a refusal set (tr, X). A process is *stable*, if no internal progress can be made. Thus the process is either waiting to communicate or has stopped (i.e., behaves like $STOP$). We call the former *communication* failures and the latter *terminal* failures. A *stable* failure (tr, X) is a failure where, after engaging in the events in tr, the process is stable and refuses to engage in events from X. When sequentially combining processes P and Q, terminal stable failures of P can become unstable as the combined process might not longer stop after those traces.

The semantic models of CSP allow for modeling various layers of abstraction as described in [Sch99]: Specification, design and implementation, where specification is most abstract, and implementation is closer to an actual implementation. CSP processes can be put into relation across all abstraction levels via *refinements*. Informally, stable failures refinement ($P \sqsubseteq_{SF} Q$) describes the reduction of (internal) non-determinism.

An important property of all the semantical models of CSP is their compositionality. From the refinements $P \sqsubseteq P'$ and $Q \sqsubseteq Q'$ it follows that in any arbitrary composition $\otimes$ also $P \otimes Q \sqsubseteq P' \otimes Q'$ holds, i.e., refinement can be shown component-wise. This enables modular verification in CSP.

The automatic refinement checker FDR3 [GABR14] supports refinement checks for both mentioned semantics.

3 Framework for Formally Relating CSP Specifications and CUC Implementations



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

Fig. 1: Overview of the workflow

In this section, we give an overview of our framework for establishing the relation between a CSP specification and a low-level implementation. We assume that a CSP specification *Spec* is given, as well as an implementation *Impl* thereof in CUC. To preserve liveness and safety properties from *Spec* to *Impl*, we aim at showing that $Spec \sqsubseteq_{\mathcal{SF}} Impl$ holds in the stable failures model. Our proposed workflow is depicted in Figure 1a and consists of three steps:

1) Manually constructing a *connecting property Conn* from *Spec*,

2) showing that *Conn* is sufficient for *Spec*, and

3) showing that *Conn* holds for *Impl*.

The property *Conn* that is constructed from *Spec* in step *1)* is a predicate on CSP failures and needs to be *stronger* than *Spec*. Thus, a connecting property *Conn* has to be sufficiently strong in the sense that it contains the semantics of a concrete CUC program *Impl* while being contained in the semantics of the original CSP specification *Spec*. The inclusion relation is visualized in Figure 1b. It also shows that the weaker *Conn* is, the more implementations can be shown to be a refinement. The ideal property *Conn* is describing exactly the possible failures of *Spec*. This 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 *Conn* are failures of *Spec*.

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 structure. This is not the case for languages with unrestricted jumps.

In step *3)*, it needs to be shown that the property *Conn* holds for the CUC program *Impl* with the Hoare logic presented in Section 5. In such a proof it has to be shown that starting with a precondition Pre, describing the initial states, the program fulfills the postcondition $Post$, which doubles as an invariant for traces due to the way the semantics is defined (this will be explained in Sections 4.2 and 5). In Section 6, we will conduct such a proof for an example consisting of a parallel combination of two simple buffers. This is formally captured by $(tr, s, X) \in [\![Impl]\!] \implies Conn(tr, X)$, i.e., ignoring the state, the failures of *Impl* fulfill *Conn*. Here, (tr, s, X) is a tuple of trace, state, and refusal set, and $[\![Impl]\!]$ denotes the stable failures semantics of *Impl*.

After completing all three steps, we get by transitivity that $(tr, s, X) \in [\![Impl]\!] \implies (tr, X) \in \mathcal{SF}(Spec)$ holds, which is equivalent to our goal $Spec \sqsubseteq_{\mathcal{SF}} Impl$. In the next section we introduce CUC and its stable failure semantics, which we need for step *3)*.

4 Communicating Unstructured Code and its Semantics

We published the language CUC and its operational semantics in [JGG15]. In this section, we give a brief overview of CUC and then proceed to one of the contributions of this paper, the *stable failures semantics*. We discuss its properties and finally define the parallel composition.

4.1 Communicating Unstructured Code

We start with a rationale and continue with some type definitions and a description of the instructions.

We aim at being as close to low-level code as possible to reduce the gap between executed code and verified code. CUC focuses on abstract communication and not its detailed implementation. The detailed implementation of the communication can be verified separately, which is not the focus of this paper. Therefore, we decided to study a generic, unstructured language with a higher

level construct for communication. CUC is generic and simple, which allows for manageable semantics design and proofs without compromising expressiveness.

A *state* s consists of a *program counter* s_{pc} and a *register store* $s_{rs} \in RS$. CUC uses events to model communication. As in CSP, let Σ denote the set of all *events* ev. In the stable failures semantics, we consider *traces* tr that are sequences of events, and *refusal sets* $X \subseteq \Sigma$.

Being a low-level language, instructions are labeled. We choose set of labels to be $\mathbb{N}$. A simple form of a program is a set of labeled instructions. To facilitate compositional reasoning about programs, we use a tree structure ([SU05,JGG15]) in the denotational semantics instead. In either case it is important, that labels are *unique*. The program counter points to the label of the current instruction.

$$\begin{aligned} \mathit{instruction} &::= \texttt{do}\ \ f \mid \texttt{cbr}\ \ b\ m\ n \mid \texttt{comm}\ \ ef\ \ f \\ \mathit{code} &::= (\mathit{label} :: \mathit{instruction}) \mid \mathit{code} \oplus \mathit{code} \end{aligned}$$

Fig. 2: Syntax of CUC Programs

The tree structure of CUC and its instructions are depicted in Figure 2. The three basic instructions are explained below. We consider as (part of) a program a tree of labeled instructions. $\oplus$ connects two program parts. All potential jumps between them are considered. When using a Hoare calculus a suitable tree structure can be used to reason compositionally. For more details, we refer to [SU05].

`do` f – The command `do` is a generalized assignment. f is a function $f\colon RS \to \mathcal{P}(RS)$ and is applied to the current state. The register store of the resulting state is one element of the set returned by `f`. The instruction can thus be thought of as a nondeterministic multiple assignment, i.e., multiple variables can be manipulated in one step. The program counter is increased by one.

`cbr` $b\ m\ n$ – The instruction `cbr` is a usual conditional branch. If the function $b\colon RS \to \{\textsc{True}, \textsc{False}\}$ evaluates to True then the program counter is set to m else to n.

`comm` $ef\ \ f$ – The command `comm` is the communication primitive. It communicates an event from the result of $ef\colon RS \to \mathcal{P}(\Sigma)$ and then changes the state according to $f\colon RS \times \Sigma \to RS$. Observe that here f is deterministic to ease reasoning. The `comm` instruction needs to modify the register store to record input data. We reserve nondeterminism of the successor state to the instruction `do` f. The program counter is increased by one.

4.2 Stable Failures Semantics

In [JGG15] we presented a *trace* semantics for CUC. We enhance this semantics to carry refusal information, i.e., the information on which communications are not possible after performing a particular trace. As a result, we get a stable

failures semantics for CUC that is designed to capture stable failures similarly to CSP.

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. When a process is combined with another, a terminal failure might no longer be terminal and thus become unstable. The unstable failures are removed in the sequential composition (see Section 2).

To be able to differentiate between terminal failures and communication failures in CUC, we introduce two kinds of states: *normal states* and *communication states*. In the former, the next instruction can be executed. In the latter, the execution is in the middle of a `comm` instruction and ready to communicate. Formally, we define a sum type over the state defined in Section 4.1:

$$NCstate \coloneqq normal\ state \mid communication\ state$$

We introduce a predicate $N(\cdot)$ to test if a state is *normal*, and a function $\cdot^C$ which converts a normal state to a communication state. We define a *failure* in CUC to be a triple (tr, s, X), where tr is a trace, s a normal-/communication-state, and X a refusal set. Let $\mathcal{SF}$ be the type of all failures. It will be clear from the context, whether we talk about CSP or CUC failures. We can now express terminal failures in CUC: A failure with normal state whose program counter is not among the labels of the considered program part is a terminal failure. As we remove (former) terminal failures with specific program counters frequently, we introduce an operator, which removes all failures with a normal state and a program counter from a given set.

$$S\backslash_{pcs} \coloneqq S \setminus \left\{(tr, s, X) \;\middle|\; s_{pc} \in pcs \wedge N(s)\right\}$$

Lastly, let $labels(code)$ be the set of all labels in *code* and let $pc \in_\ell code$ denote whether pc points to a label within *code*, i.e., $pc \in labels(code)$.

Our failures semantics is given in Figure 3. We first explain the general structure of the semantics and then the rules individually. The failures semantics is a denotational semantics, which assigns every code tree a function (also called denotation) $[\![code]\!] \colon \mathcal{P}(\mathcal{SF}) \to \mathcal{P}(\mathcal{SF})$. Allowing sets of failures as input eases the sequential composition. We also allow bogus traces as input, thus the semantics is only meaningful if used with sensible initial failures, which usually means triples of: an empty trace, a normal state, and the maximal refusal set or a subset. All failures from the initial input set that are still stable after the execution of the code are carried over to the semantics of the code. This has two reasons. *1)* To be compatible with CSP, the semantics needs to be prefix closed w.r.t. traces. *2)* States that do not point into the code are not processed and remain as they are.

We illustrate this with the first rule D-DO: Consider the initial input set $\{(\langle\rangle, t, X), (\langle\rangle, s, Y)\}$, where $\langle\rangle$ is the empty trace, X, Y are arbitrary refusal sets, $t_{pc} = 5$, $s_{pc} = 1$, both normal, and the instruction $(1{::}\ \texttt{do}\ \lambda\sigma.\ \{\sigma\})$, which does nothing but increment the program counter. As the pc of t is not pointing

D-DO

$$[\![\ell :: \texttt{do}\ \ f]\!](S) \coloneqq S \setminus_\ell \cup \Big\{ (tr,t,X) \;\Big|\; (tr,s,\_) \in S \wedge s_{pc} = \ell \wedge N(s)\ \wedge X \subseteq \Sigma \wedge t_{pc} = \ell + 1 \wedge t_{rs} \in f(s_{rs}) \Big\}$$

D-CBR

$$[\![\ell :: \texttt{cbr}\ \ b\ m\ n]\!](S) \coloneqq S \setminus_\ell \cup \Big\{ (tr,t,X) \;\Big|\; (tr,s,\_) \in S \wedge s_{pc} = \ell \wedge N(s)\ \wedge X \subseteq \Sigma \wedge s_{rs} = t_{rs} \wedge \Big(b\big(s_{rs}\big) \wedge t_{pc} = m\ \vee \neg b\big(s_{rs}\big) \wedge t_{pc} = n\Big) \Big\}$$

D-COMM

$$[\![\ell :: \texttt{comm}\ \ ef\ f]\!](S) \coloneqq S \setminus_\ell \cup \Big\{ \Big(tr, s^C, X\Big) \;\Big|\; (tr,s,\_) \in S \wedge s_{pc} = \ell \wedge N(s) \wedge X \subseteq \Sigma \setminus ef(s_{rs}) \Big\} \cup \Big\{ (tr^\frown ev, t, X) \;\Big|\; (tr,s,\_) \in S \wedge s_{pc} = \ell \wedge N(s) \wedge X \subseteq \Sigma \wedge t_{pc} = \ell + 1 \wedge ev \in ef\big(s_{rs}\big) \wedge t_{rs} = f\big(s_{rs}, ev\big) \Big\}$$

D-SEQ

$$[\![code_1 \oplus code_2]\!](S) \coloneqq \Big(\big(\mu d.\ extend(code_1, code_2)\ (d)\big)(S)\Big)\setminus_{labels(code_1 \oplus code_2)}$$

D-EXT

$$extend(code_1, code_2)\ (d) \coloneqq \lambda S.\ S \cup d\big([\![code_1]\!](S)\big) \cup d\big([\![code_2]\!](S)\big)$$

Fig. 3: Stable Failures Semantics for CUC

to this instruction, the failure is still terminal. There is no successor failure of $(\langle\rangle, t, X)$. Within the state s, the program counter s_{pc} points to the instruction, so there is "a" successor failure $\{(\langle\rangle, s', Z) \mid s' = s[pc \leftarrow 2] \wedge Z \subseteq \Sigma\}$. The initial failure $(\langle\rangle, s, X)$ is not terminal anymore, thus no longer stable and needs to be removed. Thus the resulting failures are

$$[\![1 :: \texttt{do}\ \lambda\sigma.\ \{\sigma\}]\!](\{(\langle\rangle, t, X), (\langle\rangle, s, Y)\}) = \{(\langle\rangle, t, \_)\} \cup \{(\langle\rangle, s', Z) \mid s' = s[pc \leftarrow 2] \wedge Z \subseteq \Sigma\}$$

The rule D-CBR works in similar way, but alters the subsequent program counter instead of the register store. The rule D-COMM adds two kinds of failures: The terminal failures after the execution of the instruction, in the same way as the two previous rules. Furthermore, it adds the communication failures, when it is ready to communicate.

The rule D-SEQ is the most complex, and it is based on D-EXT. The latter takes a denotation d and extends it with the execution of $code_1$ and $code_2$,

separately. More specifically, the input set S is first evaluated with the denotations for $code_1$ and $code_2$ and then passed to d, which corresponds to executing $code_1$ or $code_2$ first, and then executing d. In D-SEQ, we "loop" this construct now indefinitely, and obtain as a result all possible interleavings of $code_1$ and $code_2$. To this end, we use the *least fixpoint* over the *complete partial order* of functions $\mathcal{P}(\mathcal{SF}) \to \mathcal{P}(\mathcal{SF})$, with the pointwise subset inclusion a ordering $\big(f \leq g \coloneqq \forall S.\ f(S) \subseteq g(S)\big)$.[3] As in the other rules, we need to remove all former terminal failures.

We illustrate the rule D-SEQ with an example. Consider the initial failure $(\langle\rangle, s, X)$ with $s_{pc} = 1$, X arbitrary, and the program

$$(1{::}\ \texttt{comm}\ \lambda\sigma.\ \{a\}\ \lambda\sigma\ event.\ \{\sigma\}) \oplus (2{::}\ \texttt{cbr}\ \lambda\sigma.\ True\ 1\ 1)$$

It is a non-terminating program communicating a repeatedly with its environment. According to D-SEQ, both instructions are evaluated separately, where initially `comm` modifies the set accordingly (e.g., append a to the trace) and `cbr` does nothing, as s_{pc} does not point to it. In the next iteration of the fixpoint iteration, both instructions are again executed. This time `comm` does nothing (new) but `cbr` will now generate failures whose program counter points to `comm`, so in the next iteration the loop will be executed from the beginning. As a global fixpoint we get the failure set

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

As this program does not terminate, there are no normal states in its semantics.

4.3 Compatibility to CSP

In this section, we show that our CUC semantics enjoys basic properties of the CSP semantics. This allows us to show that CUC is compatible with CSP, which finally allows us to prove failures refinement between a CUC implementation and its CSP specification. As the refinement relation is basically just a subset relation, its use is clear for safety properties, but for liveness properties the considered failure sets need to fulfill some properties (simply speaking they may not be too small). We introduce and explain adapted versions of the properties of the CSP failures semantics (see e.g., in [Sch99]) and briefly discuss why they hold. We omit the program and the initial failures set for brevity. For each of the following properties, we require that it holds for the initial set of failures. Let $\mathcal{SF}$ be the stable failures of the omitted program, and $\mathcal{T}$ the traces according to the trace semantics given in [JGG15].

SF1: $(tr, s, X) \in \mathcal{SF} \implies (tr, s) \in \mathcal{T}$ – All trace-state pairs are included in the trace semantics. This property ensures that we still have all benefits of the traces semantics (safety properties). The trace semantics for CUC and its properties are published in [JGG15]. It holds as we extended the trace semantics in a safe way.

[3] For an introduction to denotational semantics and fixpoints see, e.g., [Rey98].

SF2: $(tr, s, X) \in \mathcal{SF} \wedge X' \subseteq X \implies (tr, s, X') \in \mathcal{SF}$ – Refusal sets are subset closed. This holds by construction.

SF3: $(tr, s, X) \in \mathcal{SF} \wedge \forall\, a \in X', t.\ (tr^\frown a, t) \notin \mathcal{T} \implies (tr, s, X \cup X') \in \mathcal{SF}$ – The refusal set can be augmented with events not possible. This is the important property ensuring that there are "enough" refusals to show liveness properties. This also holds by construction.

SF4: $(tr, s) \in \mathcal{T} \wedge s_{pc} \notin_\ell \mathit{code} \implies (tr, s, X) \in \mathcal{SF}$ – Terminal failures are stable. This also holds by construction.

Properties SF3 and SF4 ensure that all stable failures are included, and thus guarantee that the stable failures refinement relation allows to carry over (safety and) liveness properties.

4.4 Concurrent Semantics

Having defined the sequential semantics in Section 4.2, we now define the concurrent semantics. It is defined as close a possible to the concurrent CSP semantics. The purpose is to inherit the 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.

To define the concurrent semantics of CUC, we first define the notion of a concurrent state. As components communicate via events, the states of components do not share variables. We define a concurrent state to be a normal-/communication-state or pair of concurrent states:

$$\mathit{concurrent\ state} \coloneqq \mathit{NCstate} \mid \mathit{concurrent\ state} \parallel \mathit{concurrent\ state}$$

The nesting structure of a concurrent state should match the nesting structure of a parallel program. 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. As in CSP, the concurrent composition of two components considers all interleavings of the traces of the components, synchronizing on the given alphabets. As we only allow top-level composition, we assume initial failures to have an empty trace and a normal state.

$$\begin{aligned}
[\![\mathit{code}_1\ {}_\alpha\|_\beta\ \mathit{code}_2]\!](S) = \big\{ (tr, t_1 \parallel t_2, X) \mid\ & \exists\, X_1, X_2. \\
& (\langle\rangle, s_1 \parallel s_2, Y) \in S \wedge N(s_1) \wedge N(s_2) \wedge \\
& X \cap (\alpha \cup \beta) = (X_1 \cap \alpha) \cup (X_2 \cap \beta) \wedge \\
& (tr \upharpoonright\ \alpha, t_1, X_1) \in [\![\mathit{code}_1]\!]\big(\{(\langle\rangle, s_1, Y)\}\big) \wedge \\
& (tr \upharpoonright\ \beta, t_2, X_2) \in [\![\mathit{code}_2]\!]\big(\{(\langle\rangle, s_2, Y)\}\big) \wedge \\
& \mathit{set}(tr) \subseteq (\alpha \cup \beta)\big\}
\end{aligned}$$

We have created a semantics which fulfills our needs, and in particular, preserves the properties from the previous section. We are now able to combine CUC programs in parallel. As we have defined parallel composition within CUC like in CSP, we enjoy its compositionality. We thus need only to refine single components and can combine the results, thanks to the compositionality of the stable failures refinement w.r.t. parallel composition. This is the reason, why we do not need a rule for parallel composition in our Hoare calculus, which we introduce in the following section.

5 Hoare Calculus

Our assertions are predicates on single $NCstate$s. We define a Hoare triple as usual with one catch:

$$\{P\}\ code\ \{Q\} \coloneqq \forall\, s.\ \Big(P(s) \longrightarrow \forall\, t \in [\![code]\!](\{s\}).\ Q(t)\Big)$$

Observe that our semantics yields a set that includes *all intermediate* stable failures. Postconditions in our Hoare calculus are thus also *invariants for communication failures*. We still can construct usual postconditions, e.g., by setting $Q(t) \coloneqq t_{pc} \notin_\ell code \longrightarrow Q'(t)$.

We present our Hoare calculus for stable failures of CUC in Figure 4. In H-DO, H-CBR and H-COMM, it is described how pre- and postconditions can be connected for the basic instructions. As all sequential compositions potentially introduce loops in CUC, H-SEQ is based on an invariant I, which is tailored to its placement in the rule: For $code_i$, only the relevant parts of the invariant have to hold as its precondition. In the postcondition of the combination, all parts of the invariant that deal with now former terminal failures are ignored by the conjunction with the requirement that all normal states do not point into the code. H-CONS is the usual rule of consequence.

All rules of the calculus are correct w.r.t. the definition at the start of this section. This can be shown by structural induction over the structure of an arbitrary CUC program. This corresponds to a partial correctness for normal states where the program terminates (we do not show termination). However, for communication failures the postcondition holds universally thus can 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, nonterminating systems. Our Hoare calculus is not complete

In this paper, we assume that sequential system components are refined only separately. Due to the compositionality of failures refinement w.r.t. parallel composition, we do not need an additional rule for concurrent components. In summary, our overall framework allows for proving properties about sequential components and their parallel combination in a compositional way. We demonstrate its applicability with an example in the next section.

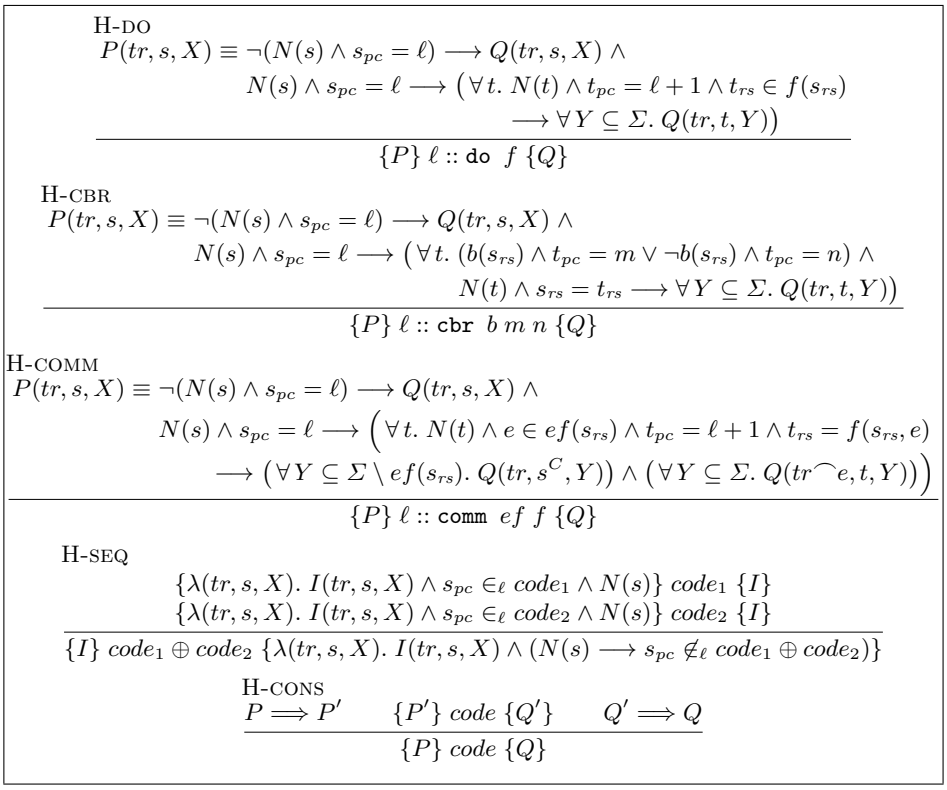


Fig. 4: Hoare Calculus for CUC

6 Example

We demonstrate the applicability of our formal framework as presented above and show that a given CSP specification *Spec* for a one place buffer is refined by a given CUC implementation *Impl* of a buffer. Both are shown in Figure 5. The elements that can be stored in the buffer are of type T. *Spec* waits for an input on channel *in*, i.e., synchronizes on any event $\{in.x \mid x \in T\}$, outputs the received value x on channel *out*, and then starts over. We define $\oplus$ to be right associative. Next, we explain *Impl* line by line:
`(1::do)` – This is the initialization. The boolean *free* indicates that the *buffer* is ready to store data.
`(2::comm)` – The `comm`-instruction both offers the events and changes the state after the communication happened. The events offered by *ef* are all values of type T on channel *in* if the buffer is free, else 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 just sets the buffer to free.

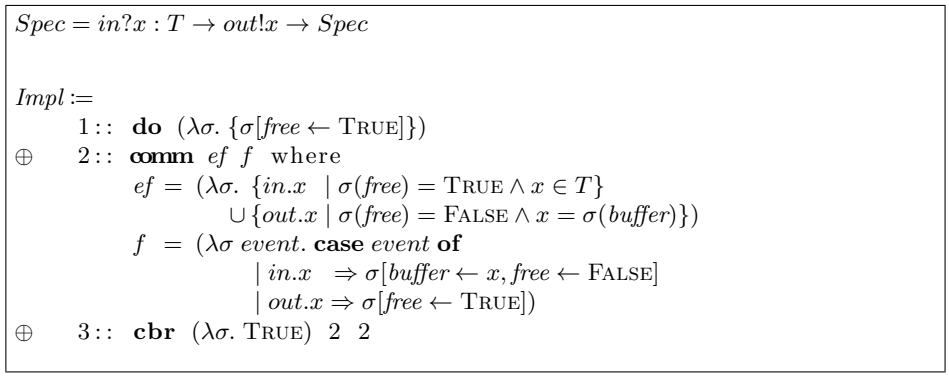


Fig. 5: CSP Specification and CUC Implementation of a One Place Buffer

`(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 *Conn* from *Spec*

First, we need to extract a connecting property *Conn* which is only true for the failures of *Spec*. Let $trace^*$ mean *trace* zero or more times concatenated, where the variable $x \in T$ is fresh in every occurrence of *trace*. We define

$$Conn(F) \coloneqq F \in \mathbb{F}_{even} \vee F \in \mathbb{F}_{odd} \qquad \text{where}$$
$$\mathbb{F}_{even} \coloneqq \left\{ \left((in.x^\frown out.x)^*, X \right) \;\middle|\; X \subseteq \Sigma \setminus \{in.y \mid y \in T\} \right\}$$
$$\mathbb{F}_{odd} \coloneqq \left\{ \left((in.x^\frown out.x)^* {}^\frown in.y, X \right) \;\middle|\; y \in T \wedge X \subseteq \Sigma \setminus \{out.y\} \right\}$$

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: Relation between *Conn* and *Spec*

We need to prove that this holds *only* for stable failures of *Spec*, i.e., $Conn(F) \Longrightarrow F \in \mathcal{SF}(Spec)$, but in this simple case it is easy to see, as *Conn* describes exactly the failures of *Spec*.

Lemma 1. $Conn(F) \Longrightarrow F \in \mathcal{SF}(Spec)$

Step 3: Relation between *Conn* and *Impl*

In the next step, we need to show that *Conn* holds for all failures of the program *Impl*, or more exactly for all elements of the projection of the failures of *Impl* onto the traces-refusal pairs. To this end, we need an invariant *Inv*, which

implies *Conn* but is effectively stronger, as we need state information such as the current program counter. We also need to specify the initial failures with a precondition *Pre*. We omit *Pre* in the formulation of several lemmas for brevity. In the following, we show $\{Pre\}\ Impl\ \{Inv\}$. First, we define *Pre* and *Inv*:

$$\begin{aligned} Pre(tr, s, X) &:= s_{pc} = 1 \wedge tr = \langle\rangle \\ Inv &:= Pre \vee I_{2,3} \\ I_{2,3}(tr, s, X) &:= s_{pc} \in \{2, 3\} \wedge \\ &\quad ((tr, X) \in \mathbb{F}_{even} \wedge s_{rs}(free) = \textsc{True} \vee \\ &\quad (tr, X) \in \mathbb{F}_{odd} \wedge s_{rs}(free) = \textsc{False} \\ &\quad \wedge \exists\, x.\ s_{rs}(buffer) = x \wedge last(tr) = in.x) \end{aligned}$$

Lemma 2. $(tr, s, X) \in [\![Impl]\!] \implies Conn(tr, X)$

Proof. We show $(tr, s, X) \in [\![Impl]\!](\{(tr, t, Y) \mid Pre(tr, t, Y)\}) \implies Inv(tr, s, X)$ with our Hoare calculus, i.e., $\{Pre\}\ Impl\ \{Inv\}$ holds. 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 $Inv(tr, s, X)$(eigentlich noch $\wedge \neg Pre$) $\implies Conn(tr, X)$ holds too, we conclude

$$(tr, s, X) \in [\![Impl]\!](\{(tr, t, Y) \mid Pre(tr, t, Y)\}) \implies Conn(tr, X)$$

□

From Lemma 1 and Lemma 2 we conclude Theorem 1 that all trace-refusal pairs of *Impl* are failures of *Spec*, i.e., $Spec \sqsubseteq_{\mathcal{SF}} Impl$ holds. $\sqsubseteq_{\mathcal{SF}}$ being CSP (stable) failures refinement, *Impl* thus enjoys all liveness and safety properties of *Spec*.

Theorem 1. $Spec \sqsubseteq_{\mathcal{SF}} Impl$

6.1 Concurrency

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

$$\begin{aligned} Spec_1 &= in?x : T \to mid!x \to Spec_1 \\ Spec_2 &= mid?x : T \to out!x \to Spec_2 \\ Spec_{\|} &= Spec_1 \ {}_{\{mid\}}\|_{\{mid\}}\ Spec_2 \end{aligned}$$

We can show that two programs $Impl_1$ and $Impl_2$ (similar to the code in Figure 5) refine $Spec_1$ and $Spec_2$ respectively (see Theorem 1). Let

$$Impl_{\|} = Impl_1 \ {}_{\{mid\}}\|_{\{mid\}}\ Impl_2$$

Due to the compositionality of CSP and the equal nature of the parallel compositions of CSP and CUC, we can immediately follow that $Spec_{\|} \sqsubseteq_{\mathcal{SF}} Impl_{\|}$, which demonstrates the compositionality of our approach. Please 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 the in the buffer example) we even can reuse the refinement proof.

7 Related Work

To the best of our knowledge there exists no other approach to define a stable failures semantics for low-level code.

A denotational semantics and a proof calculus for a high-level language with communication are defined by Zwiers [Zwi89]. The semantics deals with traces and *ready sets*, which are similar in intention to refusals. As low-level code is not considered, the semantics is not directly applicable.

There are some attempts to give unstructured code a semantics for later verification. Tews [Tew04] developed a compositional semantics for a C-like language with `goto`, which is used to verify Duff's device. However, this approach does not model communication, and is thus not appropriate to describe non-terminating systems. Saabas and Uustalu [SU05] present a compositional bigstep semantics of an unstructured language. To this end, a generic structuring mechanism for the code is presented, which makes the semantics compositional and also allows for a compositional proof calculus. Although they formally relate a high-level and a low-level language, they do not relate a process specification with the low-level language. Communication is not considered. CUC uses their structuring mechanism [JGG15].

CUC, presented in [JGG15], is based on our previous work [BJ14] enhanced with communication capabilities. The approach in [BJ14] focuses on a small-step and a bigstep operational semantics based on which a compositional proof calculus is built. We used similar semantics in [BG11] to show correspondence between unstructured code and (Timed) CSP processes. We used events as observation points, but did not consider actual communication. Furthermore, the use of a bisimulation to relate unstructured code and CSP processes allows only for equivalence, which is inappropriate for an implementation process.

8 Conclusion

In this paper, we have defined a stable failures semantics and a Hoare calculus for CUC. Both are used in our framework, which allows for formally proving stable failures refinement between specifications in CSP and implementations in CUC. Our framework thus contributes to reducing the verification gap between behavioral abstract specification and executed low-level code. This relation preserves all safety and liveness properties of the specification. Our approach is compositional w.r.t. parallel system components, i.e., we only need to show refinements

for the sequential components of the system, as the properties are preserved for the entire system due to compositionality of stable failures refinement.

In future work, we aim at extending our existing Isabelle/HOL [NPW02] formalization of the trace semantics for CUC and the corresponding Hoare calculus for stable failures. To enable a further refinement of the CUC implementation, we plan to model the detailed implementation of the `comm` instruction in a low level language with primitives to implement a channel, such as shared variables and locks. We aim at investigating the applicability of our approach with more complex systems. We are especially interested in the utility of the intra-component compositionality of CUC. Finally, our framework could be combined with other frameworks, e.g., the CSP++ framework [GGC15], where a C++ communication backbone is generated from a CSP specification.

References

[BG11] B. Bartels and S. Glesner. Verification of distributed embedded real-time systems and their low-level implementation using timed csp. In *APSEC 2011*, pages 195–202. IEEE Computer Society, 2011.

[BJ14] Björn Bartels and Nils Jähnig. Mechanized, compositional verification of low-level code. In *NASA Formal Methods*, volume 8430 of *LNCS*, pages 98–112. Springer International Publishing, 2014.

[GABR14] Thomas Gibson-Robinson, Philip J. Armstrong, Alexandre Boulgakov, and A. W. Roscoe. FDR3 - A modern refinement checker for CSP. In *TACAS 2014*, pages 187–201, 2014.

[GGC15] William B. Gardner, Alicia Gumtie, and John D. Carter. Supporting selective formalism in CSP++ with process-specific storage. In *ICESS 2015*, pages 1057–1065, 2015.

[Hoa78] C. A. R. Hoare. Communicating sequential processes. *Commun. ACM*, 21(8):666–677, 1978.

[JGG15] Nils Jähnig, Thomas Göthel, and Sabine Glesner. A denotational semantics for communicating unstructured code. In *FESCA* 2015, volume 178 of *EPTCS*, pages 9–21, 2015.

[NPW02] Tobias Nipkow, Lawrence C. Paulson, and Markus Wenzel. *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*, volume 2283 of *LNCS*. Springer, 2002.

[Rey98] John C. Reynolds. *Theories of programming languages*. Cambridge University Press, 1998.

[Sch99] Steve Schneider. *Concurrent and Real Time Systems: The CSP Approach*. John Wiley & Sons, Inc., New York, NY, USA, 1999.

[SU05] Ando Saabas and Tarmo Uustalu. A compositional natural semantics and hoare logic for low-level languages. In *SOS*, pages 151–168. Elsevier, 2005.

[Tew04] Hendrik Tews. Verifying duff's device: A simple compositional denotational semantics for goto and computed jumps. Technical report, Technische Universität Dresden, 2004.

[Zwi89] J. Zwiers. *Compositionality, Concurrency, and Partial Correctness: Proof Theories for Networks of Processes, and Their Relationship*, volume 321 of *LNCS*. Springer, 1989.