A Denotational Semantics for Communicating Unstructured Code

Nils Jähnig Thomas Göthel Sabine Glesner
Technische Universität Berlin, Germany
{nils.jaehnig, thomas.goethel, sabine.glesner}@tu-berlin.de

An important property of programming language semantics is that they should be compositional. However, unstructured low-level code contains goto-like commands making it hard to define a semantics that is compositional. In this paper, we follow the ideas of Saabas and Uustalu [7] to structure low-level code. This gives us the possibility to define a compositional denotational semantics based on least fixed points to allow for the use of inductive verification methods. We capture the semantics of communication using finite traces similar to the denotations of CSP. In addition, we examine properties of this semantics and give an example that demonstrates reasoning about communication and jumps. With this semantics, we lay the foundations for a proof calculus that captures both, the semantics of unstructured low-level code and communication.

1 Introduction

Electronic devices are increasingly integrated into our daily lives. Most of them are "invisible" and would, ideally, run forever once deployed. Often this is not a crucial feature, as a smart phone or a server can be restarted if they crash. But in cases where the system cannot be restarted, the correctness of those systems needs to be ensured. Examples of such cases are medical devices and automotive systems.

By the *correctness of a system* we mean a *formal conformance relation* between an abstract specification and its implementation. We consider the specification language of Communicating Sequential Processes (CSP) [8] as suitable technique to describe and verify the communication behavior of concurrent programs. The advantage is that properties can be verified on the abstract model of CSP and conformance ensures that these properties are preserved to the implementation level. We assume as in our modeling and verification approach presented in [4] that, having refined the CSP specification sufficiently, it is implemented in a high-level language such as C++ and then compiled to a low-level language such as LLVM. Instead of showing conformance between a high-level language and CSP, we show conformance of the corresponding low-level representation instead. This approach has the advantage that a complex formal semantics of a high-level language can be avoided and tightens the gap between verified code and executed code. We choose an LLVM-like unstructured language that we previously presented in [2] because it is platform independent. In order to model communicating programs, we have included a generic instruction that allows for communication between unstructured programs.

In this paper, we give a formal denotational semantics to the implementation level focussing on communicating unstructured code. The particular challenge lies in the fact that denotational semantics for unstructured code are usually not compositional because of the presence of some variant of the goto command. This usually requires the whole code available when showing properties. Furthermore, we consider communication. To overcome the first problem, we structure unstructured code following ideas from Saabas and Uustalu [7]. While they define a compositional bigstep semantics for an unstructured language, we define a denotational semantics based on fixpoints, which additionally copes with communication and nontermination. The latter features are close to the approach taken in CSP. As we require our approach to enable automation and also to gain confidence in the correctness of our semantical construction, we formalize all results in the theorem prover Isabelle/HOL[1] [6]. Our denotational semantics gives us the groundwork for developing a compositional proof calculus that not only allows to formulate properties about possible states of unstructured programs but also allows us to reason about the communication behavior using traces.

The rest of this paper is structured as follows. Related work is dicussed in the next section. In Section 3, we give necessary background information about structured code, the concept of denotational semantics, CSP, and Isabelle/HOL. In Section 4, we introduce our low-level language, which we call *Communicating Unstructured Code* and for which we define a smallstep operational semantics in Section 4.1. Our main contribution is described in Section 4.2 where we introduce our denotational semantics for Communicating Unstructured Code. In Section 5, we analyze the denotational semantics and prove conformance to the intuitive operational semantics. In Section 6, we define a rule to ease argumentation about the semantics. This gives an idea what a future calculus will look like. We illustrate the utility of our denotational semantics in Section 7. We give a conclusion and pointers to future work in Section 8.

2 Related Work

There are some attempts to give unstructured code a semantics for later verification. Tews [9] 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.

Saabas and Uustalu [7] 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.

A denotational semantics and a proof calculus for a high-level language with communication were already defined by Zwiers [11]. As low-level code is not considered, the semantics is not directly applicable, but we will follow this approach when designing a proof calculus in future work. A preliminary rule of this calculus is presented in this paper.

Our unstructured language that we introduce in Section 4 is based on our previous work [2] enhanced with communication capabilities. The approach in [2] focuses on a smallstep and a bigstep operational semantics based on which a compositional proof calculus is built. We used similar semantics in [1] to show correspondence between unstructured code and (Timed) CSP processes. In [1] we used events as observation points, but did not consider actual communication. In addition to considering also communication, we aim in future at relating unstructured code and CSP processes based on our denotational semantics, which facilitates refinement-based verification instead of bisimulation equivalence as used in [1].

3 Background

In this section, we introduce the concept of structuring unstructered code and give brief introductions to denotational semantics and Communication Sequential Processes.

[1]When the intended Hoare calculus (see Section 6) is finished, we plan to publish the Isabelle theories in the *Archive of Formal Proofs* (`http://afp.sourceforge.net/`). If you are interested in the current version of the theories, please contact us.

3.1 Structured Code

It is difficult to obtain a compositional semantics for unstructured low-level code, as compositionality usually involves splitting the code into parts according to its structure. The simplest structure is a list of instructions, which can be split into head and tail. But also in unstructured code there are blocks of instructions that belong to the same functionality, and it is desirable to reason about such blocks as components and not instruction after instruction appended to the previous code (as it would be the case for lists). Therefore we choose a structure which allows for those blocks to be reasoned about as a component, namely trees. To this end, we follow a similar approach to the one presented in [7] by Saabas and Uustalu.

In their approach, every instruction has a unique label (e.g., a natural number) that refers to it, and the semantics depends on an explicit *program counter* that holds the label of the next instruction to execute. After executing one instruction, the program counter is usually incremented by one so that it points to the next instruction. Only *control flow instructions*, e.g., `goto` can set the program counter to a different value. This way, conditionals or loops can be constructed.

An *unstructured program* can be intuitively formalized as a set of *labeled instructions*, i.e., pairs of labels and instructions. All labels are assumed to be unique. The shortcoming of this formalization is that there is no useful structure on which induction can be applied. E.g., as the whole code needs to be considered all the time due to the `goto` instructions, the operational smallstep semantics on it (see Section 4.1) is not compositional[2] with respect to the program. So a denotational semantics on unstructured code will also not be compositional.

To overcome this problem, we introduce a tree structure on the instructions. Let *code* adhere to the following grammar

$$code ::= (label :: instruction) \mid code \oplus code$$

where $\oplus$ is the sequential composition operator. As described in the next subsection, we can use this structure to reason about the single components (i.e., subtrees), and compose their meaning, thus obtain compositionality. To make sure that the structure does not influence the semantics of a program, all possibilities to structure a given set of instructions must have the same meaning (for our semantics this is a corollary of the conformance proved in Section 5.1). We can choose which structure on the code fits our needs best. Usually, this is a structure where as many jumps as possible are inside a component. By this, the entry and exit possibilities for each component are minimized. This will help in future work when choosing pre and post conditions, as for known code, not every instruction label needs to be considered as a possible entry point.

It is important to note that even though we call $\oplus$ the *sequential composition* (opposed to a possible *parallel composition*), it is not the sequential composition known from structured programming languages. The important difference is that instructions combined with $\oplus$ can create loops, if branch instructions are contained. As loops are only dealt with in rules about $\oplus$, we call $\oplus$ the looping construct of the language.

To be able to relate structured and unstructured code, a projection $\cup_{sc}$ from *structured code* to *labeled instruction sets* is defined, which "forgets" the structure. Observe, that the operation $\oplus$ is only defined for structured code.

Having introduced the concept of structured code in our setting, we are able to define a compositional denotational semantics of our low-level language. It is particularly amenable to fixpoint induction, which we describe subsequently.

[2]Compositionality here refers to "sequential" compsitionality. In this paper we do not consider "parallel" compositionality.

3.2 Denotational Semantics

In contrast to operational semantics, which describes 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. As denotational semantics can be characterized via operational semantics and operational semantics can be defined compositionally, boundaries between the two can be vague.

To be able to assign a denotation to programs containing loops, a function F 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 F is then the denotation of the program containing the loop.

To define a least fixpoint, we use a chain-complete partial order on the denotations. This is usually obtained by lifting the chain-complete partial order on the states, i.e., 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 [10]) requires the function F to be continuous, and the property of interest I to be admissible, i.e., if I holds for all elements of a chain, then I also holds for the supremum of the chain. Finally, for every denotation f, if $I(f)$ holds, then $I(F(f))$ has to hold as well (step case).

3.3 Communicating Sequential Processes

Note: CSP is not necessary for the following definitions, but is helpful as the denotational semantics is designed with CSP in mind.

Communicating Sequential Processes (CSP) [8] is a process algebra, originally introduced in [5]. It is designed specifically to model concurrent processes, which communicate via events. Communication is synchronous and thus can 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.

CSP is suited for modeling various layers of abstraction as described in [8]: Specification, design and implementation, where specification is most abstract, and implementation can be directly converted into program code. Abstraction levels can be mixed in a process, and CSP processes across all abstraction levels can be put in relation via *refinements*. Informally a process Q refines a process P, if the behavior of process Q is a subset of the behavior of process P.

There are two important semantic models for CSP with raising complexity and expressiveness: *1)* The trace semantics, which describes the communication histories of processes, which we use in this paper. *2)* The stable failures semantics, which additionally captures the events a process can refuse after a trace. In this paper, we focus on traces, but we plan to extend it to failures in future work.

The automatic refinement checker FDR3 [3] supports refinement checks for both mentioned semantics.

4 Communicating Unstructured Code

In this section, we introduce *Communicating Unstructured Code* (CUC), which we adapt from our previous work [2] and enhance it with a communication primitive.

On the one hand, we want to be as close to low-level code as possible to reduce the gap between executed code and verified code. On the other hand, we want to focus on communication and not its implementation. Therefore, we decided to use an unstructured language with a higher level construct for communication.

We keep our language generic and simple. Though being simple, it is powerful (as it contains conditional branches). This simplicity allows for manageable semantics design and proofs. Our language consists of three basic instructions:

$$\mathit{instruction} := \texttt{do f} \mid \texttt{cbr b m n} \mid \texttt{comm ef f}$$

In the following, we give an informal explanation of these instructions.

`do f` – The command `do` is a generalized assignment. `f` is a function from *state* to *set of states* and it is applied to the current state. The resulting state is one element of the set returned by `f`. The instruction can thus be thought of as a *nondeterministic multiple assignment*.

`cbr b m n` – The instruction `cbr` is a usual conditional branch. If the function `b` from *state* to *bool* 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` and then changes the state according to `f`, where `ef` is a function from *state* to *set of events* and `f` is a function from *state* and *event* to *state*. Observe that here `f` is deterministic to ease reasoning. We reserve nondeterminism of the successor state to the instruction `do f`.

The instructions `do f` and `comm ef f` are abstract instructions (instructions schemes), which can be instantiated. For example, the instance on the left models an assignment of y to x. The instance on the right adds x to y and stores the result to z.

$$\begin{array}{cc} \text{“}x := y\text{”} & \text{“}z := x + y\text{”} \\ \texttt{do}\ (\lambda\sigma.\ \sigma[x \leftarrow \sigma(y)]) & \texttt{do}\ (\lambda\sigma.\ \sigma[z \leftarrow \sigma(x) + \sigma(y)]) \end{array}$$

The next instance defines an input of values of type T over channel *in* storing the value in variable x.

$$\begin{array}{c} \text{“}x := \mathrm{input}(in :: T)\text{”} \\ \texttt{comm}\ (\lambda\sigma.\ \{in.v \mid v \in T\})(\lambda\sigma\ ev.\ \sigma[x \leftarrow val(ev)]) \end{array}$$

In the last example and in Section 7, events are of the type $channels \times values$ (written in the dot-notation $channel.value$ as usual in CSP). This is just a convenient instantiation, as there are no strucutral requirements on the type of events (also as in CSP). Apart from the chosen type of events, there is no concept of channels in CUC.

In this section, we have presented the instructions of our language and have given its informal semantics. In the next section, we first define the more intuitive smallstep semantics to capture our ideas, and then define the denotational semantics in Section 4.2, as it fits better our ultimate goal to relate CUC programs and CSP processes. In Section 5.1, we show conformance between the smallstep and the denotational semantics, to ensure that the denotational semantics adheres to our ideas.

4.1 Operational Smallstep Semantics

As we consider unstructured code, a program is given as a set of *labeled instructions*, i.e., pairs of identifier (label) and instructions. We assume that the identifiers are unique.

We model the state as consisting of three parts: The usual state (i.e., the values of variables), the program counter, and the trace (i.e., the communication history as a list of events). Observe that each instruction roughly corresponds to one part of the state: `do` changes the variables, `cbr` the program counter, and `comm` the trace. The exceptions are that both, `do` and `comm`, increase the program counter by one, and that `comm` can also change the state to store communicated values. See Figure 1 for the smallstep semantics, which captures the behavior described in Section 4. Each instruction can only be applied if the *pc* is pointing to its label. We explain the rules in detail:

S-DO – Only the variables are changed, possibly nondeterministically, according to f. Traces stay the same and the program counter is increased by one.

S-CBR – There are two rules, one for the case when the condition b evaluates to True, and one when it evaluates to False. In both cases only the *pc* is changed accordingly, and variables and trace remain the same.

S-COMM – ef determines which events are offered for communication. The communicated event is appended to the existing trace. The successor state is calculated according to f, this time deterministically and additionally depending on the communicated event. f is deterministic to focus on communication. Again, the program counter is increased by one.

$$\sigma \xrightarrow{code} \vartheta := (\sigma, \vartheta) \in smallstep(code)$$

$$\frac{(\ell :: \texttt{do}\ \ f) \in code \qquad t \in f(s)}{(tr, s, \ell) \xrightarrow{code} (tr, t, \ell + 1)}\ \textsc{S-do}$$

$$\frac{(\ell :: \texttt{cbr}\ \ b\, m\, n) \in code \qquad b(s) = \text{True}}{(tr, s, \ell) \xrightarrow{code} (tr, s, m)}\ \textsc{S-cbr True}$$

$$\frac{(\ell :: \texttt{cbr}\ \ b\, m\, n) \in code \qquad b(s) = \text{False}}{(tr, s, \ell) \xrightarrow{code} (tr, s, n)}\ \textsc{S-cbr False}$$

$$\frac{(\ell :: \texttt{comm}\ \ ef\ f) \in code \qquad ev \in ef(s) \qquad tr_t = tr_s{}^\frown ev \qquad t = f(s, ev)}{(tr_s, s, \ell) \xrightarrow{code} (tr_t, t, \ell + 1)}\ \textsc{S-comm}$$

Figure 1: Smallstep Semantics for CUC

The smallstep semantics defines the executions of single instructions. To relate the operational semantics with the denotational semantics we need the *multistep* relation, which is the reflexive transitive closure of the smallstep relation, and defines the semantics for executions of a set of instructions. See the multistep semantics in Figure 2. The first rule (M-BASE) is the base case, stating that every step is reachable from itself (reflexivity); the second rule (M-STEP) is the step rule, stating that if a state is reachable, its smallstep successors are reachable, too (transitivity). Please note that the operational semantics is not defined for $\oplus$, as the operational semantics is defined on unstructured code.

4.2 Denotational Semantics

We presented previously a compositional bigstep semantics for a low-level language *without* communication in [2]. In contrast, we require our extended semantics *with* communication to be close to the denotational CSP semantics. To this end, we define a denotational semantics which captures the state information but also the finite traces of program executions.

$$\sigma \xrightarrow{code}{}^{*} \vartheta := (\sigma, \vartheta) \in multistep(code)$$

$$\frac{}{(tr, s, \ell) \xrightarrow{code}{}^{*} (tr, s, \ell)} \textsc{M-base}$$

$$\frac{(tr_s, s, \ell_s) \xrightarrow{code} (tr_u, u, \ell_u) \qquad (tr_u, u, \ell_u) \xrightarrow{code}{}^{*} (tr_t, t, \ell_t)}{(tr_s, s, \ell_s) \xrightarrow{code}{}^{*} (tr_t, t, \ell_t)} \textsc{M-step}$$

Figure 2: Multistep Semantics for CUC

D-DO

$$[\![\ell :: \texttt{do}\ \ f]\!](S) := S \cup \{(tr, t, \ell+1) \mid (tr, s, \ell) \in S \wedge t \in f(s)\}$$

D-CBR

$$[\![\ell :: \texttt{cbr}\ \ b\, m\, n]\!](S) := S \cup \{(tr, s, pc) \mid (tr, s, \ell) \in S \wedge (b(s) \wedge pc = m \vee \neg b(s) \wedge pc = n)\}$$

D-COMM

$$[\![\ell :: \texttt{comm}\ \ ef\ f]\!](S) := S \cup \{(tr^\frown ev, t, \ell+1) \mid (tr, s, \ell) \in S \wedge ev \in ef(s) \wedge t = f(s, ev)\}$$

D-SEQ

$$[\![code_1 \oplus code_2]\!] := \big(\mu d.\ extend(code_1, code_2)\ (d)\big)$$

D-EXT

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

Figure 3: Denotational Semantics for CUC

See Figure 3 for the denotational semantics defined on the structured variant of CUC. The semantic function $[\![\cdot]\!]$ maps *code* to its *denotation*. We use the μ operator to denote a *least fixpoint* and the underlying chain-complete partial order is the point-wise subset relation ($f \leq g := \forall S.\ f(S) \subseteq g(S)$ where S is a set of states).

The smallstep semantics operates on single states, but we use sets of states in the denotational semantics to capture nondeterminism. Furthermore, we require sets also as input of the denotations so that we are able to chain functions directly. This facilitates sequential composition, as introduced in the background. Finally, we can keep track of all intermediate states and the corresponding traces, by simply extending the set input to the denotation ("$S \cup$" on the right-hand side of function definitions). This construction is also used in CSP and enables *prefix closure* (see Section 5.2.3).

The single step rules D-DO, D-CBR and D-COMM are very similar to their operational semantics counterparts. For all states where the program points to the label ℓ of the instruction, all successor states are added to the resulting set.

extend (D-EXT) extends a given denotation d with the executions of the two components $code_1$ and $code_2$ respectively. (*extend* corresponds to the function F in Section 3.2.) With its help we describe in D-SEQ the sequential composition $\oplus$ which is modeled as fixpoint of *extend* and thus as the repeated, possibly alternating application of the two components. Remember, that $\oplus$ is also the looping construct (see Section 3.1).

We have presented the operational and denotational semantics and show important properties of the latter in the next section.

5 Analysis of the Denotational Semantics

Our longterm goal is to define a refinement proof calculus, which relates CUC programs and CSP processes. This can only succeed, if we can ensure that the CUC semantics is close enough to the CSP semantics. Therefore we show a property related to structured code and two selected properties of CSP for the denotational semantics: *i)* Compositionality, which is also important for the future construction of a compositional calculus, and *ii)* prefix closure, which will be important when showing that a CUC program refines a CSP process considering failures (which allow for the verification of liveness properties) in future work. We start outlining the proof that the denotational semantics conforms to the operational multistep semantics, to ensure that the ideas formulated in the smallstep semantics are also captured by the denotational semantics.

5.1 Conformance Proof

By conformance we mean that, starting in an arbitrary state, the reachable states through the denotational semantics and the multistep semantics of a given structured code and its unstructured projection respectively are the same:

$$(tr_t, t, pc_t) \in [\![code]\!](\{(tr_s, s, pc_s)\}) \Longleftrightarrow (tr_s, s, pc_s) \xrightarrow{\cup_{sc} code}{}^{*} (tr_t, t, pc_t)$$

We outline the two directions of the conformance proof. We have formalized full proofs in Isabelle/HOL. First, we argue that all states in the returned set of a denotation are reachable through the multistep relation, and then the inverse direction. As we use the structuring technique for unstructured code from Saabas and Uustalu [7], we follow their proofs adapted for our denotational semantics. In both directions we use the similarity of single steps of the two semantics. The interesting case in each direction is the sequential composition case.

Denotational Semantics $\Longrightarrow$ Multistep We perform induction over the structure of *code*. The induction hypothesis states that for the subcomponents the denotational semantics implies the multistep semantics. To show the sequential composition case, we combine the hypotheses for the subcomponents using fixpoint induction.

Multistep $\Longrightarrow$ Denotational Semantics The main problem is that the multistep relation operates on instruction sets, thus is insensible to the structure of *code*. We need to extend the *multistep* relation with a step counter k, in order to invoke induction over the number of steps.

We still perform induction over the structure of *code*, but within each case we use case-distinction or induction over k, respectively.

An important insight from Saabas and Uustalu [7] for the induction in the case of the sequential composition is that each time we point into one part of the code, we make at least one step (in multistep), and thus in the other part, we have less steps left than we started with. This way we can decrease the value of k and use the induction hypothesis for the single component and for a smaller k, respectively.

5.2 Properties

In this subsection, we show that $\oplus$ is associative and commutative, and that the denotational semantics is compositional and prefix closed.

5.2.1 Associativity and Commutativity

Corollary 1 (Associativity and Commutativity)**.** $\oplus$ *is associative and commutative, i.e.,* $\forall code_1, code_2, code_3$*:*

$$[\![code_1 \oplus code_2]\!] = [\![code_2 \oplus code_1]\!]$$
$$[\![(code_1 \oplus code_2) \oplus code_3]\!] = [\![code_1 \oplus (code_2 \oplus code_3)]\!]$$

Proof. This follows directly from the conformance. As $\cup_{sc}$ removes all structure, any structure on a given set of instructions has the same semantic function. □

5.2.2 Compositionality

A semantics is *compositional*, if the semantics of a combination of components is defined directly using the semantics of the components and not details of them.

Theorem 2 (Compositionality)**.** *The denotational semantics is compositional, as the semantic function for sequential composition (*D-SEQ, D-EXT*) uses solely the semantic functions of the components and does not need further details about the components.* □

5.2.3 Prefix Closure

A set of traces is *prefix closed*, if for each trace in the set, all prefixes are contained in the set, too. We need to slightly adapt this definition, as the argument set of the denotation does not need to be prefix closed. So we formulate the property as an invariant over the semantic functions (see Section 6 for a definition of invariants):

Theorem 3 (Prefix Closure)**.** *If the argument set S is prefix closed, so is the returned set* $[\![code]\!](S)$.

Proof. The adapted prefix closed property holds: We only need to take a closer look at D-COMM, as it is the only rule changing the trace. Looking at D-COMM, we see that the yielded set is a union of the argument set S and all newly calculated states. As the new states only contain traces that are constructed by appending to existing traces, we get by using the assumption that all the prefixes of all constructed traces are also in the returned set. □

If we now assume that the set of initial states only contains empty traces, than the semantics for a given program returns sets which are prefix closed.

We have shown important properties, which are a prerequisite to relate the denotational semantics and CSP and have outlined a proof that the denotational semantics relates the same states as the smallstep semantics. In the next section, we give a preliminary rule for an invariant-based Hoare calculus.

```
    1 ::  do  ($\lambda\sigma.\ \{\sigma[free \leftarrow \text{True}]\}$)
⊕   2 ::  comm  ef  f  where
          ef  =  ($\lambda\sigma.\ \{in.x \mid \sigma(free) = \text{True} \land x \in T\}$
                      $\cup \{out.x \mid \sigma(free) = \text{False} \land x = \sigma(buffer)\}$)
          f   =  ($\lambda\sigma\ event.$ case $event$ of
                      | $in.x \Rightarrow \sigma[buffer \leftarrow x, free \leftarrow \text{False}]$
                      | $out.x \Rightarrow \sigma[free \leftarrow \text{True}]$)
⊕   3 ::  cbr  ($\lambda\sigma.\ \text{True}$)  2  2
```

Figure 4: One Place Buffer in CUC

6 Towards a Hoare calculus

Reasoning with semantics directly is generally complex. Therefore a proof calculus is usually introduced to make verification of properties manageable. We leave the proof calculus for future work, but we define an invariant and present a preliminary rule in this section.

In order to show properties of communicating, possibly nonterminating programs traditional Hoare logics with pre- and postconditions are not well suited, as the postcondition may never be reached. This is why we enhance the Hoare calculus with invariants to assert properties over the communication history. Using a calculus with invariants, we are still able to express properties about the program behavior even if the postcondition is never reached. This approach is, e.g., taken by Zwiers [11]. Invariants come quite naturally to reasoning about unstructured code, as the sequential composition is also the looping construct, and therefore already the sequential composition rule in the Hoare calculus defined by Saabas and Uustalu [7] uses invariants in pre- and postconditions.

We say an *invariant I*, holds for the semantic function of *code*, if it holds before, during, and after the execution for arbitrary initial states. Let *I* be a predicate on a *state*, then we define formally:

$$\frac{\forall S.\,(\forall s \in S.\, I(s)) \longrightarrow (\forall t \in [\![code]\!](S).\, I(t))}{I\colon [\![code]\!]}\ \textsc{Inv Intro}$$

In the example in the next section, we will use the following rule to reason about the components of sequential composition separately:

$$\frac{I\colon [\![code_1]\!] \qquad I\colon [\![code_2]\!]}{I\colon [\![code_1 \oplus code_2]\!]}\ \textsc{Inv}\ \oplus$$

We have proven soundness of the rule Inv $\oplus$ in Isabelle/HOL. The intuition is that sequential composition leads to the repeated execution of the code sections $code_1$ and $code_2$ in a possibly alternating order, and if both keep the invariant valid, so does the sequential composition as it is compositional. Admissibility of the invariant is ensured, as we defined it on single states.

7 Example

We demonstrate the applicability of our formal framework as presented above and show for a simple implementation (written in CUC) of a one-place buffer that it is correct, i.e., it only outputs what was

input before. We do so using the denotational semantics and a preliminary rule from the proof calculus. See Figure 4 for the code listing (we define $\oplus$ to be right associative). The code implements a buffer that can hold one value of type T. We explain the code 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.
`(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.

We show that, starting with an empty trace and with the program counter pointing to the first instruction (as described by the assertion *Pre*), only values that are input are also output (as described by *Inv*):

$$Pre(tr,\sigma,pc) := tr = \langle\rangle \wedge pc = 1$$

$$\begin{aligned}&Inv(tr,\sigma,pc) := tr \in TR_{even} \cup TR_{odd}\\ &\text{ where } TR_{even} := \{in.x^\frown out.x \mid x \in T\}^*\\ &\text{ and } TR_{odd} := \{tr^\frown in.x \mid tr \in TR_{even} \wedge x \in T\}\end{aligned}$$

The operator $\cdot^*$ being the *Kleene Star*, TR_{even} is the set of all *even* traces, i.e., where every input of a value is followed by the output of the same value. TR_{odd} is the set of *odd* traces, which are the even traces with an appended single input.
We name instructions by their label, e.g., in this example *instruction 3* means (3:: cbr $(\lambda\sigma.$ True$)$ 2 2).

Following the structure of the code, we start reasoning about the sequential composition of instructions 2 and 3, and then combine it with instruction 1.

To show *Inv* we need to strengthen it by adding state and program counter information, as the instructions depend on the state and on the program counter. This gives us a stronger invariant, denoted $I_{2,3}$, which we will use as invariant for instructions 2 and 3.

$$\begin{aligned}I_{2,3}(tr,\sigma,pc) :=\ &(tr \in TR_{even} \wedge \sigma(free) = \text{True}\\ &\vee tr \in TR_{odd} \wedge \sigma(free) = \text{False}\\ &\quad \wedge \sigma(buffer) = x \wedge \exists tr'.\, tr = tr'^\frown in.x)\\ &\wedge pc \in \{2,3\}\end{aligned}$$

We use the rule INV $\oplus$ to reason about instructions 2 and 3 separately.
$I_{2,3}$: $[\![2 :: \texttt{comm}\ ef\ f]\!]$ – Either we start with an even trace ($\in TR$) and $free =$ True, then we accept input and the second disjunct holds or we start in the second disjunct with an odd trace, output the corresponding value, and then end up in the first disjunct. In both cases we end up with $pc = 3$.
$I_{2,3}$: $[\![3 :: \texttt{cbr}\ (\lambda\sigma.\ \text{True})\ \ 2\ \ 2]\!]$ – `cbr` does neither change state nor trace, only the program counter is set to 2, so the validity of $I_{2,3}$ is preserved by instruction 3.

We are left to show that instruction 1 brings us from the initial state to the $I_{2,3}$: Assume *Pre* holds, then all states reachable through instruction 1 either still fulfill *Pre* or

$$I_2(tr,\sigma,pc) := tr = \langle\rangle \wedge \sigma(free) = \text{True} \wedge pc = 2$$

where the latter implies $I_{2,3}$. As instruction 1 does not change the state when $pc \in \{2,3\}$ holds,

$$I_{1,2,3}(tr,\sigma,pc) := Pre(tr,\sigma,pc) \vee I_{2,3}(tr,\sigma,pc)$$

is an invariant for instruction 1.

Following a similar argumentation, $I_{1,2,3}$ is also an invariant for instructions 2 and 3, as neither does change the state where $pc = 1$ holds and neither sets $pc = 1$, so in the first case the validity of *Pre* is maintained, and in the latter case *Pre* as a disjunct (of the postcondition) can be ignored. The easy combination of disjuncts is due to the fact that all instruction labels are unique. The future proof calculus will make use of this property to combine invariants.

As $I_{1,2,3}$ holds for both instruction 1 and instructions 2 and 3, we deduce using INV $\oplus$ that $I_{1,2,3}$ holds for all three instructions. As *Inv* follows from $I_{1,2,3}$, we have shown that starting in *Pre* the validity of *Inv* is maintained by the program.

We have shown how a preliminary Hoare calculus build on the denotational semantics can be used to reason about a nonterminating program. The proved property is an assertion about traces. Being able to show properties about traces will help to relate CUC and CSP.

8 Conclusion

In this paper, we have defined a compositional denotational semantics for a generic low-level language. Our denotational semantics is based on least fixpoints, it copes with communication, and it enables reasoning about nonterminating programs. We have proved that it conforms to the more intuitive operational semantics of our language. We have analyzed our semantics with respect to compositionality and prefix closure and have demonstrated how it could be used to verify properties of unstructured code using a proof rule based on invariants.

In future work, we aim at using our denotational semantics as the basis for a refinement proof calculus that enables convenient conformance proofs between unstructured low-level programs and CSP-based specifications. To this end, we have based the behavioral semantics part within our denotational semantics on denotations as used in the CSP process calculus. To realize such a refinement proof calculus, we will first define a Hoare calculus for our semantics from which we have shown a preliminary rule. The refinement proof calculus will relate CSP processes and proof obligations in the Hoare calculus. To fully support CSP, we will extend the denotational semantics with concurrency in the form of an *alphabetized parallel operator* as in CSP, i.e., synchronous progress on shared events, otherwise interleaving. To not only be able to show safety properties but also liveness properties using such a refinement proof calculus, we want to extend the denotational semantics and the Hoare calculus to cope with failures.

References

[1] B. Bartels & S. Glesner (2011): *Verification of Distributed Embedded Real-Time Systems and their Low-Level Implementation Using Timed CSP*. In T. Dan Thu & K. Leung, editors: *Proceedings of the 18th Asia Pacific Software Engineering Conference (APSEC 2011)*, IEEE Computer Society, pp. 195–202, DOI: 10.1109/APSEC.2011.52.

[2] Björn Bartels & Nils Jähnig (2014): *Mechanized, Compositional Verification of Low-Level Code*. In JuliaM. Badger & KristinYvonne Rozier, editors: *NASA Formal Methods*, *LNCS* 8430, Springer International Publishing, pp. 98–112, DOI: 10.1007/978-3-319-06200-6_8.

[3] Thomas Gibson-Robinson, Philip J. Armstrong, Alexandre Boulgakov & A. W. Roscoe (2014): *FDR3 - A Modern Refinement Checker for CSP*. In: *Tools and Algorithms for the Construction and Analysis of Systems - 20th International Conference, TACAS 2014*, pp. 187–201, DOI: `10.1007/978-3-642-54862-8_13`.

[4] Sabine Glesner, Björn Bartels, Thomas Göthel & Moritz Kleine (2010): *The VATES-Diamond as a Verifier's Best Friend*. In Simon Siegler & Nathan Wasser, editors: *Verification, Induction, Termination Analysis*, *LNCS* 6463, Springer Berlin Heidelberg, pp. 81–101, DOI: `10.1007/978-3-642-17172-7_5`.

[5] C. A. R. Hoare (1978): *Communicating Sequential Processes*. *Commun. ACM* 21(8), pp. 666–677, DOI: `10.1145/359576.359585`.

[6] Tobias Nipkow, Lawrence C. Paulson & Markus Wenzel (2002): *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*. *LNCS* 2283, Springer, DOI: `10.1007/3-540-45949-9`.

[7] Ando Saabas & Tarmo Uustalu (2005): *A compositional natural semantics and Hoare logic for low-level languages*. In: *Proceedings of the Second Workshop on Structured Operational Semantics*, Elsevier, pp. 151–168, DOI: `10.1016/j.entcs.2005.09.031`.

[8] Steve Schneider (1999): *Concurrent and Real Time Systems: The CSP Approach*. John Wiley & Sons, Inc., New York, NY, USA. Available at `http://www.computing.surrey.ac.uk/personal/st/S.Schneider/books/CRTS.pdf`.

[9] Hendrik Tews (2004): *Verifying Duff's device: A simple compositional denotational semantics for Goto and computed jumps*. Technical Report, Technische Universität Dresden. Available at `http://askra.de/papers.html.de`.

[10] Glynn Winskel (1993): *The Formal Semantics of Programming Languages: An Introduction*. MIT Press, Cambridge, MA, USA.

[11] J. Zwiers (1989): *Compositionality, Concurrency, and Partial Correctness: Proof Theories for Networks of Processes, and Their Relationship*. *LNCS* 321, Springer. Available at `http://www.gbv.de/dms/bowker/toc/9783540508458.pdf`.