Formal Verification of Low-Level Code in a Model-Based Refinement Process (Technical Report: Isabelle/HOL Formalization)

Nils Berg Björn Bartels Armin Danziger
Guilherme Grochau Azzi Matthias Bentert

TU Berlin
nils.berg@alumni.tu-berlin.de

This is the technical report for the Isabelle/HOL [6] formalization accompanying the dissertation of Nils Berg [3]. For explanations with regard to content please refer to the dissertation. The intention of this document is to give a mapping from the formalization in the dissertation to the formalization in Isabelle/HOL. Formalized are the parts where user interaction is required, i.e., the first part of the dissertation, where *Communicating Sequential Processes* (CSP) processes and *Communicating Unstructured Code* (CUC) programs are related. More specifically, the sections 5.2 to 5.5 of the dissertation are formalized. Due to technical implementation reasons and because the Isabelle/HOL formalization is a bit older than the formalization in the dissertation, which has evolved since, there is not a one-to-one mapping from the formalization in the dissertation to the formalization in Isabelle/HOL. Most notably, the operational semantics of CUC is not concurrent. The reasoning (that it is sufficient to consider single components) is explained in Section 5.6 of the dissertation. The concurrent cases of the correspondence proofs between the operational and denotational semantics that are missing in this formalization are given in the Appendix A.1 of the dissertation.

This document is intended to be read in its source code form in an Isabelle IDE (e.g. [2]). The source code of this document can be found in the file `START_HERE.thy`, which is published together with this document [4]. It will load all necessary dependencies. Please note that the 2017 version of Isabelle/HOL is required for the provided theory files. In a suitable Isabelle IDE, all terms in antiquotations (like this *concurrent-buffer-example*) are links to the definitions, lemmas, theories, etc. they refer to. To go to the link target, hold the Command Key on a Mac or the Control Key on Linux/Windows and and click the term. Hovering the link holding the

modifier key (without clicking) usually gives a small preview of the linked formalization.

In the Isabelle/HOL formalization, we use the concept of a locale. A common example for the application of a locale is group theory, where an arbitrary but fixed operator and a base set are assumed. The group theory locale can then be instantiated for concrete base sets and operators. In general, a locale allows to define abstract theories by fixing one or more abstract entities (in our case usually the unstructured code). The locale groups assumptions, definitions and lemmas (similar to a namespace). It allows for instantiation: the fixed abstract entities are replaced by a concrete entities. The instantiation of a locale requires proofs for all the assumptions and then provides instantiated versions all the definitions and lemmas that are grouped by the locale.

For the example in Section 5.5.1, we use the CSP formalization provided by the CSP-Prover library [5] by Yoshinao Isobe and Markus Roggenbach (in version 5-1-2016), which is available at its website [1]. We have included it with our source code.

In the remainder of this document, we give a mapping from the defintions, assumptions, lemmas, and theorems from dissertation to their counterparts in the Isabelle/HOL formalization. The numbering is the same as in the dissertation.

5.2 Syntax and Semantic States

Definition 5.1 (Basic Data Types)**.** The local states (LStates) are defined by the record *'data-store 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 *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, see for example *SFsemSigma*.

Definition 5.2 (Instructions of CUC)**.** The instructions of CUC are formalized in (*'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 wff_{sc}.

Definition 5.4 (Labels of a Program labels)**.** To extract the labels of an (unstructured) program, we use the definition $op \in_{lis}$.

Definition 5.5 (Concurrent Program *cp*)**.** We do not define unstructured concurrent programs in this formalization. This is due to the fact that we 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 semantics, e.g., see $densem_p$: We define successor states only for states whose tree structure matches the tree structure of the code.

Definition 5.6 (Structured Program *sp*)**.** The typ 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 $\cup_{sc}$.

Definition 5.8 (Labels of a Structured Program labels)**.** The labels of a structured program are extracted directly in this formalization via *definitions.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 wff_{sc}.

Definition 5.9 (Structured Concurrent Program *scp*)**.** The type of structured concurrent programs is defined by (*'state*, *'event*) *structured-code-parallel*.

5.3 Semantics

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') *liSet.Smallstep* inside a locale.

The concurrent operational semantics *are not* formalized. See the dissertation for the explanation. The concurrent denotational semantics *are* formalized.

The reflexive transitive hull of small step is defined in *liSet.multistep*. For historical reasons (see the next definition) multistep is defined by prepending to the execution traces. Which is in contrast to the dissertation, where it is defined by appending.

Definition 5.11 (Terminating Execution)**.** Terminating executions are defined by *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.

5.3.2 Defining Denotational Semantics with Fixpoints

For fixpoint theory, we use the existing HOL library *ccpo-class* which provides most notably

$$\begin{array}{l}[\![ccpo.admissible\ Sup\ op \leq P;\ monotone\ op \leq op \leq f;\ P\ (Sup\ \{\});\\ \bigwedge x.\ P\ x \Longrightarrow P\ (f\ x)]\!]\\ \Longrightarrow P\ (ccpo\text{-}class.fixp\ f)\end{array}$$

.

5.3.3 Traces Semantics

Definition 5.12 (Traces Semantics of CUC)**.** The sequential part of the traces semantics is defined in *densem*. The concurrent part of the traces semantics is defined in $densem_p$.

Definition 5.13 (Operational Characterization of the Traces of CUC)**.** This is not defined explicitly in the formalization. The operational and the traces semantics are related directly in a lemma. See the correspondence theorem 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 *Impl2.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.

$[\![liSet\ lis;\ \cup_{sc}\ c \subseteq lis;\ t \in [\![c]\!]S]\!]$
$\Longrightarrow \exists\, s{\in}S.\ (s,\ \cup_{sc}\ c,\ t) \in liSet.multistep\ lis$

$[\![liSet\ lis;\ \cup_{sc}\ strInstr \subseteq lis;\ s \in S;$
$(s,\ \cup_{sc}\ strInstr,\ t) \in liSet.multistep\ lis]\!]$
$\Longrightarrow t \in [\![strInstr]\!]S$

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).

$[\![(code1.0 \oplus code2.0)]\!]S = [\![(code2.0 \oplus code1.0)]\!]S$

$[\![liSet\ lis;\ \cup_{sc}\ ((code1.0 \oplus code2.0) \oplus code3.0) \subseteq lis]\!]$
$\Longrightarrow [\![(code1.0 \oplus code2.0 \oplus code3.0)]\!]S = [\![((code1.0 \oplus code2.0) \oplus code3.0)]\!]S$

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 *SFsemSigma.sfsem*. Only stable states are included in the semantics. That the stable failure semantics works as expected is shown in *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 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 denotational-parallel.parallelState*.

Definition 5.18 (Test for Normal State and Conversions)**.** 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. *SFsemSigma.trace-state-set-to-failure-set*

Definition 5.19 (Removal of Former Terminal Failures)**.** The former terminal failures are removed by $op\ -_{sf}$.

Definition 5.20 (Stable Failures Semantics of CUC)**.** The sequential part of the traces semantics is defined in *SFsemSigma.sfsem* The concurrent part of the traces semantics is defined in $sfsem_p$.

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.

$$\begin{array}{l}[\![SFsemSigma\ lis;\ (tr,\ s,\ X) \in SFsemSigma.sfsem\ \Sigma\ c\ \{(tro,\ so,\ Xo)\};\\ \cup_{sc}\ c \subseteq lis;\ wff_{sc}\ c]\!]\\ \Longrightarrow ((tro,\ NCS\ so),\ \cup_{sc}\ c,\ tr,\ NCS\ s) \in liSet.multistep\ lis\end{array}$$

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

$$\begin{array}{l}[\![SFsemSigma\ lis;\ \cup_{sc}\ c \subseteq lis;\ wff_{sc}\ c;\ (tr',\ NCstate.normal\ s',\ X') \in S;\\ ((tr',\ s'),\ \cup_{sc}\ c,\ tr,\ s) \in liSet.Exec\ lis;\ (tr',\ s') \neq (tr,\ s);\ X \subseteq \Sigma]\!]\\ \Longrightarrow (tr,\ NCstate.normal\ s,\ X) \in SFsemSigma.sfsem\ \Sigma\ c\ S\end{array}$$

and one for non-terminating executions

$$\begin{array}{l}[\![SFsemSigma\ lis;\\ ((tr',\ s'),\ \cup_{sc}\ c,\ tr'\ @\ r\ @\ [a]\ @\ l,\ s) \in liSet.multistep\ lis;\\ (tr',\ NCstate.normal\ s',\ X') \in S;\ \cup_{sc}\ c \subseteq lis;\ wff_{sc}\ c]\!]\\ \Longrightarrow \exists\ sx\ X\ evf\ sf.\\ \qquad (tr'\ @\ r,\ communication\ sx,\ X) \in SFsemSigma.sfsem\ \Sigma\ c\ S\ \wedge\\ \qquad (PC\ sx,\ comm\ evf\ sf) \in \cup_{sc}\ c\ \wedge\\ \qquad X \subseteq \Sigma - evf\ (R\ sx)\ \wedge\\ \qquad ((tr',\ s'),\ \cup_{sc}\ c,\ tr'\ @\ r,\ sx) \in liSet.multistep\ lis\ \wedge\\ \qquad ((tr'\ @\ r,\ sx),\ \cup_{sc}\ c,\ tr'\ @\ r\ @\ [a]\ @\ l,\ s) \in liSet.multistep\ lis\ \wedge\\ \qquad PC\ sx \in_{lis} \cup_{sc}\ c\end{array}$$

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 $s \in S \Longrightarrow s \in [\![code]\!]S$ 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

$$\begin{array}{l}[\![liSet\ lis;\ instrSet \subseteq lis;\\ ((tr',\ s'),\ instrSet,\ tr'\ @\ r\ @\ l,\ s) \in liSet.multistep\ lis]\!]\end{array}$$

$\Longrightarrow \exists\, sa.\ ((tr',\ s'),\ instrSet,\ tr' \mathbin{@} r,\ sa) \in liSet.multistep\ lis \wedge$
$((tr' \mathbin{@} r,\ sa),\ instrSet,\ tr' \mathbin{@} r \mathbin{@} l,\ s) \in liSet.multistep\ lis \wedge$
$(s \neq sa \vee l \neq [] \vee PC\ s \in_{lis} instrSet \longrightarrow PC\ sa \in_{lis} instrSet)$

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 linked to in the source file of this document: first for the sequential case, then for the concurrent case.

5.4 Hoare Calculus

Definition 5.21 (Hoare Tripel for CUC)**.** The Hoare triple is defined in *SFsemSigma.hoare-valid*.

Definition 5.22 (Hoare Calculus for CUC)**.** The Hoare calculus itself is defined in *SFsemSigma.hoare*.

Theorem 5.3 (Soundness of Our Hoare Calculus)**.** We have proven the soundness of our Hoare Calculus in

$[\![SFsemSigma\ lis;\ SFsemSigma.hoare\ \Sigma\ P\ code\ Q]\!]$
$\Longrightarrow SFsemSigma.hoare\text{-}valid\ \Sigma\ P\ code\ Q$

5.5 Relating CSP and CUC

Lemma 5.1 (Traces Imply Stable Failures)**.** When we only consider divergence free CUC programs, then all traces appear also in the stable failures semantics.

$[\![SFsemSigma\ lis;\ \forall\, tr\ s\ X.\ SFsemSigma.sfsem\ \Sigma\ c\ \{(tr,\ s,\ X)\} \neq \{\};\ wff_{sc}\ c;$
$(tr,\ s) \in [\![c]\!]failureSetToTraceStateSet\ S;\ \forall\, tr\ s\ X.\ (tr,\ s,\ X) \in S \longrightarrow N\ s;$
$\cup_{sc}\ c \subseteq lis]\!]$
$\Longrightarrow \exists\, s\ X.\ (tr,\ s,\ X) \in SFsemSigma.sfsem\ \Sigma\ c\ S$

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.

$[\![SFsemSigma\ lis;\ all\text{-}commands\text{-}are\text{-}followed\text{-}by\text{-}comm\text{-}or\text{-}leave\ c;$
$SFsemSigma.sfsem\text{-}refinement\ (SFsemSigma.sfsem\ \Sigma\ c\ S)$
$(SFsemSigma.sfsem\ \Sigma\ c'\ S);$
$\cup_{sc}\ c \subseteq lis;\ \cup_{sc}\ c' \subseteq lis;\ wff_{sc}\ c;\ wff_{sc}\ c';$
$\forall\, tr\ s\ X.\ (tr,\ s,\ X) \in S \longrightarrow N\ s;$
$\forall\, f.\ (\exists\, lbl.\ (lbl,\ do\ f) \in \cup_{sc}\ c) \longrightarrow (\forall\, x.\ f\ x \neq \{\});$
$\forall\, ef.\ (\exists\, lbl\ sf.\ (lbl,\ comm\ ef\ sf) \in \cup_{sc}\ c) \longrightarrow (\forall\, x.\ ef\ x \neq \{\})]\!]$
$\Longrightarrow [\![c]\!]failureSetToTraceStateSet\ S \subseteq_T [\![c']\!]failureSetToTraceStateSet\ S$

Assumption 5.7 (Divergence Freedom of CUC Programs)**.** We assume divergence freedom directly in lemmas and theorems (see the theorems linked 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 *static-code-analysis*.

5.5.1 Example

Reminder: For the formalization of CSP, we use the CSP Prover library [5] by Yoshinao Isobe and Markus Roggenbach.

We have formalized the example in the theory file *concurrent-buffer-example*. 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 dissertation, in the formalization we need many more fine grained definitions and lemmas.

The CUC program is defined in *SingleBuffer.LIS*. The CSP specification is defined in the assumptions of the lemmas that use it, e.g.

$$\begin{array}{l}[\![SingleBuffer\ TIn\ TOut;\ SingleBuffer.Conn\ TIn\ TOut\ (tr,\ X);\\ \ [[\$spec]]Ff\ [[PN]]Ffix =\\ \ [[Pair\ TIn\ ?\ k\ -{>}\ (TOut,\ k)\ -{>}\ \$spec]]Ff\ [[PN]]Ffix[\![\\ \Longrightarrow (CV_{tr}\ tr\ EvtCV,\ CV_X\ X\ EvtCV)\ :f\ sndF\ ([[\$spec]]Ff\ [[PN]]Ffix)\end{array}$$

It is defined in fixpoint notation

$$[[\$spec]]Ff\ [[PN]]Ffix = [[Pair\ TIn\ ?\ k\ -{>}\ (TOut,\ k)\ -{>}\ \$spec]]Ff\ [[PN]]Ffix$$

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, F-empty and F-full, which are named F-even and F-odd in the Isabelle/HOL formalization. The sufficient property is caled "connecting property" or "conn".

The sufficient property implies the CSP specification

$$\begin{array}{l}[\![SingleBuffer\ TIn\ TOut;\ SingleBuffer.Conn\ TIn\ TOut\ (tr,\ X);\\ \ [[\$spec]]Ff\ [[PN]]Ffix =\\ \ [[Pair\ TIn\ ?\ k\ -{>}\ (TOut,\ k)\ -{>}\ \$spec]]Ff\ [[PN]]Ffix[\![\\ \Longrightarrow (CV_{tr}\ tr\ EvtCV,\ CV_X\ X\ EvtCV)\ :f\ sndF\ ([[\$spec]]Ff\ [[PN]]Ffix)\end{array}$$

The sufficient property holds for all stable failures of the CUC program:

$$
\begin{array}{l}
[\![SingleBuffer\ TIn\ TOut; \\
\ (tr,\ s,\ X) \\
\ \in SFsemSigma.sfsem\ (SingleBuffer.\Sigma\ TIn\ TOut) \\
\quad (SingleBuffer.TSeq123\ TIn\ TOut)\ \{x.\ Impl2.TPre\ x\}]\!] \\
\Longrightarrow SingleBuffer.Conn\ TIn\ TOut\ (tr,\ X)
\end{array}
$$

The conclusion that the CUC program refines the CSP process

$$
\begin{array}{l}
[\![SingleBuffer\ TIn\ TOut; \\
\ [[\$spec]]Ff\ [[PN]]Ffix = \\
\ [[Pair\ TIn\ ?\ k\ ->\ (TOut,\ k)\ ->\ \$spec]]Ff\ [[PN]]Ffix]\!] \\
\Longrightarrow SFsemSigma.spec\text{-}impl\text{-}refinement\text{-}sf\text{-}seq\text{-}no\text{-}conn\text{-}pre\text{-}set \\
\quad (SingleBuffer.\Sigma\ TIn\ TOut)\ (\$spec)\ PN\ EvtCV \\
\quad (SingleBuffer.TSeq123\ TIn\ TOut)
\end{array}
$$

We have proven the properties about the single buffer inside a locale *Single-Buffer*. 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

$$\$Spec1\ |[CS,CS]|\ \$Spec2\ PNdef\ <=(EvtCV)=F\ impl12$$

.

Bibliography

[1] Csp-prover website. URL https://staff.aist.go.jp/y-isobe/CSP-Prover/CSP-Prover.html.

[2] Isabelle IDE 2017. URL https://isabelle.in.tum.de/website-Isabelle2017.

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

[4] N. Berg, B. Bartels, A. Danziger, G. Grochau Azzi, and M. 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:http://dx.doi.org/10.14279/depositonce-8636.

[5] Y. Isobe and M. Roggenbach. A generic theorem prover of csp refinement. In N. Halbwachs and L. 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. doi:10.1007/978-3-540-31980-1_8.

[6] T. Nipkow, L. C. Paulson, and M. Wenzel. *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*, volume 2283 of *LNCS*. Springer, 2002. doi:10.1007/3-540-45949-9.