Designing an Unbounded Buffer in Rely-Guarantee

Diego Dias and Leo Freitas
Designing an Unbounded Buffer in Rely-Guarantee

D. Dias and L. Freitas

Abstract

We use a variety of consumer-producer specifications to illustrate the range of choices and rationale that arise in concurrent specifications using rely-guarantee. We also extend postconditions with a new argument to make explicit the environment assigned to the process, and present a semantics for possible values that explores the new parameter in postconditions. All specifications presented in this document were mechanised in Isabelle/HOL.
Abstract

We use a variety of consumer-producer specifications to illustrate the range of choices and rationale that arise in concurrent specifications using rely-guarantee. We also extend postconditions with a new argument to make explicit the environment assigned to the process, and present a semantics for possible values that explores the new parameter in postconditions. All specifications presented in this document were mechanised in Isabelle/HOL.

About the authors

Diego Dias is a PhD student in Formal Methods at Newcastle University, working under supervision of Dr Leo Freitas. Diego gained his BSc in Computer Science at Federal University of Bahia, Brazil. There he worked in collaboration with Leo Freitas on a mechanisation of a simple kernel using Z notation. He continued his studies with a MSc in Computer Science at Federal University of Pernambuco, Brazil. His MSc thesis 'Behavioural Preservation in Fault Tolerant Patterns' applies HOL4 to formalise a notion of behavioural preservation of replication patterns used in the industry.

Leo Freitas is a lecturer in Formal Methods working on the EPSRC-funded AI4FM project at Newcastle University. Leo received his PhD in 2005 from the University of York with a thesis on 'Model Checking Circus', which combined refinement-based programming techniques with model checking and theorem proving. Leo's expertise is on theorem proving systems (e.g. Isabelle, Z/EVES, ACL2, etc.) and formal modelling (e.g. Z, VDM, Event-B), with particular interest on models of industrial-scale. Leo has also contributed extensively to the Verified Software Initiative (VSTTE).

Suggested keywords

ABSTRACTION
MECHANISMS
POSSIBLE VALUES
RELY-GUARANTEE
Designing an unbounded buffer in rely-guarantee

Diego Dias, Leo Freitas

School of Computing Science
Newcastle University
NE1 7RU, UK

Abstract. We use a variety of consumer-producer specifications to illustrate the range of choices and rationale that arise in concurrent specifications using rely-guarantee. We also extend postconditions with a new argument to make explicit the environment assigned to the process, and present a semantics for possible values that explores the new parameter in postconditions. All specifications presented in this document were mechanised in Isabelle/HOL.

Keywords: abstraction mechanisms, possible values, rely-guarantee

1 Introduction

This report continues the investigation sketched in [DFJ14] on abstraction mechanisms in rely-guarantee [Jon83]. We propose a syntactic and semantic definition for possible values [JP11]: a convention to refer to intermediate states that arise due to interference in concurrent programs. Our definition is illustrated via its application to a producer-consumer specification operating on a buffer. Two different representations for the buffer are explored: sequence, and sequence with a counter. The findings in this document support the claims from [DFJ14]: i) possible values enhances the separation of concerns between the rely and postcondition, and (ii) possible values reduces the gap between sequential and concurrent versions of the same process.

Rely-guarantee [Jon83,Jon96] is a concurrent program logic based on Hoare logic [Hoa69]. A specification of a process in this formalism is a 4-tuple composed by a precondition (pre), a rely (rely), a guarantee (guar) and a postcondition (post). In the following specification sketch, $PNAME$ is the process name, $a$ is an input variable of type $T$ and $r$ is a output variable of type $S$. The precondition ($pre_{PNAME}$) is an one-state predicate, and the rely, guarantee and postcondition ($rely_{PNAME}$, $guar_{PNAME}$ and $post_{PNAME}$, respectively) are two-state predicates.

\[
\begin{align*}
PNAME\ (a : \ T)\ r : S \\
pre\ &\ pre_{PNAME} \\
rely\ &\ rely_{PNAME} \\
guar\ &\ guar_{PNAME} \\
post\ &\ post_{PNAME}
\end{align*}
\]
The precondition is an assumption about the initial state a process is expected to deal with, and the postcondition is a relation between the initial and final states (there can exist more than one final state). Guarantee is a restriction made upon the process about the transitions between intermediate states it can perform, and the rely is an assumption about the transitions between intermediate states the environment can perform.

This specification is to be interpreted as “if $\text{pre}_{\text{PNAME}}$ holds for the initial state, and every visible intermediate transition of the environment is bounded by $\text{rely}_{\text{PNAME}}$, then $\text{PNAME}$ must establish the postcondition between the initial and final state, and every visible intermediate transition of $\text{PNAME}$ must respect $\text{guar}_{\text{PNAME}}$. Additionally, $\text{post}_{\text{PNAME}}$ holds between the initial state and all states subsequent to the final state”. Definition of processes and their parallel composition in rely-guarantee generate proof obligations that must be discharged to ensure consistence [CJ07,IJH12]. The rely-guarantee proof obligations (POs) are explained in the Section 2.

1.1 Main contributions

The main contributions of this document are:

- Formal definition and Isabelle mechanisation of possible values and parametrisation of postconditions;
- A proof obligation for parametrised specifications;
- Convention to simplify the use of possible values and parametrisation;
- A collection of lemmas about possible values;
- Comparison among invariant templates;
- Notion of safety properties in rely-guarantee;
- Directions for future work.

Section 2 revisits the POs of rely-guarantee and introduces two new POs. Sections 3 and 4 present different approaches to specifying a single producer-consumer pair acting in an unbounded buffer using rely-guarantee. Possible values and parametrisation of postconditions are introduced in Section 3; and the usage of auxiliary variables as alternative to parametrisation is illustrated in Section 4. Section 5 discusses the proposed extensions to rely-guarantee and Section 6 covers the mechanisation issues in Isabelle/HOL. Conclusions are summarised in Section 7.

2 Rely-guarantee proof obligations

The POs shown here are required to hold for rely-guarantee specifications. We classify these POs into four groups: coherence and stability checkings come from [CJ07], feasibility checking comes from [IJH12] and closure checking is proposed by us to discharge the assumption made in [CJ07] about the reflexivity and transitivity of rely and guarantee relations.
In the next subsections, si denotes initial state (or before state), sf denotes final state (or after state), and sint denotes intermediate state. Additionally, $A \circ B(si, sf) \equiv \exists sint. A(si, sint) \land B(sint, sf)$, $\land$ denotes generalised conjunction, and $\Rightarrow$ denotes implication.

2.1 Closure checking

The next two POs must be proved for each process $p$, hence $\forall p$. They ensure that rely and guarantee relations are both reflexive and transitive. The reason for closure checking is to discharge the assumption made in [CJ07] on the reflexive transitive closure of rely and guarantee conditions. Reflexivity allows for atomic steps that do not change the state, and transitivity implies that consecutive steps of a process (or environment) must satisfy the guarantee condition (rely condition, respectively) when taken as a whole.

\[
\text{RT\_Guar: } \forall p \cdot \forall u, s, t \cdot \text{guar}_p(u, u) \land \text{guar}_p(u, s) \land \text{guar}_p(s, t) \Rightarrow \text{guar}_p(u, t)
\]

\[
\text{RT\_Rely: } \forall p \cdot \forall u, s, t \cdot \text{rely}_p(u, u) \land \text{rely}_p(u, s) \land \text{rely}_p(s, t) \Rightarrow \text{rely}_p(u, t)
\]

Alternatively, the assumption made in [CJ07] could be avoided if the occurrences of rely-guarantee conditions in next POs were replaced by their respective reflexive-transitive closures.

2.2 Coherence checking

In the general case of the parallel composition of $n$ processes, for each process $p$, the conjunction of the guarantees of all other processes rather than $p$ must imply the rely of $p$. For the parallel composition of two processes this means that the guarantee of one must imply the rely of the other.

\[
\text{Coh\_GuarRely: } \forall p \cdot \forall si, sf \cdot ((\land_{x \neq p} \text{guar}_x(si, sf)) \Rightarrow \text{rely}_p(si, sf))
\]

This proof obligation is assigned to the parallel composition rule of rely-guarantee. It ensures that the processes composed in parallel can coexist [Jon83,CJ00]. The rule for parallel composition can be found in [CJ07].

2.3 Stability checking

Stability means that the postcondition to be established by a process cannot be invalidated by the environment, and cannot depend on actions of the environment to be established. Additionally, it also means that preconditions cannot be invalidated by the actions of the environment. Each process $p$ in a specification must be stable under the interference from the environment. We say that a process is stable if it satisfies three POs: Sta\_PreRely, Sta\_RelyPost, and Sta\_PostRely. These POs are taken from [CJ07]. The first PO states that preconditions must tolerate the interference on the initial state:
\textbf{Sta\_PreRely}: $\forall p \cdot \forall si, sint \cdot \text{pre}_p(si) \land \text{rely}_p(si, sint) \Rightarrow \text{pre}_p(sint)$.

The second PO states that if a process establishes the postcondition from an intermediate state obtained from \textit{si} through interference, it must also be able to establish the postcondition directly from \textit{si}.

\textbf{Sta\_RelyPost}: $\forall p \cdot \forall si, sf \cdot \text{pre}_p(si) \land (\text{rely}_p \circ \text{post}_p)(si, sf) \Rightarrow \text{post}_p(si, sf)$.

Intuitively this means that processes cannot depend on interference in the initial state to establish their postcondition. The last stability PO states that if a process establishes its postcondition, the environment cannot invalidate the postcondition.

\textbf{Sta\_PostRely}: $\forall p \cdot \forall si, sf \cdot (\text{post}_p \circ \text{rely}_p)(si, sf) \Rightarrow \text{post}_p(si, sf)$.

This means that the postcondition must hold not only between the initial and final state, but between the initial state and all states subsequent to the final state.

2.4 Rely-guarantee feasibility checking

Feasibility establishes the possibility of implementing the postcondition from a state that satisfies the precondition by using only atomic steps that respect the guarantee condition [IJH12].

\textbf{RG\_Feasibility}: $\forall p \cdot \forall si, inp \cdot \text{pre}_p(si, inp) \Rightarrow \exists sf, out \cdot \text{post}_p(si, sf, inp, out) \land \text{guar}_p(si, sf)$.

The postcondition in this PO includes inputs and outputs. A discussion about the mechanisation of POs can be found in Section 6.5. Notice that the rely condition does not appear in this PO; this reflects the fact that stability and feasibility are treated as different concerns, i.e., a feasible specification can be unstable and vice-versa.

2.5 Discussion

The number of proof obligations in a rely-guarantee specification grows linearly with the number of processes composed in parallel. Table 1 summarises the amount of proof obligations for the scenario of two and \textit{n} processes.

For the sake of simplicity, we omit inputs and outputs in \textit{Sta\_RelyPost} and \textit{Sta\_PostRely}. The mechanised version of these POs differ from those shown here only by passing extra parameters, namely the inputs and outputs, to the components of the specification.
Designing an unbounded buffer in rely-guarantee

<table>
<thead>
<tr>
<th>Proof obligation</th>
<th>2 processes</th>
<th>n processes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Closure checking</td>
<td>4</td>
<td>2n</td>
</tr>
<tr>
<td>Coherence checking</td>
<td>2</td>
<td>n</td>
</tr>
<tr>
<td>Stability checking</td>
<td>6</td>
<td>3n</td>
</tr>
<tr>
<td>Feasibility checking</td>
<td>2</td>
<td>n</td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td><strong>14</strong></td>
<td><strong>7n</strong></td>
</tr>
</tbody>
</table>

Table 1: Proof obligations - General view

We also propose additional POs which are summarised in Section 5. A library of POs was built in Isabelle\(^1\); it centralises the risk of typesetting errors to compromise the verification. It is still our responsibility to instantiate the POs properly as Isabelle does not alert us if we forget to instantiate POs, or do not instantiate them properly. General properties of possible values and invariants discussed in later sections of this document were also mechanised\(^2\). We are currently looking into using Isabelle locales \((i.e., \text{theory modules})\) to enforce the precise instantiation of POs.

3 State as a sequence

In this section we discuss five specifications for a consumer-producer pair operating on a buffer modelled as a sequence. This is to explore the space of viable specifications with respect to their clarity, as well their expressivity. The first two specifications fail a sanity checking, \((i.e., \text{a property of interest for producer-consumer specifications})\). The third specification uses possible values, but also fails the sanity checking. The fourth specification differs from the third by extending rely-guarantee with an extra parameter in the postconditions; its sanity checking succeeds. The fifth specification uses the first specification as its basis, and proposes a convention to enhance the usability of possible values. All proof obligations for the specifications in this section were discharged using Isabelle/HOL (Version 2013-2). Section 6.3 discusses the translation from VDM to Isabelle.

3.1 Non-blocking buffer

For the first specification\(^3\), a buffer is represented by a sequence of type T. The consumer removes the head of the sequence, and the producer concatenates at the end of the sequence. We use VDM-style notation to write the specification [Jon83].

\[
\text{Buffer} = T^* 
\]

\(^1\) Filename: RGPOs.thy  
\(^2\) Filename: Metatheory.thy  
\(^3\) Filename: sequence/Sec_31_NonBlockingBuffer.thy
is_empty : Buffer → \mathbb{B}

\[ is_empty(buf) \triangleq buf = [] \]

\textit{CONS} () res : T
\textbf{pre} \leftarrow is_empty buf
\textbf{rely} buf prefix buf'
\textbf{guar} buf' suffix buf
\textbf{post} res = hd buf ∧ (tl buf) prefix buf'

\textit{PROD} (e : T)
\textbf{pre} true
\textbf{rely} buf' suffix buf
\textbf{guar} buf prefix buf'
\textbf{post} buf' suffix (buf ↷ [e])

This specification represents a producer-consumer pair with no locks, i.e., the producer is never blocked, and the consumer can run whenever the buffer is not empty. The consumer removes elements from the left, and the producer adds elements to the right of the buffer. To ensure stability, the postconditions use \textit{prefix} and \textit{suffix} to cope with the effects from the environment. The repetition of the operators used to define the rely condition in the postconditions means that the interference is being directly encoded into the postconditions. The guarantees make clear that neither insertion of elements by the consumer, nor exclusion of elements by the producer, are allowed.

Looking carefully to the producer’s postcondition one can see that an implementation that refuses to produce when the initial buffer is empty complies with the specification. We call such an implementation a \textit{LazyProducer}.

\textit{LazyProducer} : Buffer → T → Buffer
\textbf{LazyProducer}(buf, e) \triangleq \begin{cases} [] & \text{if } buf = [] \\ [] & \text{else } buf ↷ [e] \end{cases}

The last row on Table 2 states that this first specification fails to rule the \textit{LazyProducer} implementation out, as desired. Thus we have work to do.
Table 2: Summary of proof obligations for the non-blocking buffer.

The sanity checking in the last row is indeed the feasibility PO (see RG\textunderscore Feasibility in Section 2.4), where the after state (i.e., buf') is instantiated as LazyProducer(buf, e). The desired result is to get a counterexample in Isabelle. In this case, a proof was found using the mentioned instantiation. This proof means that we can find a refinement for the producer process (PROD) that uses LazyProducer to compute the after state (i.e., buf').

### 3.2 Blocking buffer

This specification\(^4\) is akin to the counting semaphore example from [OHe07]. Here we depart from the non-blocking buffer, and strengthen the producer’s post-condition to ensure that the resulting buffer is not empty (i.e., \(\neg \text{is\_empty buf}\')). This rules out LazyProducer, but requires adjustment in the producer’s rely in order to maintain the postcondition stable (see Sta\textunderscore PostRely PO in Section 2.3). The modification on the producer’s rely is to ensure that the environment never empties the buffer (i.e., \(\text{is\_empty buf}' \Rightarrow \text{is\_empty buf}\)). Consequently, to maintain coherence (see Coh\textunderscore GuarRely PO in Section 2.2) we need to strengthen the consumer’s guarantee. Finally, in order to keep the consumer feasible (see Feasibility PO in Section 2.4) we strengthen its precondition (i.e., \(\text{len buf} > 1\)). The strengthening of this precondition can be understood as the introduction of a lock in the last node of the buffer, i.e., a blocking mechanism.

\[\text{Buffer} = T^*\]

\[\text{establishingInv : Buffer} \rightarrow \mathbb{B}\]

\[\text{establishingInv}(b) \triangleq \neg \text{is\_empty b}\]

\[P : \text{Buffer} \rightarrow \mathbb{B}\]

\[P(b) \triangleq \text{is\_empty b}\]

\(^4\) Filename: sequence/Sec_32_BlockingBuffer.thy.
This specification introduces the concepts of establishing invariant and safety properties. Establishing invariant is an one-state predicate that continues to hold once it is established; it is not required to be established when the system is initialised. If an establishing invariant \( ev \) is defined, the predicate \( ev S \Rightarrow ev S' \) becomes part of all rely and guarantee conditions. Section 5.5 is dedicated to establishing invariants. Safety properties are one-state predicates that are never satisfied by intermediate states of an implementation; they are discussed in Section 5.1. For the blocking buffer, we proved that \( P \) is a safety property.

This specification rules out LazyProducer at the cost of introducing a lock in the last node of the buffer. It is not a definitive specification yet, as it is unable to rule out another sanity check of a LazyConsumer, an implementation that refuses to consume in a particular case, namely, when the tail of the buffer is a prefix of itself.

A scenario where the LazyConsumer refuses to consume is one where all elements in the buffer are the same, e.g. \( buf = [a, a, a] \). In a sequential scenario (i.e., one without interference), we would expect the final buffer to be \( buf' = [a, a, a] \) after consumption, however, the LazyConsumer does not remove the leftmost element and terminate with \( buf' = [a, a, a, a] \).

Table 3 summarises the theorems proved for this specification. The sanity check is the feasibility PO (see RG_Feasibility in Section 2.4). We instantiated \( buf' \) as LazyConsumer\((buf)\) on the proof of this sanity check. The desired result would be to find a counterexample in Isabelle. Instead, a proof using the mentioned instantiation was found. This proof means that we can find a refinement.
for the consumer process (CONS) that uses LazyConsumer to compute the after state (i.e., \( \text{buf}' \)).

<table>
<thead>
<tr>
<th>Proof obligation</th>
<th>Status</th>
</tr>
</thead>
<tbody>
<tr>
<td>Closure checking</td>
<td>✓</td>
</tr>
<tr>
<td>Coherence checking</td>
<td>✓</td>
</tr>
<tr>
<td>Stability checking</td>
<td>✓</td>
</tr>
<tr>
<td>Feasibility checking</td>
<td>✓</td>
</tr>
<tr>
<td>Sanity checking</td>
<td>✗</td>
</tr>
</tbody>
</table>

Table 3: Summary of proof obligations for the blocking buffer.

Alternatively, this specification could also have used other spurious implementation to illustrate the sanity checking. A simple variant of LazyProducer is an implementation that refuses to produce when all the elements in the buffer are identical to the element to be inserted, e.g. \( \text{buf} = [e, e, e] \).

### 3.3 Posvals blocking buffer

In the attempt to rule out LazyConsumer, we changed the previous consumer and producer’s postcondition by introducing the possible values operator\(^5\). The introduction of this operator does not sort out the problem of ruling out spurious implementations, but serves to illustrate a different approach to specifying a consumer-producer pair. In the following definitions we use State when referring to Buffer to mean that a definition is independent of the underlying state, and we use Buffer when a definition is dependent of the underlying state.

\[
\text{State} \triangleq \text{Buffer}
\]

\[
\text{pv} : \text{State} \rightarrow (\text{State} \rightarrow \text{State} \rightarrow \mathbb{B}) \rightarrow \mathcal{P} (\text{State})
\]

\[
\text{pv}(s, r) \triangleq \{ \text{sint} \mid r \; s \; \text{sint} \}
\]

The \( \text{pv} \) (i.e., possible values) operator takes a state \( s \), and a rely relation \( r \) and builds the set of reachable states. The definition of possible values comprises only states a process should tolerate to be restored in, if it is interrupted in a state \( s \) while running in an environment bounded by \( r \). Three additional operators are defined in this specification, namely \( \text{lift}_-\_tl \), \( \text{lift}_-\_concat \) and \( \text{lift}_-\_pv \). Such lifting operators are pervasive in the presence of \( \text{pv} \) in specifications. They lift the \( \text{pv} \) definition and the sequence operators used to a set of states.

\(^5\) Filename: sequence/Sec_3_3_PosvalsBlockingBuffer.thy.
lift\_tl : \mathcal{P}(Buffer) \rightarrow \mathcal{P}(Buffer)

\begin{align*}
\text{lift\_tl}(S) & \triangleq \{ t1 \; x \mid x \in S \land \neg\text{is\_empty} \; x \} \\
\text{lift\_concat} : \mathcal{P}(Buffer) \rightarrow T \rightarrow \mathcal{P}(Buffer)
\end{align*}

\begin{align*}
\text{lift\_concat}(S, e) & \triangleq \{ x \uparrow [e] \mid x \in S \land \text{true} \} \\
\text{lift\_pv} : \mathcal{P}(State) \rightarrow (State \rightarrow State \rightarrow B) \rightarrow \mathcal{P}(State)
\end{align*}

\begin{align*}
\text{lift\_pv}(S, r) & \triangleq \bigcup \{ pv \; sint \; r \mid sint \in S \}
\end{align*}

The \textit{pv} operator is used to abstract the interference on the initial state, whereas the lifted operators are used to calculate the result over all eligible possible initial values, and the lifted \textit{pv} to abstract the interference on the intermediate result computed by the other lifted operators. In general, the lifted version of \textit{op}: \(X \rightarrow \cdots \rightarrow X\) to \(\mathcal{P}(X)\) is \text{lift\_op}(S, \cdots) \triangleq \{ op(x, \cdots) \mid x \in S \land \text{pre\_op}(x, \cdots) \}.

The inclusion of \text{pre\_op}(x, \cdots) in the set comprehension filters out values that do not satisfy the precondition of \textit{op}. The decision of including the \text{pre\_op}(x, \cdots) as a guard is discussed in Section 5.9. Using these operators, the specification of the producer and consumer requires changing postconditions of Section 3.2 to:

\text{(CONS)} \; res: T
\begin{align*}
\text{post} \; res & = \text{hd} \; buf \land \\
& \text{buf}' \in \text{lift\_pv} \left( \text{lift\_tl} \left( \text{pv\;buf\;rely\_CONS} \right) \right) \text{relvyCONS}
\end{align*}

\text{(PROD)} \; (e: T)
\begin{align*}
\text{post} & \neg \text{is\_empty} \; buf' \land \\
& \text{buf}' \in \text{lift\_pv} \left( \text{lift\_concat} \left( \text{pv\;buf\;rely\_PROD} \right) \; e \right) \text{relvyPROD}
\end{align*}

These postconditions are proved equivalent to the ones in Section 3.2. Theorem 1 formalises the equivalence between the postconditions of this section and their respective previous versions.

**Theorem 1 (Blocking buffer equivalence).** The postconditions of the posvals buffer and blocking buffer are equivalent.

\begin{align*}
\vdash (\text{post\_CONS\_BlockingBuffer} \iff \text{post\_CONS\_PosvalsBuffer}) \land \\
(\text{post\_PROD\_BlockingBuffer} \iff \text{post\_PROD\_PosvalsBuffer})
\end{align*}

A consequence of the equivalence is that the posvals buffer is unable to rule out \textit{LazyConsumer} as shown in the last row of Table 4.
Although the new postconditions do not solve the problem from Section 3.2, they are useful to illustrate an interference abstraction mechanism through possible values.

3.4 Parametrised blocking buffer

In this section, our attempt to rule out the LazyConsumer succeed. It differs to the previous specification by its postconditions. Here, the postconditions take an extra parameter, which is the rely assigned to them. This novel technique is further referred as parametrisation.

The parametrised postconditions differ from the previous ones (Section 3.3) by replacing all occurrences of relyCONS and relyPROD by $r$ and by making the rely relation $r$ a parameter.

\[
\begin{align*}
CONS\;() \; res;\; T \\
\text{post }\; r \equiv res = \text{hd } buf \land \\
& buf' \in \text{lift}_{pv} (\text{lift}_{tl} (pv \; buf \; r)) \; r \\
\end{align*}
\]

\[
\begin{align*}
PROD\; (e;\; T) \\
\text{post }\; r \equiv \neg \text{is\_empty } buf' \land \\
& buf' \in \text{lift}_{pv} (\text{lift}_{concat} (pv \; buf \; r) \; e) \; r \\
\end{align*}
\]

The introduction of a parameter in the postcondition is justified by the proposal of a parametrised feasibility checking. The new PO states that there exists a final state that satisfies the postcondition in presence of interference, and also satisfies the postcondition when interference is removed. The parametrised feasibility checking for the consumer is:

\[
\forall si \cdot \text{preCONS}(si) \Rightarrow \exists sf, res \cdot \text{guarCONS}(si, sf) \land \\
\text{postCONS relyCONS (si, sf, res)} \land \\
\text{postCONS ID (si, sf, res)}
\]

\footnote{Filename: sequence/Sec_34_ParametrisedBlockingBuffer.thy.}
where $ID$ is the identity relation over the state:

$$ID : \text{State} \rightarrow \text{State} \rightarrow \mathbb{B}$$

$$ID(b, b') \triangleq b = b'$$

The parametrised feasibility PO can be used as a sanity check to rule out spurious implementations such as LazyConsumer and LazyProducer. The intuition behind the sanity check is that, if we have a sequential specification for comparison purposes, and we deploy the concurrent implementation in an environment that does not interfere, its behaviour should comply with the sequential specification.

<table>
<thead>
<tr>
<th>Proof obligation</th>
<th>Status</th>
</tr>
</thead>
<tbody>
<tr>
<td>Closure checking</td>
<td>✓</td>
</tr>
<tr>
<td>Coherence checking</td>
<td>✓</td>
</tr>
<tr>
<td>Stability checking</td>
<td>✓</td>
</tr>
<tr>
<td>Feasibility checking</td>
<td>✓</td>
</tr>
<tr>
<td>Sanity checking</td>
<td>✓</td>
</tr>
</tbody>
</table>

Table 5: Summary of proof obligations for the fourth specification.

Table 5 summarises the theorems proved for this specification. The tick at the last row of the table means that if we instantiate the parametrised feasibility PO using the spurious implementations, Isabelle finds a counterexample. The counterexample means that the behaviour of the parametrised posvals buffer is different from that of the spurious implementations, thus one cannot find a refinement of the producer process (or consumer process) that uses LazyProducer (or LazyConsumer, respectively) to compute the after state ($buf'$).

### 3.5 Parametrised non-blocking buffer

The fifth specification\(^7\) is built from the first by using parametrisation and possible values. The intention is to highlight that parametrisation with possible values can rule out the LazyProducer and the LazyConsumer without requiring the introduction of blocking mechanisms in the buffer.

A syntactic convention ($\triangleq$) is proposed for parametrisation to shorten the writing of specifications based on possible values. In the following expressions $a$ is the after state, $b$ is the set of possible values for the before state, $\text{unary}_\text{op}$ is any unary operator, $\text{bin}_\text{op}$ is any binary operator and $v$ is the right-hand side parameter of $\text{bin}_\text{op}$. The convention applies only if the rely can be inferred from the context (here the rely is the same given as the parameter to the post-conditions, i.e., $r$):

\(^7\) Filename: sequence/Sec_35_ParametrisedNonBlockingBuffer.thy.
Designing an unbounded buffer in rely-guarantee

\[(a \doteq b) \equiv a \in (pv \ b \ r)\]
\[(a \doteq \text{unary}\_\text{op} \ b) \equiv a \in \text{lift}\_\text{pv} (\text{lift}\_\text{unary}\_\text{op} (pv \ b \ r)) \ r\]
\[(a \doteq b \ \text{bin}\_\text{op} \ v) \equiv a \in \text{lift}\_\text{pv} (\text{lift}\_\text{bin}\_\text{op} (pv \ b \ r) \ v) \ r\]

Using this convention the specification can be presented as:

\[CONS() \ res: T\]
\[\text{pre} \neg \text{is}\_\text{empty} \ buf\]
\[\text{rely} \ buf \ \text{prefix} \ buf'\]
\[\text{guar} \ buf' \ \text{suffix} \ buf\]
\[\text{post} \ r \equiv res = \text{hd} \ buf \land buf' \doteq \text{tl} \ buf\]

\[PROD(e: T)\]
\[\text{pre true}\]
\[\text{rely} \ buf' \ \text{suffix} \ buf\]
\[\text{guar} \ buf \ \text{prefix} \ buf'\]
\[\text{post} \ r \equiv buf' \doteq buf \ \cap^\sim [e]\]

The actual values used to instantiate \(r\) are either the rely of the process, or the identity relation (\(ID\)). Outside the parametrised feasibility PO, the parameter \(r\) in \(post_{CONS}\) (or \(post_{PROD}\)) always denote the same as \(\text{rely}_{CONS}\) (\(\text{rely}_{PROD}\), respectively). The postconditions of this section are similar to the those from the sequential specification of [DFJ14, §4.1], which are transcribed below.

\[CONS_{SEQ}() \ res: T\]
\[\text{pre} \neg \text{is}\_\text{empty} \ buf\]
\[\text{post} \ res = \text{hd} \ buf \land buf' = \text{tl} \ buf\]

\[PROD_{SEQ}(e: T)\]
\[\text{pre true}\]
\[\text{post} \ buf' = buf \ \cap^\sim [e]\]

The similarity supports the claim that possible values reduces the gap between concurrent and sequential specifications of the same process by being an adequate abstraction. This specification passes in all proof obligations and the sanity checking designed to rule out the spurious implementations discussed.

3.6 Discussion

We extended postconditions with a new parameter to make explicit which relation they should be stable under. This extension targeted the recasting of the feasibility PO used as sanity check in Sections 3.1-3.5.

To improve the usability of possible values, we proposed a convention (\(\doteq\)) to hide the mathematics behind possible values from a specification. The convention can be applied whenever the rely can be inferred from the context. Using
this convention, pre and postconditions become similar to the those presented in the sequential specification of [DFJ14, §4.1]. This observation supports the claim that possible values reduces the gap between concurrent and sequential specifications. The proposed definition for possible values gives a step towards the separation of concerns between the components of a specification with an appropriate abstraction. It also eliminates the direct encoding of the interference within postconditions.

For posterior uses of possible values, we introduce a new parameter to it: the selector of the variable it refers to. When records are used to model the state, field selectors suffice for this purpose. This allow us to build the set of reachable values rather than reachable states. So far, the usage of possible values was simplified by the fact that in this section the state is not a record. Further mathematical abstractions would be required for lifting records, yet we do not think this to be a problem, given they are trivial, if not automatically inferable from conventions in specifications.

It is worth noting that parametrisation without possible values could solve the problem of ruling out spurious implementations. This, however, requires us to refrain from directly codifying the rely in the postconditions (e.g., let \( \text{post}_{\text{PROD}} \) be \( r(\text{buf} \, \text{↷} \, [e]) \, \text{buf}' \) and \( \text{post}_{\text{CONS}} \) be \( \text{res} = \text{hd} \, \text{buf} \, \land \, r(\text{tl} \, \text{buf}) \, \text{buf}' \)). This is because the PO used as sanity check is defined independently of possible values. The other way round is not true: possible values without parametrisation cannot sort the problem of ruling out spurious implementations. This is because the use of possible values \( \text{per si} \) does not increase the expressiveness of rely-guarantee: it is just an abbreviation for a set build from a state and a rely condition.

4 State as sequence with counter

In this section a sequence and a counter (i.e., an auxiliary variable) are used to represent the state of a concurrent buffer with a producer-consumer pair operating on it. The intention is to show that the spurious implementations from Section 3 can be ruled out by introducing auxiliary variables instead of extending rely-guarantee. Two specifications are designed: the first one records the number and history of consumed elements, whereas the second records the number of consumed elements without keeping their history. Again, all proof obligations were discharged using Isabelle/HOL.

Both specifications are biased in the sense of [Jon77,Jon90], i.e., they contain unnecessary data that cannot be distinguished by the processes acting on the buffer. The unnecessary data \( \text{bias} \) are the history and the number of consumed elements. These data cannot be thrown away by implementations, because they are needed to define a retrieve function to the abstract specification. Thus, the \( \text{bias} \) at the specification level needs to be carried over into the implementations.

The motivation for this part of our investigation is to show the common technique of using auxiliary (ghost) specification variables and its consequence, namely increased bias, with the benefit of ruling out undesired implementations. Arguably, that is because we are bringing to the specification aspects that belong...
Designing an unbounded buffer in rely-guarantee

...to the implementations. That serves to illustrate the trade-off when choosing an appropriate specification paradigm and also to motivate the need for compositional abstraction mechanisms as much as possible, instead of ‘specification tricks’ that tend to not scale well and clutter specification clarity.

Specifications in this Section could be proved as data refinements of previous specifications in Section 3.

4.1 History preserving buffer

In this specification a buffer is a record with two components: \( \text{buf} \) and \( \text{consumed} \). The component \( \text{buf} \) is a sequence of the type \( T \), and \( \text{consumed} \) is the number of elements already read by the consumer process. The state invariant (\( \text{inv-Buffer} \)) establishes that the length of \( \text{buf} \) is greater or equal to \( \text{consumed} \). This reflects the fact that elements are not removed from the buffer after consumption, and that consumption is merely done by incrementing the \( \text{consumed} \) component. In this specification each process touches a different part of the state\(^8\).

\[
\text{Buffer} :: \quad \begin{align*}
\text{buf} &: T^* \\
\text{consumed} &: \mathbb{N}
\end{align*}
\]

where

\[
\text{inv-Buffer} : \text{Buffer} \rightarrow \mathbb{B}
\]

\[
\text{inv-Buffer}(\text{mk-Buffer}(\text{buf}, \text{consumed})) \triangleq \text{consumed} \leq \text{len buf}
\]

An empty buffer is a record where the value stored in \( \text{consumed} \) is greater or equal to the length of \( \text{buf} \). From the state invariant (\( \text{inv-Buffer} \)) we conclude that \( \text{consumed} \) cannot be greater than the length of \( \text{buf} \).

\[
\text{is_empty} : \text{Buffer} \rightarrow \mathbb{B}
\]

\[
\text{is_empty}(\text{mk-Buffer}(\text{buf}, \text{consumed})) \triangleq \text{consumed} \geq \text{len buf}
\]

The head of a buffer is the first element of \( \text{buf} \) which has not been consumed yet. This corresponds to skipping the first \( \text{consumed} \) elements and getting the next one. The partial function \( \text{buffer.hd} \) is only defined for non-empty buffers.

\[
\text{buffer.hd} \ (\text{mk-Buffer}(\text{buf}, \text{consumed}): \text{Buffer}) \ r : T
\]

\[
\text{pre} \leftarrow \text{is_empty \ mk-Buffer(\text{buf}, \text{consumed})}
\]

\[
\text{post} \ r = \text{buf (consumed + 1)}
\]

\(^8\) Filename: sequence-and-counter/Sec_41_HistoryPreservingBuffer.thy.
The part of the buffer which has not been consumed yet is obtained using the function `consumable`. This function returns a buffer without history, where all elements are available for consumption.

\[
\text{consumable} : \text{Buffer} \rightarrow \text{Buffer} \\
\text{consumable} (\text{mk-Buffer}(\text{buf}, \text{consumed})) \triangleq \text{mk-Buffer} (\text{drop} \ \text{consumed} \ \text{buf}, 0)
\]

In the definition above, `drop` is an inductive function over sequences. It takes two parameters: a natural number \(n\) and a list \(l\), and return the suffix of \(l\) obtained by skipping the first \(n\) elements of \(l\), e.g. \(\text{drop} \ 2 \ [a, b, c, d, e] = [c, d, e]\). If the list \(l\) has less than \(n\) elements, \(\text{drop}\) returns the empty list.

\[
\text{drop} : \mathbb{N} \rightarrow T^{*} \rightarrow T^{*} \\
\text{drop}(n, []) \triangleq [] \\
\text{drop}(n, \text{cons}(x, xs)) \triangleq \text{cases} \ n \ \text{of} \\
\quad 0 \rightarrow \text{cons}(x, xs) \\
\quad \text{succ}(n) \rightarrow \text{drop} \ n \ xs \\
\text{end}
\]

We use `consumable` to define suffix and prefix for the buffer.

\[
\text{is\_prefix} : \text{Buffer} \rightarrow \text{Buffer} \rightarrow \mathbb{B} \\
\text{is\_prefix}(s, t) \triangleq \exists\ us \cdot (\text{consumable} \ s).\text{buf} \sqsubseteq us = (\text{consumable} \ t).\text{buf}
\]

\[
\text{is\_suffix} : \text{Buffer} \rightarrow \text{Buffer} \rightarrow \mathbb{B} \\
\text{is\_suffix}(s, t) \triangleq \exists\ us \cdot us \sqsupset (\text{consumable} \ s).\text{buf} = (\text{consumable} \ t).\text{buf}
\]

The producer and consumer’s specification are straightforward. Each process writes on a different part of the state: the consumer is the only process to update the `consumed` component, whereas the producer is the only process to write on `buf`. This means that the consumer does not remove elements from `buf`, and the history of consumed elements is preserved.

\[
\text{CONS} () \ \text{res} : T \\
\text{pre} \sim\text{is\_empty}(b) \\
\text{rely} \ \text{is\_prefix} (b, b') \land \\
\quad b'.\text{consumed} = b.\text{consumed} \\
\text{guar} b'.\text{buf} = b.\text{buf} \land \\
\quad b'.\text{consumed} \geq b.\text{consumed} \\
\text{post} \ \text{res} = \text{buffer\_hd} \ b \land \\
\quad b'.\text{consumed} = b.\text{consumed} + 1
\]
PROD \( (e : T) \)

\begin{align*}
\text{pre} & \quad \text{true} \\
\text{rely} & \quad b'.buf = b.buf \\
\text{guar} & \quad \text{is_prefix} \ (b, b') \land \\
& \quad b'.\text{consumed} = b.\text{consumed} \\
\text{post} & \quad b'.buf = b.buf \overset{\rightarrow}{=} [e]
\end{align*}

Both processes are apparently underspecified, i.e., they do not define the whole final state in their postconditions, but their guarantees state that those underspecified components in their postconditions are not allowed to be modified by the them. The reason for apparent underspecification is stability, e.g. adding \( b'.\text{consumed} = b.\text{consumed} \) to \( \text{post}_{\text{PROD}} \) would make \( \text{Sta}_{\text{RelyPost}} \) and \( \text{Sta}_{\text{PostRely}} \) to become unprovable. We noticed that possible values can be used to make an apparent underspecified process to become fully specified. The steps of such transformation are discussed in Section 5.10.

In the specification above, part of the consumer’s guarantee is not included in the producer’s rely. This means that \( b'.\text{consumed} \geq b.\text{consumed} \) is not relevant to the producer process. If we read the antecedent and consequent of the coherence checking (see Coh.\_GuarRely PO Section 2.2) as “saying” and “understanding”, respectively, we see that processes under specification need to reach consensus between what is “said” by the environment and what needs to be “understood” by each of the parts.

<table>
<thead>
<tr>
<th>Proof obligation</th>
<th>Status</th>
</tr>
</thead>
<tbody>
<tr>
<td>Closure checking</td>
<td>✓</td>
</tr>
<tr>
<td>Coherence checking</td>
<td>✓</td>
</tr>
<tr>
<td>Stability checking</td>
<td>✓</td>
</tr>
<tr>
<td>Feasibility checking</td>
<td>✓</td>
</tr>
</tbody>
</table>

Table 6: Summary of proof obligations.

Table 6 summarises the theorems proved for this specification. Because the postconditions in this specification use equalities, they are deterministic w.r.t. to the processes footprint, i.e., the part of the state touched by the processes. Thus, there is no need for checking the specification against spurious implementations, as we did in the previous section. This is because the reason for sanity checking is to identify if a specification allows spurious ways of solve non-determinism w.r.t. the processes footprint.

4.2 Buffer without history

Here the state is a record, where \( \text{buf} \) is a sequence of the type \( T \), and the component \( \text{consumed} \) stores the number of elements already read by the consumer
There is no state invariant relating these components (i.e., we omit \texttt{inv-Buffer} as it is always \texttt{true}), but an \textit{implicit invariant} was discovered \textit{a posteriori}. Implicit invariants are two-state predicates that guard all transitions in a program, but that are not intentionally defined by the user. Their discovery is not essential to complete the proofs, but can provide intuition about the specification. In this case, the intuition behind the implicit invariant is that the producer can insert elements in the buffer at any time, but the consumer cannot remove elements from \texttt{buf} unless it increments \texttt{consumed} previously or simultaneously. Section 5.4 discusses more about implicit invariants.

\begin{verbatim}
Buffer : : buf: T* 
  consumed: N

buffer_len : Buffer \rightarrow N
buffer_len(mk-Buffer(buf, c)) \triangleq \text{len } buf

impInv : Buffer \rightarrow Buffer \rightarrow \mathbb{B}
impInv(b, b') \triangleq buffer_len(b') \geq buffer_len(b) - (b'.consumed - b.consumed)
\end{verbatim}

The implicit invariant above (\texttt{impInv}) is a weaker version of \texttt{buffer_len(b')} = \texttt{buffer_len(b) - (b'.consumed - b.consumed)}, which is part of the consumer’s guarantee. The consumer’s guarantee states that whenever \texttt{consumed} and \texttt{buf} are updated, the length of \texttt{buf} reduces by the exactly number of consumed elements (\texttt{\Delta consumed}). The invariant relaxes the equality to cope with the actions of the producer, which can insert new elements in the buffer while the consumer is running.

Although the data type used in this specification is the same used in Section 4.1, its usage differ from that (which is evidenced by the omission of the state invariant). To make specifications in Section 4 comparable among themselves, we preserve the interface of the operations used in the previous specification. To check if a buffer is empty the operator \texttt{is_empty} is used. An empty buffer has the component \texttt{buf} equals to the empty sequence.

\begin{verbatim}
is_empty : Buffer \rightarrow \mathbb{B}
is_empty(mk-Buffer(buf, consumed)) \triangleq \text{buf} = []
\end{verbatim}

\footnote{Filename: sequence-and-counter/Sec_42_NoHistoryBuffer.thy.}
The operations for the buffer (prefix, suffix, head, tail and concatenation) are defined based solely on the $buf$ component. This means that these definitions are independent of the value of $consumed$, and highlights the fact that $consumed$ is a bias in this model.

\[
is\_prefix : Buffer \to Buffer \to \mathbb{B} \\
is\_prefix(mk\text{-}Buffer(bфа}, x), mk\text{-}Buffer(bфb, y)) \triangleq \text{.bufa prefix bufb}
\]

\[
is\_suffix : Buffer \to Buffer \to \mathbb{B} \\
is\_suffix(mk\text{-}Buffer(bфа}, x), mk\text{-}Buffer(bфb, y)) \triangleq \text{bufa suffix bufb}
\]

\[
\text{buffer} \cdot \text{hd} (b : \text{Buffer}) : T \\
\text{pre} \sim \text{is} \_\text{empty} b \\
\text{post} r = \text{hd} b . buf
\]

\[
\text{buffer} \cdot \text{tl} (mk\text{-}Buffer(bф, consumed) : \text{Buffer}) : \text{Buffer} \\
\text{pre} \sim \text{is} \_\text{empty} mk\text{-}Buffer(bф, consumed) \\
\text{post} r = mk\text{-}Buffer(tl bф, consumed)
\]

\[
\text{buffer} \cdot \text{concat} : Buffer \to T \to Buffer \\
\text{buffer} \cdot \text{concat}(mk\text{-}Buffer(bф, c), e) \triangleq mk\text{-}Buffer(bф ↾ [e], c)
\]

Using these operators the consumer process can be presented as:

\[
\text{CONS} () : \text{res} : T \\
\text{pre} \sim \text{is} \_\text{empty}(b) \\
\text{rely} \ \text{is} \_\text{prefix}(b, b') \land \\
\quad b'.\text{consumed} = b.\text{consumed} \\
\text{guar} \ \text{is} \_\text{suffix}(b', b) \land \\
\quad \text{buffer} \cdot \text{len}(b') = \text{buffer} \cdot \text{len}(b) - (b'.\text{consumed} - b.\text{consumed}) \\
\text{post} \ \text{res} = \text{buffer} \cdot \text{hd} b \land \\
\quad b'.\text{consumed} = b.\text{consumed} + 1 \land \\
\quad \text{is} \_\text{prefix}(\text{buffer} \cdot \text{tl} b, b')
\]

Differently from Section 4.1, the history is not preserved in this specification. The consumer increments $consumed$ by one and removes the head of the buffer.
\textit{PROD} (e; T)
\begin{align*}
&\text{pre true} \\
&\text{rely } is\_\text{suffix}(b', b) \land \\
&\quad\text{buffer}\_\text{len}(b') = \text{buffer}\_\text{len}(b) - (b'.\text{consumed} - b.\text{consumed}) \\
&\text{guar } is\_\text{prefix}(b, b') \land \\
&\quad b'.\text{consumed} = b.\text{consumed} \\
&\text{post } is\_\text{suffix}(b', \text{buffer}\_\text{concat}(b, e)) \land \\
&\quad\text{buffer}\_\text{len}(b') = \text{buffer}\_\text{len}(b) - (b'.\text{consumed} - b.\text{consumed}) + 1
\end{align*}

The producer concatenates a new element to the right of the buffer. When producer and consumer interleave in their execution, the final value of \textit{consumed} can be used to determine if the consumer finished before the producer. Table 7 summarises the theorems proved for this specification.

\begin{table}[h]
\centering
\begin{tabular}{|c|c|}
\hline
\textbf{Proof obligation} & \textbf{Status} \\
\hline
Closure checking & ✓ \\
Coherence checking & ✓ \\
Stability checking & ✓ \\
Feasibility checking & ✓ \\
Sanity checking & ✓ \\
\hline
\end{tabular}
\caption{Summary of proof obligations.}
\end{table}

The use of a auxiliary variable (\textit{consumed}) suffices to rule out spurious implementations such as variations of \textit{LazyConsumer} and \textit{LazyProducer} discussed in Section 3.

4.3 Possible values and buffers without history

This section illustrates the application of possible values to states which are records rather than simple variables. We choose to apply possible values to the buffer without history because the postconditions there still encode part of the rely condition. The specification of this section differs from that of the previous section by the postconditions\textsuperscript{10}.

We cannot reuse the lifted operators for tail and concatenation introduced in Section 3.3 in this specification. This is because every time the state representation changes, we need to update the lifted operators accordingly. The process of building a lifted operator $\textit{lift}\_\text{op}: \mathcal{P}(\text{State}) \to \cdots \to \mathcal{P}(\text{State})$ from a state operator $(\textit{op}: \text{State} \to \cdots \to \text{State})$ can be syntetised into two steps. The first is the identification of the precondition of the state operation. We use the precondition to ensure that lifted operators are total (see discussion in Section 5.9 about partial and total lifted operations). The second step is to write the lifted

\textsuperscript{10} Filename: sequence-and-counter/Sec_43_PosvalsNoHistoryBuffer.thy.
Designing an unbounded buffer in rely-guarantee

operator as \( \text{lift}_{\text{op}}(S, \cdots) = \{ \text{op}(x, \cdots) \mid x \in S \land \text{pre}_{\text{op}}(x, \cdots) \} \). To illustrate the process we lift \( \text{buffer}\_\text{concat} \) and \( \text{buffer}\_\text{tl} \):

\[
\text{lift}\_\text{concat} : \mathcal{P}(\text{Buffer}) \rightarrow T \rightarrow \mathcal{P}(\text{Buffer})
\]

\[
\text{lift}\_\text{concat}(S, e) \triangleq \{ \text{buffer}\_\text{concat}(x, e) \mid x \in S \land \text{true} \}
\]

\[
\text{lift}\_\text{tl} : \mathcal{P}(\text{Buffer}) \rightarrow \mathcal{P}(\text{Buffer})
\]

\[
\text{lift}\_\text{tl}(S) \triangleq \{ \text{buffer}\_\text{tl}(x) \mid x \in S \land \neg \text{is}\_\text{empty}(x) \}
\]

We used \( \text{Buffer} \) instead of \( \text{State} \) to emphasize that the lifted operators are redefined every time the state representation changes. When the state is represented by a record, as is the case, we also include an operator to lift record selectors. These are functions that receive a state (i.e., a data structure) and return a component of it, e.g. \( \text{buf}(\text{mk}\_\text{Buffer}(b, c)) = b \), \( \text{fst}(e, f) = e \), etc.

\[
\text{lift}\_\text{select} : \mathcal{P}(\text{State}) \rightarrow \{ \text{State} \rightarrow T \} \rightarrow \mathcal{P}(T)
\]

\[
\text{lift}\_\text{select}(S, f) \triangleq \{ f(x) \mid x \in S \}
\]

The purpose of \( \text{lift}\_\text{select} \) is to transform a set of states (its first argument) into a set of values. The set of values returned is obtained through the selector given as second argument. We now have all operators we need to write the specification:

\[
\text{CONS}() \quad \text{res: T}
\]

\[
\text{post} \quad r \equiv \text{res} = \text{buffer}\_\text{hd}(b) \land
\quad b'.\text{consumed} = b.\text{consumed} + 1 \land
\quad b'.\text{buf} \in \text{lift}\_\text{select} (\text{lift}\_\text{pv} (\text{lift}\_\text{tl} (\text{pv} b r)) r) \quad \text{buf}
\]

\[
\text{PROD} (e: T)
\]

\[
\text{post} \quad r \equiv b'.\text{buf} \in \text{lift}\_\text{select} (\text{lift}\_\text{pv} (\text{lift}\_\text{concat} (\text{pv} b r) e) r) \quad \text{buf} \land
\quad \text{buffer}\_\text{len}(b') = \text{buffer}\_\text{len}(b) - (b'.\text{consumed} - b.\text{consumed}) + 1
\]

The \textit{modus operandi} behind the lift composition in this specification illustrates our intuition: \( \text{pv} \) abstracts the interference on the initial values, and yields a set containing all possible intermediate states; \( \text{lift}\_\text{tl} \) and \( \text{lift}\_\text{concat} \) are applied over this set, yielding a set of intermediate results (i.e., ‘intermediate’ means that these states are subject to interference from the environment); \( \text{lift}\_\text{pv} \) is used to abstract the interference after the application of \( \text{lift}\_\text{tl} \) and \( \text{lift}\_\text{concat} \); finally, the component of interest (\( \text{buf} \)) is extracted from using \( \text{lift}\_\text{select} \). The spectrum of possibilities for the composition of lift operators is discussed in Section 5.8. Note that we only lifted those state operators that were applied over the set of possible values.

We can also extend the syntactic convention \( \equiv \) to cope with selectors. In the following expressions \( a \) is the after state, \( b \) is the set of possible values for the
before state, $\text{unary\_op}$ is any unary operator, $\text{bin\_op}$ is any binary operator and $v$ is the right-hand side parameter of $\text{bin\_op}$. The convention applies only if the rely and selector can be inferred from the context. In case of records, we assume the selector has the same name of the field it retrieves:

\[
(a \triangleq b.x) \equiv a \in \text{lift\_select} (pv b r) x
\]
\[
(a \triangleq (\text{unary\_op } b).x) \equiv a \in \text{lift\_select} (\text{lift\_pv} (\text{lift\_unary\_op} (pv b r)) r) x
\]
\[
(a \triangleq (b \text{ bin\_op } v).x) \equiv a \in \text{lift\_select} (\text{lift\_pv} (\text{lift\_bin\_op} (pv b r) v) r) x
\]

Using this convention, and taking $\triangleleft$ as an infix version of $\text{buffer\_concat}$, the postconditions can be presented succinctly:

\[
x \triangleleft y \triangleq \text{buffer\_concat}(x, y)
\]

### CONS ($\epsilon$) res: $T$

**post** $r \equiv \text{res} = \text{buffer\_hd}(b) \land b'.\text{consumed} = b.\text{consumed} + 1 \land b'.\text{buf} \triangleq (\text{buffer\_tl} \, b).\text{buf}$

### PROD ($e$: $T$)

**post** $r \equiv b'.\text{buf} \triangleq (\text{buffer\_tl} \, (\epsilon \triangleleft e)).\text{buf} \land \text{buffer\_len}(b') = \text{buffer\_len}(b) - (b'.\text{consumed} - b.\text{consumed}) + 1$

**Aside.** The definition of $\text{lift\_tl}$ used in this specification has a latent issue that went unnoticed in this specification: it is built from a sate operator that does not comply with the guarantee of the process where it is used (i.e., $\text{buffer\_tl}$ removes the head of $\text{buf}$ without increment $\text{consumed}$, although $\text{guar\_CONS}$ states that “$\text{buffer\_len}(b') = \text{buffer\_len}(b) - (b'.\text{consumed} - b.\text{consumed})$”.

This brings no problems to the specification, because the result of $\text{lift\_pv}$ is not directly used to define the final state ($b'$). If however, we had defined **post\_CONS** to be “$\text{res} = \text{buffer\_hd}(b) \land b' \in \text{lift\_pv} (\text{lift\_tl} (pv b r) r)$”, then proof of the parametrised feasibility PO (see RG\_Feasibility$_\text{Par}$ in 5.13) would reveal that the lift operator does a step which is not bounded by the consumer’s guarantee.

### 4.4 Discussion

We introduced an auxiliary variable ($\text{consumed}$) in the specification of the a single producer-consumer to illustrate an alternative approach to eliminate the spurious specifications discussed in the previous section. In the first specification, the whole history of consumed elements was preserved. This allowed us to separate the updates in disjoint parts of the memory, which were manipulated by different processes.

The weakness of the specification in Section 4.1 is that it is biased, i.e., it inflicts the need for keeping the auxiliary variable $\text{consumed}$, and the history of
consumed elements in the implementation. The bias is reduced (but not removed) in Section 4.2. This second specification does not keep the history of consumed elements. We discovered \textit{a posteriori}, in the second specification, the existence of an \textit{implicit invariant}, which restricts the updates of the components \texttt{buf} and \texttt{consumed}.

The last specification (Sec. 4.3) differs from the second (Sec. 4.2) by the use of parametrisation and possible values. So far, the application of possible values has produced three reusable operators: \texttt{lift pv}, \texttt{lift select} and \texttt{pv}. These operators are independent of context, \textit{i.e.}, they do not require adjustments to match the data types involved. On the other hand, the lift operators such as \texttt{lift tl} and \texttt{lift concat} are dependent of the data structure used to model the state. In general, lift operators must not violate the invariants of a specification.

In order to standardise the use of \texttt{lift select}, we could consider the identity function (\textit{i.e.}, $\lambda x. x$) as selector for the specifications in Sections 3.3-3.5. We have not done this for sake of simplicity. Instead, we overloaded the $\doteq$ convention to make the distinction among different data structures used to represent the state.

During mechanisation, we plan to explore the use of Isabelle’s high order unification to find suitable selector abstraction functions.

5 Theoretical investigation

This section discusses all theoretical contributions of this document, and proposes new directions for investigating extensions for rely-guarantee.

5.1 Safety checking

By safety checking we refer to the verification of one-state properties that should never be satisfied by intermediate states of an implementation. In the following PO schema we use $P$ to denote such an undesired property:

$$
\forall si \cdot \text{pre}(si) \Rightarrow \\
(\forall sk \cdot sk \neq si \land \text{guar}(si, sk) \land \text{guar}(sk, sf) \land \text{post}(si, sf) \Rightarrow (\neg P sk))
$$

In the mechanisation of the specification contained in Section 3.2 we use $P \texttt{ buf} = \texttt{is empty buf}$ to illustrate the concept of safety properties. We proved that both producer and consumer process are \textit{safe} with respect to $P$. This PO is included in our library of proof obligations (RGPOs.thy).

5.2 State invariant

\textit{State invariants} are one-state predicates ($\text{State} \rightarrow \mathbb{B}$). They establish a property that must be preserved by every state in a system (\textit{e.g. inv-Buffer} in Section 4.1). State invariants are encoded in a specification as follows:

\texttt{pre s: inv s \wedge ...}
\( \text{guar } s \ s' : \text{inv } s \Rightarrow (\text{inv } s' \land ...) \)
\( \text{rely } s \ s' : \text{inv } s \Rightarrow (\text{inv } s' \land ...) \)
\( \text{post } s \ s' : \text{inv } s \land \text{inv } s' \land ... \)

Implication \( (\Rightarrow) \) is used in rely and guarantee conditions instead of conjunction \( (\land) \) to mean that if the before state \( (s) \) violates the state invariant nothing can be enforced upon after state \( (s') \).

### 5.3 Evolutionary invariant

Evolutionary invariant are two-state predicates defined by the user that must be enforced by each of the actions of a system. The literature [Stø91,Mid93,CJ00] has not reached a consensus in the terminology for referring to evolutionary invariant: dynamic invariant, evolution invariant, binary invariant, etc.

The purpose of using evolutionary invariants is to improve the readability of a specification. Instead of repeating the invariant, \( evInv(s, s') \), as a conjunct in all rely and guarantee conditions, the user defines the evolutionary invariant in one place and whoever uses the specification should read relies and guarantees as follows:

\( \text{guar } s \ s' : evInv(s, s') \land ... \)
\( \text{rely } s \ s' : evInv(s, s') \land ... \)

Evolutionary invariants must be reflexive and transitive, otherwise the expanded rely and guarantees would fail the closure checking. After the investigation about evolutionary invariants being conducted, it was discovered that this type of invariant is described in [CJ95,CJ00]. There, the authors use the name evolution invariant. Similarly, our state invariants are called data invariants. We did not use evolutionary invariants in our specifications.

### 5.4 Implicit invariant

Implicit invariants are two-state predicates that guard the transitions between atomic steps in a program, but they are not intentionally defined by the user. They are consequences of the rely and guarantee relations. Implicit invariants can be revealed by proving the following PO:

\[ \text{Imp.Inv: } \forall p, si, sf \cdot (\text{guar}_p \ si \ sf \Rightarrow ev \ si \ sf) \land \]
\[ (\text{rely}_p \ si \ sf \Rightarrow ev \ si \ sf) \land \]
\[ (\text{pre}_p \ si \Rightarrow \text{post}_p \ si \ sf \Rightarrow ev \ si \ sf) \]

where \( p \) is the process identifier, and \( si \) and \( sf \) are the before and after states. Implicit invariants are reflexive. Reflexivity is a consequence of \( \text{Imp.Inv} \), and the POs \( RT_*Guar \) and \( RT_*Rely \) discussed in Section 2.1. The next theorem
Theorem 2 (Implicit Invariant Closure). Implicit invariants are closed over the domain of rely and guarantee relations.

\[ \forall x, y, z \cdot (R x x) \land (R x y \land R y z \Rightarrow R x z), \]
\[ \forall a, b \cdot (R a b \Rightarrow I a b) \]
\[ \vdash \forall x, y, z \cdot (I x x) \land (R x y \land R y z \Rightarrow I x z) \]

In Theorem 2, two assumptions are taken: the first is that relies and guarantees (both denoted by \( R \)) are reflexive and transitive; the second is that any transition guarded by the guarantee or rely is also guarded by the implicit invariant \( I \). The first assumption is discharged by \( RT\_Guar \) and \( RT\_Rely\) POs (see Section 2.1), whereas the second assumption is discharged by the \( Imp\_Inv\) PO. The theorem concludes that implicit invariants are reflexive, and also transitive within the domain of the rely and guarantee relations. Implicit invariants are not required to be transitive outside the the domain of the rely and guarantee relations.

The specification from Sec. 4.2 has \( \text{len} \text{buf}' + \text{consumed}' \geq \text{len} \text{buf} + \text{consumed} \) as an implicit invariant. From the meta-theorem above we know that this implicit invariant is reflexive, and transitive within the domain of the rely and guarantee relations. The PO for implicit invariants is also included in the library of proof obligations (RGPOs.thy)

5.5 Establishing invariant

An establishing invariant is an one-state predicate that continues to hold once it is established. Its establishment is not required to be part of the initialisation of the system. It has the type \( \text{State} \rightarrow \text{B} \), and its definition generates an evolutionary invariant with the shape \( \text{establishingdInv} \ S \Rightarrow \text{establishingdInv} \ S' \). This corresponding evolutionary invariant establishes a causal relation that must be preserved by transitions in a system. Given it has fixed format, we define a ‘lift’ operator \([.]\) to represent it.

\[ [\text{establishingInv}] \equiv (\lambda x \ x'. \text{establishingInv} \ x \Rightarrow \text{establishingInv} \ x') \]

We call the predicate above lifted establishing invariant. It is reflexive and transitive: these properties come as consequence of the shape of this predicate. The specification from Section 3.2 includes a establishing invariant to state that once the buffer becomes non-empty, it never becomes empty again. The state invariant of Section 4.1 (\text{inv-Buffer}) can also be viewed as an establishing invariant; in this case, it starts to hold after initialisation of the system. The lifted
establishing invariant is to be treated as an evolutionary invariant, \textit{i.e.}, the formula \( [\mathit{establishingInv}] (s, s') \) must be encoded as conjunct of rely and guarantee relations.

5.6 State x establishing invariant

No confusion should arise about the choice between state and establishing invariant. State invariants must be part of the initialisation of the system, while establishing invariant do not need to be part of the initialisation. Additionally, nothing can be enforced upon an action that starts from a state that does not satisfy the state invariant, while the lifted establishing invariant reduces to \texttt{true} if the action starts in a state that does not satisfy the establishing invariant.

We emphasize two particular usages and propose a distinction in the names to clarify the different roles involved. In general, one can say that a state invariant is also an establishing invariant, but the other way round is not true.

5.7 Evolutionary x implicit invariant

Evolutionary invariants are used to enforce a property upon transitions between states. This type of invariant is proposed \textit{a priori} or during the specification phase. However, some invariants may be implicitly encoded in the specification without the designer to become aware of them. The discovery of these invariants may provide intuition about a specification.

In general, whenever a user wishes to include a two-state invariant in a specification, evolutionary invariants should be used, as this makes the design decision clear. The discovery of implicit invariants may result from the proof task or hindsight.

5.8 Possible values

Possible values was proposed in [JP11] as a convention to refer to intermediate states in a specification. The concept was devised to fix an earlier flaw in the rely-guarantee specification of Simpson’s 4-slot algorithm [JP08].

In [JP11], the authors specify a \texttt{Read} operation that depends on the value of a variable, namely \( \texttt{fresh-w} \), which can be modified by another process (Write). The flawed version stated that \( \texttt{fresh-w} \) could acquire the initial or final value of \( \texttt{fresh-w} \), however, this misses the case of intermediate updates to this variable that can be used by the operation. Indeed, this is what the authors wished to model. To sort the problem, they create the convention that \( \texttt{fresh-w} \) should denote a set containing “any possible values that can occur during the execution of the operation”. The convention was applied as “\( \texttt{hold-r} \in \texttt{fresh-w} \)” to denote that the value to be used to update \( \texttt{hold-r} \) should be taken from the set of possible values of \( \texttt{fresh-h} \).

Differently from this convention, our definition of possible values (\( \texttt{pv s r} \), defined in Section 3.3) generates a set of states rather than a set of values. The
states in this set may differ from that one passed as parameter (s) by updates made by the environment (r). To transform this set of reachable states into a set of reachable values, we use a function (lift-select, defined in Section 4.3) to select a particular component of the state, by applying a selector to each of the states in the set of reachable states.

To operate over the set of possible values (i.e., pv s r), lifted operations are used. These lifted operations shall have the type \( \mathcal{P}(\text{State}) \to \cdots \to \mathcal{P}(\text{State}) \). Their role is to apply a function (e.g. buffer_tl) to each state contained in the set of states passed as parameter; the result of applied function (e.g. buffer_tl) is another state. Examples of lifted operations were given in Sections 3.3 and 4.3 (e.g. lift_tl and lift_concat).

The result of the application of a lifted operation over possible values is another set of states (i.e., ‘lift_op (pv s r)’ has the type \( \mathcal{P}(\text{State}) \)). We call intermediate result the set generated by the application of a lifted operation: it represents the application of the lifted operation to a set states derived from the s by interference bounded by r. The name ‘intermediate’ refers to the fact that this set of states does not account for interference that may occur a posteriori. To abstract the interfere that happens after the application of the lifted operation, the operator lift_pv (defined in Section 3.3) is used.

Figure 1 shows that the application of possible values can be categorised into 4 stages: stage 1 (a) is the application of pv to a state S; it represents the abstrac-
tion of interference in the before state S. Stage 2 (b) is the application of lifted operations to pv S r and is called intermediate result; the transition (f) in stage 2 means that lifted operations can be composed with their own result; we still call the result of the composition an intermediate result. Stage 3 is the application of lift_pv to the intermediate result; it represents the abstraction of the interference a posteriori. The transition (e) highlights that the composition of lifted operations can be specified as atomic or non-atomic. Following the path \( f^* \to e \), the composition results in an expression like \( \text{lift}_{OP_n} \ldots \text{lift}_{OP_2} \text{lift}_{OP_1} (pv s r) \) which does not allow the environment to interfere between applications. Following the path \( (e \to f \to e)^* \), the composition results in an expression that interpolates a lift_pv between each application of a lift_{OP}. This means that the composition allow the environment to act before and after each lift_{OP} being applied. Stage 4 (d) is optional, and denotes the selection of a component of interest from the set of final states.
To ensure that the composition of lifted operators with possible values operators (\textit{pv}, \textit{lift\_pv} and \textit{lift\_select}) never generates undefined states, we propose a systematic way of lifting state operators. Our approach is discussed in the next section.

### 5.9 Systematic lifting and its implication

We propose a systematic way of lifting operations that ensures the lifted operation (\textit{lift\_op}) to be total even when state operation (\textit{op}) is partial. Our preference by total lifted operators means that the composition of lifted operators with \textit{pv}, \textit{lift\_pv} and \textit{lift\_select} never denotes a set of undefined states or values. To lift a state operation (\textit{op}) to a set of states, we require the identification of its precondition (\textit{pre\_op}) and use a fixed set comprehension format:

\[
\text{lift\_op} : \mathcal{P}(\text{State}) \to \cdots \to \mathcal{P}(\text{State})
\]

\[
\text{lift\_op}(S, \cdots) \triangleq \{ \textit{op}(x, \cdots) \mid x \in S \land \textit{pre\_op}(x, \cdots) \}
\]

This set comprehension denies the application of \textit{op} to states that do not satisfy the precondition of \textit{op}. When \textit{op} is total, \textit{pre\_op}(x, \cdots) can be omitted. If a lifted operator \textit{lift\_op} is applied over a set of states that do not satisfy \textit{pre\_op}(x, \cdots), it returns the empty set. In such case, the composition of this application with possible values operators propagates the empty set, and makes the membership operator (\textit{\in}) in expressions like \(a \in \textit{lift\_pv}(\textit{lift\_op}(\cdots) \cdots)\) to return \textbf{false}. In a postcondition, this turns the specification infeasible.

The problem of allowing partial lifted operators is that their result may denote a set containing undefined states, e.g., \textit{mk\_Buffer}(\textit{tl}[[], 0]). In such case, POs would be necessary to ensure that compositions involved lifted operators never result in a set built from undefined values. At this early stage of our investigation, the most prudent decision is to eliminating the risk of getting to undefined states, thus we advocate that lifted operators should be total.

### 5.10 Apparent underspecification and possible values

Processes that are \textit{apparent underspecified} leave part of the final state underspecified in their postconditions, but their guarantees state that those underspecified
components are not allowed to be modified by them. The reason for writing apparently underspecified processes is stability, i.e., although the processes do not change a component of the state, the environment may be allowed to do it.

An alternative approach to write such specifications is to add to the original postcondition extra restrictions to make explicit that those underspecified components are defined by the environment. We used possible values to apply this approach to the producer’s specification in Section 4.1. Our experiment revealed a subtle relation between invariants and the intuition behind the use of lift\_select.

In the first attempt to complete the producer’s specification\(^{12}\), we added the restriction \(b'.consumed = b.consumed\) to its postcondition. The intention was to abstract the interference over the consumed component. However, attempting to prove \(\text{Sta\_PostRely}\) we noticed that this required \(b'.consumed \leq \text{len}(b.buf)\), i.e., the consumed component of the after state should respect the state invariant (inv-Buffer) w.r.t. the buf component of the before state. This is equivalent to say that the the consumer process could not consume the most recent element added by the producer. This inconsistency showed us that the right state to apply possible values was not \(b\), but \(mk\text{-Buffer}(b'.buf, b.consumed)\). This means that we should abstract the interference from an intermediate state, rather than the before state. In our experiment, the specification differs from that from Section 4.1 by the producer’s postcondition. In following specification snippet, we use the abbreviation \(b_i\) to mean \(mk\text{-Buffer}(b'.buf, b.consumed)\).

\[
\begin{align*}
\text{PROD} \\
\text{post } & b'.buf = b.buf \bowtie [e] \wedge \\
& b'.consumed = b_i.consumed
\end{align*}
\]

From this example, we see that possible values can be used to complete an apparent underspecified process. It is not clear however, what are the benefits of this transformation. In our experiment, the transformation increased the proof effort involved in discharge the POs. Moreover, we noticed that care is necessary to pick the right state to abstract interference.

5.11 Collection of lemmas for possible values

We proved five theorems about possible values: two simplification rules, two theorems about ownership, and a theorem about equivalence\(^{13}\). The first of them (Theorem 3) states that repeated applications of \(\text{lift\_pv}\) can be simplified.

**Theorem 3 (LiftPV-AbsorbsItself).** Accumulative applications of \(\text{lift\_pv}\) to reflexive transitive relies do not extend the set of possible values.

\[
r^* \subseteq r \vdash S, r \cdot \text{lift\_pv} (\text{lift\_pv} S r) r = \text{lift\_pv} S r
\]

\(^{12}\) Filename: Sec\_510.HistoryPreservingBuffer_underspecification.thy

\(^{13}\) Filename: Metatheory.thy.
It is worth noting that relies are required to be reflexive and transitive by \(RT_\cdot RelY\) (see Section 2.1), thus the assumption of Theorem 3 is realistic. Theorem 4 is also a simplification rule: it states that applying \(lift\_pv\) to the result of \(pv\) does not extend the set of possible values. The same assumption about reflexive transitive relies from Theorem 3 is used here.

**Theorem 4 (LiftPV-AbsorbsPV).** *Abstract the interference twice does not extend the set of possible values if the rely condition is reflexive transitive.*

\[
r^* \subseteq r \vdash lift\_pv \ (pv \ s \ r) \ r = pv \ s \ r
\]

The next two theorems are about ownership. In rely-guarantee, saying that a process has ownership over part of a state means that the process has exclusive right to write on this part of the state.

**Theorem 5 (PV-ID).** *The application of possible values to any state \(s\) and ID (i.e., the non-interfering environment) results in a singleton set containing the state \(s\). This is because the use of ID as rely represents the ownership over the whole state by the process. This correspond to the sequential scenario, and in such cases \(v \simeq s \iff v = s\).*

\[
\vdash \forall s \cdot pv \ s \ ID = \{s\}
\]

Theorem 5 states the application of \(pv\) is unnecessary if the environment is the identity relation (ID was defined in Section 3.4). Although this theorem is an obvious consequence of ID, it is useful to illustrate how possible values is tightly linked to its underlying relation, in this case the rely. The next theorem is a result about ownership over part of a state, in contrast with ownership over the whole state.

**Theorem 6 (PV-SelectID).** *If an environment \(R\) is not allowed to change the field \(f\) of a state (retrieved through the function \(f\)), the value of this field is expected to be unchanged in the elements of the set of possible values obtained using \(R\). In this scenario, \(a \simeq \bar{s}.f \iff a = f(s)\).*

\[
\forall x, x' \cdot R x x' \land (R x x' \Rightarrow f x = f x')
\]

\[
\vdash \forall s \cdot lift\_select \ (pv \ s \ R) f = \{f \ s\}
\]

Theorem 6 states that application of possible values to a state \(s\) is unnecessary if we are interested in recover a field \(f\) that is not changed by the rely condition \(R\). Using this theorem we can justify why we did not applied possible values to the specification of Section 4.1: there, rely\(_{PROD}\) is not allowed to change \(buf\), which is the only part of the state updated by the producer process; additionally, rely\(_{CONS}\) is not allowed to change \(consumed\), which is the only part of the state updated by the consumer process. In both cases, the updates do not depend on data which is subject to changes by the environment.

The next theorem suggests that we can get away from \(pv\) using \(lift\_pv\). Although \(lift\_pv\) uses \(pv\) in its definition, this can be easily avoided as shown in Section 6.4.
Theorem 7 (PV-LiftPV-Exchange). The operator \( pv \) is made redundant by \( \text{lift}_{pv} \).

\[ \vdash pv s r = \text{lift}_{pv} \{ s \} r \]

These theorems are formalised in \texttt{Metatheory.thy}, which also contains a result about implicit invariants (Theorem 2).

5.12 Parametrisation

By \textit{parametrisation} we mean the introduction of parameters in components of a specification (e.g. precondition, guarantee, rely and postcondition). The purpose of parametrisation is to generalise a more specific model. In this document, parametrisation refers solely to the introduction of a parameter \( r \) in the postconditions. The parameter \( r \) is a two-state predicate which shall be interpreted as the rely (or environmental interference) assigned to the operation.

The purpose of applying parametrisation to postconditions is to promote a separation of concerns between the postcondition and the rely condition. The POs for rely-guarantee require postconditions to be stable under the rely. In general, this requires part of the rely condition to be encoded in the postcondition. By passing the rely condition as a parameter to the postcondition, we freed the designer from directly encoding part of the rely condition into the postcondition, and allow the automatic extraction of the rely condition from the context. This is an essential step to enable conventions such as possible values (\( \forall \bar{a}r \)) and parametrised equality (\( \equiv \)).

5.13 Parametrised feasibility checking

It is necessary to replace the feasibility PO from Section 2.4 by a new PO when parametrised specifications are used. The new PO requires postconditions to be established without assistance of the environment. This is not a new idea: the same conclusion can be taken from \texttt{Sta_RelyPost} and \texttt{Sta_PostRely}.

This PO appears in the mechanisation of Sections 3.4, 3.5 and 4.3. It was also used as sanity checking to eliminate spurious implementations, such as \textit{LazyProducer} and \textit{LazyConsumer}. The identity rely relation (\( ID \)) is used:

\[
\begin{align*}
\text{RG-Feasibility}_{\text{Par}} : \forall p \cdot \forall s_i, inp \cdot \text{pre}_p(s_i, inp) & \Rightarrow \exists sf, out \cdot \text{guar}_p(s_i, sf) \land \\
& \text{(post}_p \text{ rely}_p)(s_i, sf) \land \\
& \text{(post}_p \text{ ID})(s_i, sf)
\end{align*}
\]

Any pair of initial and final state (\( s_i \) and \( sf \), respectively) that satisfy the postcondition in presence of interference (\( \text{post}_p \text{ rely}_p \)), must also satisfy the postcondition when the process runs in isolation (\( \text{post}_p \text{ ID} \)). Additionally, only steps bounded by the guarantee (\( \text{guar}_p \)) can be used to establish the postcondition.
This version is more comprehensive than the one discussed in [IJH12], which is reproduced in Section 2.4. The reason is that this version not only concerns the precondition, guarantee and postcondition, but also includes the rely condition.

6 Mechanisation issues

This section summarises all mechanisation issues of the work described in this document. The proofs involving possible values were quite easy to be structured. In general, we have not developed lemmas for possible values\(^{14}\), but these will be necessary, given the ability of reuse the concepts discussed in Section 4.4.

Most of the proofs were carried out using the expansion command of Isabelle (unfolding), the high level proof tactics (i.e., auto, simp, rule_tac and safe) and automation (i.e., sledgehammer and metis). Proofs of sanity checking were carried out by stating a conjecture to be falsified by nitpick, a counterexample generator of Isabelle. We had small issues with the mechanisation, which are categorized in the next subsections.

6.1 Subtraction of natural numbers

Isabelle treats \(x - y = 0\), when \(x \in \mathbb{N} \land y \in \mathbb{N} \land x < y\). The axioms for naturals in Isabelle forced us to rephrase the producer’s rely from Section 4.2 into \(\text{len}(b'.\text{buf}) + b'.\text{consumed} = \text{len}(b.\text{buf}) + b.\text{consumed}\) in order to avoid modelling negative results as zero.

6.2 Partial functions

The function buffer\(_{hd}\), used to access the head of the buffer in Section 4.1, is defined in VDM as a partial function, but the mechanised version is a total function. The discrepancy is because partial functions in Isabelle are tricky. The mechanised specification returns undefined whenever buffer\(_{hd}\) is applied outside its precondition:

\[
\begin{align*}
\text{buffer\(_{hd}\) : Buffer} & \rightarrow \text{T} \\
\text{buffer\(_{hd}\)(mk-Buffer(buf, consumed))} & \triangleq \\
& \text{if } (\neg \text{is\_empty mk-Buffer(buf, consumed)}) \\
& \text{then } buf (\text{consumed} + 1) \\
& \text{else } \text{undefined}
\end{align*}
\]

\(^{14}\) Apart from those in Metatheory.thy.
6.3 VDM translation to Isabelle

VDM sequences are indexed from 1. We used Isabelle lists to represent VDM sequences. A consequence of this is that our mechanised specification is indexed from 0. This difference becomes clear in Section 4.1: there, buffer\_hd uses buf \((\text{consumed} + 1)\) to access the first non-consumed element of the buffer, while the mechanisation uses buf \((\text{consumed})\).

6.4 Possible values

Our mechanised version of lift\_pv differs from that presented in this document. As expected, we proved the equivalence between the version presented in Section 3.3 and the mechanised one.

\[
\begin{align*}
\text{lift\_pv}_{\text{Mech}}(S, r) \triangleq \{ \text{sfinal} \mid \exists \text{sint} \in S \land r \text{sint sfinal} \}
\end{align*}
\]

**Theorem 8 (LiftPV-Equivalence).** The mechanised version of lift\_pv is equivalent to the one from Section 3.3.

\[
\vdash \forall S, r \cdot \text{lift\_pv}_{\text{Mech}} S r = \text{lift\_pv} S r
\]

6.5 Proof obligations and dealing with inputs and outputs

Differently from the POs in Section 2.3, the mechanised POs account for the use of input and output in pre and postconditions. Inputs and outputs are not discussed in [CJ07], where most of the POs were taken from. However, in general, postconditions can refer to input and output. Assuming that inputs share the same type, and outputs share the same type, one can model postcondition using the type \(\text{State} \rightarrow \text{State} \rightarrow \text{Inp}^* \rightarrow \text{Out}^* \rightarrow \mathbb{B}\).

The POs in our mechanisation fit exactly the problem that we model. In particular, our postconditions refer to an input or an output, but not to both simultaneously. No restrictions are made over the input. Thus, we use \(\text{State} \rightarrow \mathbb{B}\) to model preconditions, and \(\text{State} \rightarrow \text{IO} \rightarrow \mathbb{B}\) to model postconditions. Rely and guarantee conditions continue to be simple relation over states, i.e., \(\text{State} \rightarrow \text{State} \rightarrow \mathbb{B}\).

The mechanised Sta\_RelyPost\_Mech and Sta\_PostRely\_Mech POs were proved for each process \(p\), where composition is explicit over \(s, s', s''\).

**Sta\_RelyPost\_Mech:**

\[
\forall s, s', s'', \text{io} \cdot \text{pre}_p s \land \text{rely}_p s s' \land \text{post}_p s' s'' \text{ io} \Rightarrow \text{post}_p s s'' \text{ io}
\]

**Sta\_PostRely\_Mech:**

\[
\forall s, s', s'', \text{io} \cdot \text{post}_p s s' \text{ io} \land \text{rely}_p s' s'' \Rightarrow \text{post}_p s s'' \text{ io}
\]
Their respective parametrised versions take the rely as a parameter. The instantiation of the parametrised POs must use \( \textit{rely}_p \) in place of \( r \):

\[
\text{Sta\_RelyPost}_{\text{ParMech}}(r):
\forall s, s', s'', \text{io} \cdot \text{pre}_p s \land r s s' \land \text{post}_p s' s'' r \Rightarrow \text{post}_p s s'' \text{io} r
\]

\[
\text{Sta\_PostRely}_{\text{ParMech}}(r):
\forall s, s', s'', \text{io} \cdot \text{post}_p s s' \text{io} r \land r s s'' \Rightarrow \text{post}_p s s'' \text{io} r
\]

Similarly, our mechanised PO for feasibility and parametrised feasibility make use of the fact that postconditions can take arguments, but only one argument per time. Thus we have two POs, one for the producer process (which has an input variable), and one for the consumer produces (which has an output variable):

\[
\text{Feasible\_Prod}_{\text{Mech}}:\forall s, \text{inp} \cdot \text{pre}_p s \Rightarrow \exists s' \cdot \text{post}_p s s' \text{inp} \land \text{guar}_p s s'
\]

\[
\text{Feasible\_Cons}_{\text{Mech}}:\forall s \cdot \text{pre}_p s \Rightarrow \exists s', \text{out} \cdot \text{post}_p s s' \text{out} \land \text{guar}_p s s'
\]

The corresponding parametrised versions take the rely as parameter. Here, some of the occurrences of the rely parameter are already replaced by \( \text{ID} \). The remaining occurrences of \( r \) must be replaced by \( \textit{rely}_p \):

\[
\text{Feasible\_Prod}_{\text{ParMech}}(r):
\forall s, \text{inp} \cdot \text{pre}_p s \Rightarrow \exists s' \cdot \text{post}_p s s' \text{inp} r \land \text{post}_p s s' \text{inp} \text{ID} \land \text{guar}_p s s'
\]

\[
\text{Feasible\_Cons}_{\text{ParMech}}(r):
\forall s \cdot \text{pre}_p s \Rightarrow \exists s', \text{out} \cdot \text{post}_p s s' \text{out} r \land \text{post}_p s s' \text{out} \text{ID} \land \text{guar}_p s s'
\]

The mechanised POs are available through the library \textit{RGPOs.thy}. The POs not mentioned in this section were mechanised exactly in the same way they are presented in Section 2.

### 6.6 Proof obligations: quantification over processes

The mechanisation of POs from Sections 2.2-2.4, 5.4 and 5.13 do not include the operators \( \forall p \) and \( \land_{x \neq p} \). These operators appear to represent the general case. We tailor made the mechanisation, and achieved the purpose of these quantifiers by instantiating the POs for each of the processes manually.

In case of \( \forall p \cdot P(p) \), we manually instantiated \( P(p) \) for each process \( p \), that in our case, could be the consumer and producer processes. This achieves the same effect, i.e., \( \forall p \in \{\text{CONS, PROD}\} \cdot P(p) = P(\text{CONS}) \land P(\text{PROD}) \).

In case of \( \land_{x \neq p} \cdot P(x) \) we manually flatten this expression into the conjunction of its members \( P(x) \), such that \( x \neq p \). In our case, \( \land_{x \neq p} \cdot P(x) \) turns out to be \( P(\text{PROD}) \) in one case, and \( P(\text{CONS}) \) in the other case.
7 Conclusion and future work

This document illustrates the usage of possible values in specifications. The concept is not strictly necessary to represent the problems discussed, but it helped to reduce the complexity of writing specifications by creating abstractions. Alongside possible values, we extended postconditions to accept an extra parameter: the rely condition to be assigned to them, which enables parametrised interference, as well as a more comprehensive feasibility checking that uses the rely condition.

The introduction of a new parameter in the postconditions is referred as parametrisation. The new parameter allowed us to devise a convention to hide the complexity of the use of possible values. A consequence of the convention is that the postcondition of concurrent and sequential specifications may become more similar. This “gap” reduction was already envisaged in our previous report [DFJ14].

Whenever parametrised specifications were used, we replaced the original feasibility proof obligation by a parametrised version. The new proof obligation highlights the fact that processes in rely-guarantee do not depend on the environment to make progress.

We proved a small collection of lemmas about the concept of possible values using Isabelle/HOL. The application of possible values to different examples produced three operators that can be reused independently of context: $\text{lift}_{\text{pv}}$, $\text{lift}_{\text{select}}$ and $\text{pv}$. We noticed that lifting operators are pervasive in the presence of possible values. Moreover, each application of possible values usually requires the design of lifted operators for the data types and invariants involved. The process of lifting operations however, is systematic.

We also developed other concepts in rely-guarantee: distinct uses of invariants, and proposal of names accordingly. Hopefully, the discussion in Sections 5.6 and 5.7 can be a step towards a naming consensus about invariants. We also proposed a concept of safety property checking, and discuss a PO template that can be used to verify if an implementation violates such safety properties. Next, we indicate future work that complement our investigation.

- **Multiple occurrences of possible values within an expression.** The intention is to explore specifications with multiple occurrences of the possible values operator within an expression, and the link with the set-of-values approach [HBDJ13]. This could prove, for example, that if a variable (e.g. $v$) which is not owned by a process is sampled twice (e.g. say, in instants $i$ and $j$) then the sum of the sampled values (e.g. $v_i + v_j$) can be an odd number, i.e., $v_i = v_j$ can return false. To investigate the link between possible values and set-of-states approach we may need to introduce a notion of time in rely-guarantee.

- **Atomic invariant.** The closure requirement imposed on rely and guarantee relations reduces their usefulness to delimit fine-grained events, such as memory updates. For example, properties like “keep the original value or change
in a particular way” (e.g. \(\lambda x \ x'. x' = x \lor x' = x + 1\)) cannot be stated as rely or guarantee conditions because of the transitivity requirement that these relations need to satisfy (see POs \(RT\_Rely\) and \(RT\_Guar\)). Moreover, such properties do not fit as evolutionary invariants. This suggests that a new type of invariant (or complement to the rely and guarantee conditions) could be useful. Such properties could be enforced for atomic transitions. An alternative approach to the creation of a new concept is to sort this expressiveness weakness by removing the closure checking from the POs, and using the reflexive-transitive closure of relies (i.e., \(r^*\)) and guarantees (i.e., \(g^*\)). In [IJH12], authors allow the use of relations that are not reflexive neither transitive, and apply the closure operator for obtaining the correspondent reflexive-transitive relations. In [Nie02], rely and guarantees are required to be reflexive, but not transitive.

- Cooperating processes. In rely-guarantee we cannot express mutual progression of processes. This is because a process has to establish its postcondition without any assistance of the environment. An interesting problem is to investigate what needs to be changed in the formalism to overcome this limitation.

- Study of stability and persistence. Intuitively, possible values abstracts the interference in the initial and final states. This suggests that postconditions defined solely via possible values and parametrisation are stable by construction. From this perspective, it is worth investigating if we can get away from \(Sta\_PostRely\) and \(Sta\_RelyPost\) when using parametrisation and possible values. Additionally, a notion of persistence should be pursued. Persistence means the preservation of changes made in a state by a process that has completed its execution. We suspect that stability and persistence are different concepts, and the investigation of persistence can help the discussion about cooperating process in rely-guarantee, i.e., process that are required to complete a task together.

- Intermediate properties. Following the study of stability and persistence, we can conceive a program to establish intermediate properties (or change the state in a particular way) without requiring the properties or changes to persist. Thus, we propose the creation of an example for this scenario, and the investigation of the expressiveness of this concept. Note that this idea is somehow similar to the verification of safety properties: there, we want to ensure that a property is never satisfied by intermediate states, here, we want to say that we only accept an implementation if it makes a particular property to be true in some point of the execution.

- Varying the rely parameter in postconditions. In our view, the parameter in the postcondition should denote the rely of the process, except in the parametrised feasibility theorem. So far, we have not experimented varying the rely parameter in postconditions and see the effect in the be-
haviour of a specification, neither if it would make sense to use a rely that is different from the process’ rely. Experimentations in this direction could drive a method to minimise the assumptions a process has to make about the environment.

- **Proof obligations for framing convention.** To deal with framing notation, new POs or, alternatively, encoding patterns would be necessary. In case of encoding patterns, these would be similar to those used for encoding evolutionary invariants. We have sketched some potential POs, but we decided not to use frames in the specifications of this document. The sketched POs would apply if one decides to use frames, but not encode them into rely and guarantee conditions. In such case, they would be simple redundant specification mechanisms, and we could write a PO for checking contradictions. Assuming that each process would have a set of variables that it has permission to write, and the set of all variables in a specification is $\text{VAR}$, these POs would be:

\[
\begin{align*}
\text{FPO1:} & \quad \text{rely}_p(S1, S2) \implies (\forall w \in \text{VARS} \cdot w \in \text{writeVars}_p \implies S1.w = S2.w) \\
\text{FPO2:} & \quad \text{guar}_p(S1, S2) \implies (\forall w \in \text{VARS} \cdot w \notin \text{writeVars}_p \implies S1.w = S2.w)
\end{align*}
\]

- **Variations of the producer-consumer.** This aims to investigate the extension of the specification paradigm to multiple consumers and producers, and producers that write more than an element in the buffer at a time (i.e., to a family of cooperating processes). The purpose is to analyse specifications which include several instances of possible values, e.g., $a_1 \cdot \leftarrow [i] \land a_2 \cdot \leftarrow [j] \land a_3 \cdot \leftarrow [k]$ denotes a producer that insert the string “lift” in the buffer without recur to locking mechanisms. The specifications could be used to illustrate the atomic and non-atomic composition of lifted operations discussed in Section 5.8.

- **Mechanisation improvements.** We aim to use locales of Isabelle to encapsulate and enforce proof obligation consistency for rely-guarantee and also parametrised rely-guarantee specifications. The purpose is to provide a systematic way of generating proof obligations for specifications, and freed the designer from the need of instantiation. As discussed in Section 2.5, Isabelle does yet not alert us if we do not instantiate all proof obligations or instantiate them using wrong parameters. The work proposed in this extension takes the responsibility of the designer and transfers it to Isabelle. Additionally, we expect to investigate if there is any benefit of representing partial functions using different approaches than the one we used.

- **Invariant preservation for possible values.** If the state $s$ provided to $p_v$ respects the state invariant of a specification, then the set of possible values ($p_v \ s \ r$) only includes states that preserve the invariants of a specification. However, when lifted operations are used, these might violate some invariants and generate a set of states that do not comply with the invariants.
We however could not find an example to illustrate this scenario, but more experimentation with possible values shall elucidate if this case can actually happen. If propose the investigation of a PO to be assigned to lifted operations that are not created systematically (see Section 5.9).

- **Relation of safety properties and linear temporal logic.** Usually, safety properties are expressed using temporal logic operators (*i.e.*, *always* and *eventually*), but we managed to express one of such properties using first order logic (see Section 5.1). An interesting investigation is to explore the range of temporal properties that can be expressed in rely-guarantee without introducing temporal logic operators.

- **Investigate refinement laws for possible values.** We believe that once we have explored the concept of possible values over a enough number of examples, the next step is to investigate refinement laws, and how they could relate to the semantics used in [LIH12]. We also expect to investigate the refinement of the spurious implementations which is discussed in the examples of this document via feasibility, and if the POs for rely-guarantee need to be checked at each level of refinement.

- **Possible values in guarantee conditions.** Accordingly to our view, atomic actions of a process must be bounded by the guarantee, and during their execution the environment is not allowed to interfere. In [JP11], the authors use possible values within a guarantee condition. The use of our formulation of possible values within guarantee conditions would mean that the separation between program steps and environment steps stops holding. Thus, we plan to investigate if our formulation of possible values captures the intention of [JP11].

- **Specification case studies.** A potential case study for possible values can be found in [CJ95]. The specification there revolves around the problem of computing equivalence classes. The adaptation of *P-TEST* to use possible values is trivial, but the adaptation of *P-EQUATE* may require more thinking. Other potential case study to possible values is a rely-guarantee specification of *DOM-tree* operations which is currently being produced by colleagues in our department. We believe that possible values and parameterisation can be used there to make the concurrent and sequential specifications to look more similar. Additionally, we consider to explore the use of possible values in preconditions as well.

**Acknowledgement**

We thank to Cliff Jones for having suggested the buffer example from [OHe07] as a case study for possible values, and for all the support provided. The first author thanks Leo Freitas for the discussions about modelling, writing style and the assistance in the mechanisation of the specifications discussed. We also
Designing an unbounded buffer in rely-guarantee

thanks to colleagues of our department for providing us access to a preliminary version of their technical report.

References


