COMPUTING SCIENCE

Abstracting Interference in Postconditions

Diego Dias, Leo Freitas and Cliff Jones

TECHNICAL REPORT SERIES

No. CS-TR-1415 March 2014

TECHNICAL REPORT SERIES

No. CS-TR-1415 March, 2014

Abstracting Interference in Postconditions

D. Dias, L. Freitas and C. Jones

Abstract

Specification of concurrent processes in rely-guarantee may require a postcondition of a process to account for changes made by the environment on the shared state. This leads to complicate postconditions, and distracts the designer from specifying the changes the process should make on the program state. We found that, when used in postconditions, the notion of possible values shifts the designer's perspective from a global view of the parallelism to a local view of it. This enhances the separation of concerns between the rely and the postcondition and may reduce the gap between a sequential and a concurrent version of the same process. In view of this finding, this document is concerned with a preliminary investigation of a semantics for possible values, and the consequence on the proof obligations.

Printed and published by Newcastle University,
Computing Science, Claremont Tower, Claremont Road,
Newcastle upon Tyne, NE1 7RU, England.

Bibliographical details

DIAS, D., FREITAS, L., JONES, C.

Abstracting Interference in Postconditions
[By] D. Dias, L. Freitas and C. Jones
Newcastle upon Tyne: Newcastle University: Computing Science, 2014.

(Newcastle University, Computing Science, Technical Report Series, No. CS-TR-1415)

Added entries

NEWCASTLE UNIVERSITY
Computing Science. Technical Report Series. CS-TR-1415

Abstract

Specification of concurrent processes in rely-guarantee may require a postcondition of a process to account for changes made by the environment on the shared state. This leads to complicate postconditions, and distracts the designer from specifying the changes the process should make on the program state. We found that, when used in postconditions, the notion of possible values shifts the designer's perspective from a global view of the parallelism to a local view of it. This enhances the separation of concerns between the rely and the postcondition and may reduce the gap between a sequential and a concurrent version of the same process. In view of this finding, this document is concerned with a preliminary investigation of a semantics for possible values, and the consequence on the proof obligations.

About the authors

Diego Machado 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).

Cliff B. Jones is currently Professor of Computing Science at Newcastle University. As well as his academic career, Cliff has spent over 20 years in industry. His 15 years in IBM saw among other things the creation –with colleagues in Vienna– of VDM which is one of the better known “formal methods”. Under Tony Hoare, Cliff wrote his doctoral thesis in two years. From Oxford, he moved directly to a chair at Manchester University where he built a world-class Formal Methods group which –among other projects– was the academic lead in the largest Software Engineering project funded by the Alvey programme (IPSE 2.5 created the “mural” (Formal Method) Support Systems theorem proving assistant). He is now applying research on formal methods to wider issues of dependability. He is PI of two EPSRC-funded responsive mode projects “AI4FM” and "Taming Concurrency", CI on the Platform Grant TrAmS-2 and also leads the ICT research in the ITRC Program Grant. Cliff is a Fellow of the Royal Academy of Engineering (FREng), ACM, BCS, and IET. He has been a member of IFIP Working Group 2.3 (Programming Methodology) since 1973.

Suggested keywords

INTERFERENCE
POSSIBLE VALUES
RELY-GUARANTEE CONDITIONS

Abstracting interference in postconditions

Diego Dias, Leo Freitas, Cliff Jones

School of Computing Science
Newcastle University
NE1 7RU, UK
{d.machado-dias,leo.freitas,cliff.jones}@ncl.ac.uk

Abstract. Specification of concurrent processes in rely-guarantee may require a postcondition of a process to account for changes made by the environment on the shared state. This leads to complicate postconditions, and distracts the designer from specifying the changes the process should make on the program state. We found that, when used in postconditions, the notion of *possible values* shifts the designer's perspective from a global view of the parallelism to a local view of it. This enhances the separation of concerns between the rely and the postcondition and may reduce the gap between a sequential and a concurrent version of the the same process. In view of this findings, this document is concerned with a preliminary investigation of a semantics for possible values, and the consequence on the proof obligations.

Keywords: interference, possible values, rely-guarantee conditions

1 Introduction

The issues around concurrency have long challenged researchers to develop tools to aid the design, understanding and verification of concurrent programs. In this context, Cliff Jones formulated in 80s *rely-guarantee* [1], a compositional program logic to reason about interference boundaries. Rely-guarantee records interference as a pair of relations called rely and guarantee: the *rely* records assumptions on the interference the developer is invited to take about the environment where the process will run, whereas the *guarantee* restricts the ways a process can change the state to implement its specification. This program logic inspired works like as RGSep [2], and has been used to give semantics for refinement calculus theories [3, 4].

A process specification in rely-guarantee is a tuple of four components: precondition, rely, guarantee, and postcondition. Preconditions and postconditions come from Hoare logic [5]. The *precondition* restricts the initial state and input variables a process must deal with. If the initial state and the input variables satisfy the precondition, then the *postcondition* must be true of the final state. For example, for a process Q whose purpose is to calculate the square root of a real number x, the precondition could be that $x \geq 0$ and the postcondition could be $x' = \sqrt{x}$. Notice that the precondition is a predicate, whereas the postcondition is a relation on the initial state (x) and the final one (x').

Ideally, preconditions and postconditions should provide a local description of the process, without concern about the interference caused by the environment. In practice the postcondition may need to explicitly mention the effect of the environment to be consistent. For example, if a process P that inserts elements to the end of a *queue* is running in parallel with a process C, which is removing the elements from the front of this *queue*, then the concurrent behaviour of P cannot be described solely as the insertion of elements in the end of this *queue*. This is because during the insertion some elements from the front of the *queue* may be removed by the environment. Thus, the sequential specification is not enough to describe the process in a concurrent scenario.

The example in the previous paragraph illustrates an issue that may happen in specifications using rely-guarantee. The problem is that having postconditions as a relation between the initial and final states shifts the designer's perspective from a local view to a global view of the parallelism. Consequently, it is necessary to be aware of the rely in order to write the postcondition. Also, as there is no syntactic operator to refer to the intermediate state of shared variables in original rely-guarantee, existential quantification over shared variables may be required to refer to intermediate states. This solution creates ambiguity and leaves decisions about the behaviour of processes on the implementor's hand. Consequently, the implemented code may not work as expected by the designer.

The necessity for expressing intermediate states in the postcondition was already noticed in [6]. There, the authors propose a convention named *possible values*: it represents all possible values that a variable can acquire during the process computation. That definition, however, is imprecise and lacks investigation of a semantics. This report reuses the buffer example from [7, 8] as case

study to compare different semantics in order to attempt a formalisation of *possible values*. So far, we envisage that the main outcome of this research will be to establish a separation of concerns among the rely, the guarantee, precondition and postcondition, and to reduce the gap between the sequential and concurrent versions of the the same process. As a consequence of this gap reduction, the specification of concurrent process in rely guarantee can become more abstract. It is also our purpose to investigate the effect of the concept of possible values in rely-guarantee refinement calculus [3], however this first report limits to attempt to formalise the concept of possible values.

Section 2 discusses different approaches to concurrency, and why the notion of *possible values* is not required to define postconditions in these approaches. Section 3 looks more carefully to the 4-tuple version of rely-guarantee, and presents the proof obligations required to check if a specification in this logic is consistent. In Section 4 a case study is presented to illustrate the usage of possible values in rely-guarantee and to argue our claiming that this concept can improve the separation of concerns between the rely and the postcondition. Next, Section 5 discusses a semantics for possible values and the impact on the proof obligations. Section 6 points out ongoing research and preliminary conclusions.

2 Characterising concurrent postconditions

Proving that concurrent algorithms behave as expected is far from being trivial. The difficulty arises because interleaving opens the way for intricate behaviours such as *data races*, *interference* and *non-determinism*. Concurrent program logics ground their correctness criteria on constraining the behaviours the programs can exhibit, as in [9–11, 7, 2]. When looking at specifics programming logics, the reader might find interesting that it is omnipresent a contract between a precondition and a postcondition.

In visiting concurrent program logics we discuss which restrictions are made over the programs that these logics can deal with. Our attention is focused in rely-guarantee and concurrent separation logic, as they illustrate well interference reasoning and mechanisms to show interference avoidance. The intention is to convince the reader that when interference and data races are supported by the logic, the notion of possible values can become an useful abstraction for the designer. At the end of this Section we also introduce concepts that the reader might not be familiar with.

2.1 Towards a Theory of Parallel Programming (TTPP)

A pioneer attempt to reason about concurrent programs was carried out by Hoare in the seventies [9]. There, he introduces the concept of *conditional critical regions* (CCR), an abstraction of mutual exclusion that can be implemented by concurrent programming languages. That abstraction allows to a compiler to enforce serialisation in the access to shared resources and to ensure absence of deadlocks. Hoare proposes two statements to implement CCR:

{***resource*** $r_1, ..., r_n$; $P_1 \parallel ... \parallel P_n$} and ***with*** **r** ***when*** **b** ***do*** **c**. The first of them assigns a semaphore to each of the shared resources in the declaration part ($r_1, ..., r_n$) and defines a parallel composition of processes P_i, which may access any of the declared resources. The second of them confines the access of a resource (r) to a critical region controlled by its respective semaphore[1]. A process can enter in a critical region only when the resource (**r**) is available and the current state satisfies a given predicate (**b**). In such case, the process is granted exclusive use the resource controlled by the critical region. Notice that this exclusive access rules out the need of a concept such as possible values, because no interference on the shared resources is allowed while a process is inside the critical region associated to that resource.

The TTPP paper extends Hoare logic [5] to support programs with CCR and parallel composition. In the extended axiomatic approach, the reasoning revolves around a logically defined resource invariant, and rules for critical regions and parallel composition with side conditions that ensure a discipline in the access to shared resources. Additionally, it discusses a strategy to reduce the granularity of access to arrays. The strategy is based on an interesting concept of *remapping*, which assigns different names for different positions in a array. Thus, it is possible to different pieces of the array to be manipulated simultaneously. Remapping solves two problems at the same time: i) it allows to disjoint parts of the same data structure to appear into two different critical regions (which may overlap in their execution), and overcomes the global nature of axiom assignment in Hoare logic [9, §3].

A proof that a program satisfies its specification, i.e. $\{P\}C\{Q\}$, using this axiomatic system ensures to the programmer that whenever the program is executed starting in a state that satisfies its precondition, if the program terminates, it does it in a state satisfying its postcondition. Additionally the programmer is given a guarantee of absence of data races on the shared resources.

This axiomatic approach is very neat and motivates the design of tractable programming languages by including abstractions to enforce a good programming discipline. However, to increase the performance, most of the languages do not impose syntactic restrictions on the use of shared resources. The applicability of this theory to current programming languages is thus, limited. It should be clear that to grant correctness to algorithms that does not follow such discipline, an approach should not base the reasoning on a syntactically separation of resources enforced by the programming language.

2.2 Concurrent Separation Logic

Concurrent separation logic (CSL) [7] is designed to reason about programs that share resources defined on a heap. Differently from TTPP, this logic does not require to the programming language to provide any syntactic constructor to explicitly separate resources. The word "separation" in the title, inherit from

[1] The semaphore is transparent to the programmer.

separation logic [12], would be more intuitive if read as "ownership". This because proofs on this logic are structured using ownership transfer as the central idea.

A specification in CSL is a pair composed by a precondition and the postcondition, as in TTPP [9]. A proof in this logic maintains the Hoare-like interpretation of the contract between the precondition and the postcondition, but additionally it also ensures absence of data races and null pointer dereference on the heap manipulations. These extra guarantees are achieved by proving that the code preserve two basic properties: i) no code fragment access an address in the heap before it acquires ownership of it and, ii) no ownership is shared between processes or groups of mutual exclusion. Although early versions of CSL stress the fact that ownership of the heap cannot be shared, this limitation can be overcome adding fractional permissions to the logic as shown by Boyland and Bornat et al. [13, 14]. Thus, shared read on the heap is not a problem for concurrent separation logic. Shared writing, on the other hand, is not allowed in CSL. This rules out the need of possible values when writing specifications in concurrent separation logic, because if a process starts its execution with ownership of a shared resource, the resource will not be changed by the environment unless the process gives up the ownership of the resource.

May the most notorious feature in this logic is the fact that ownership is not static. O'Hearn's seminal paper [7] uses semaphores to model resource-holders, which are entities that hold and release ownership of portions of heap during the execution of a program. The interplay between processes and resource-holders gets more interesting in presence of parallelism. In such case permissions to access disjoint parts of a data structure may be distributed to several processes. The understatement of ownership as a permission to access an memory address leads to a new way of reading assertions in CSL. For instance, the assertion $\{m \mapsto -\}$ can be read as "there exists an value which is associated to the memory location m", but it is more elucidating if read as "the code that follows this assertion is granted right to read, write and dispose the memory address m".

An achievement of CSL is that it allows *local reasoning*. This means that specifications focus on the process *footprint*, which is the set of resources the process is allowed to access to implement its specification. Different parts of a specification can be reasoned separately then composed using the rules for introduction of parallelism. The *frame rule* from CSL takes local reasoning and gives a step further, turning CSL into a modular logic. This mean that proofs on this logic can be reused independently of the context.

It is worth to mention however that data race *per si* does not characterize bad designed programs. Algorithms as *Sieve of Eratosthenes*, which perform concurrent idempotent assignments to a value, do not fit to the notion of ownership, because two process may assign a value to shared resource before acquire ownership of it. In cases of that of this algorithm, instead of restrict data races, restrict the interference is exactly what is needed to prove correctness. Another drawbacks of separation logic is the difficult to account for the ownership transfer of stack variables.

Finally, it should be noted that CSL is a bottom-up approach. The reason happens at the implementation level, and an abstract needs to be found from this point to define the resource invariant. This is particulaly useful for legacy code, but may not be the best option for systems that are in the design phase, where errors might be find before the implementation takes place.

2.3 Rely/Guarantee

Rely-guarantee is a compositional verification method for shared memory concurrency based on the Owicki-Gries [10] method. The main feature of this method is that it gives to the designer the ability to record the interference the process must tolerate from the environment and the interference the process may inflict in the environment.

Differently from its predecessor, as well TTPP and CSL, here postconditions are relations over the previous and the after states. This decision is made to avoid unnecessary auxliary variables in the specification process. Formally, a specification in Rely/Guarantee is a 4-tuple, (P, R, G, Q), composed by a precondition P, a postcondition Q, a rely predicate R and a guarantee G. The pre and postcondition have the usual meaning, while the rely R and the guarantee G summarise the properties of the individual atomic actions invoked by the environment and the thread itself. The rely condition models all atomic actions the environment can execute concurrently with the thread, and the thread should tolerate, on the other hand, the guarantee models all the atomic actions the thread can execute during its life cycle and what the effects the environment will be exposed.

The rely/guarantee has a well-formedness condition: the precondition P, and the postcondition Q must be stable under the rely R. This means that these predicates are resistant to the environment, or in other words, the environment actions cannot invalidate these predicates. A drawback of rely-guarantee is the inability to refer to intermediate states in the definition of the post-condition.

2.4 RGSep

2.5 Refining rely-guarantee

In [3], Hayes, Jones and Colvin extend Morgan's refinement calculus [15] to cope with rely-guarantee. The change of notation proposed in this refinement calculus, from an 4-tuple style to an algebraic style, provides a better understanding of the roles of the relies and the guarantees, because it allows to refine each of them separately. This notation leads to a theory with a reduced set of basic laws, where complex laws are derived from basic ones.

2.6 Fundamental concepts

Foundational concepts, such as critical sections, semaphores, interleaving, monitors, etc. were lay, among others, by Edsger Dijkstra [16, 8], Tony Hoare [9]

and Per Brinch Hansen [17] in the sixties and seventies. This subsection sets the vocabulary used to distinguish some approaches.

Interference and data races

When comparing concurrent separation logic and rely-guarantee there is a need for clarify the difference between interference and data-races. The term "interference" is used to refer that a variable that is manipulated by a program may change its values due to the environment action. The shared resource may be controlled by a mutual exclusion mechanism or not. As long as the values setted by a process can change due to the environment, the variable is susceptible to interference.

Data races presuppose that the modifications happen in an uncontrolled way, and that at any time the processes can access the resource, but this access comes with no guarantee of consistent reading and writing. In general, algorithms tend to avoid data races, however, this behaviour *per si* is not enough to judge a algorithm as incorrect. In some cases, algorithms may tolerate data races as the *Sieve of Eratosthenes*

Ownership reasoning

The exclusive access (or right to write access) to a resource is a common feature of several algorithms. Having this in mind, a way of reasoning about algorithms is to logically verify if the shared resources can be accessed simultaneously by different processes which can write on them. This approach aims to restrict data races, and is the path followed by concurrent separation logics. It allows for processes to exchange the right of writing and reading the shared resources. Ownership of resources may be hold by processes or resource holders (e.g. semaphores, monitors, etc.).

Interference reasoning

Limiting interference is another way of reasoning about algorithms. In practice, concurrent algorithms can only tolerate a limited interference from the environment. This is the path followed by rely-guarantee. The central idea is to keep explicit relations establishing the interference a process tolerate from the environment and the interference the process inflict on the environment. When these relations are part of a specification, this relations may be used during correctness proofs.

Atomicity

Atomicity of an action defines if an action can be interrupted or not. When an action can be interrupted, concurrent process can observe intermediate values before the operation completes. Atomicity is determined by the operational semantics provided to the programming language. To illustrate the issue of atomicity, regard the assignment $x := x + x$. If the operational semantics define assignments to be atomic, there is no possibility of the x to change during execution of this assignment. However, if assignments are defined to be non-atomic, a statement $x := x + x$ might actually result in an odd number being assigned to the final value of x.

3 Specifications in rely-guarantee

This program logic (RG) [11, 18] is a compositional version of the Owicki-Gries method [10]. It records the interference concurrent processes impose an one another. The name rely-guarantee is a reference to a pair of relations called rely and guarantee.

A process specification in RG is a 4-tuple composed by a precondition, a rely, a guarantee and a postcondition. The pairs, precondition and postcondition, and rely and guarantee, may be seen as contracts. The is a contract between the initial and final state, and the latter is a contract between the process and environment. Precondition and postcondition have the same meaning from TTPP [9]. The new bit is the contract, between the process and the environment, it limits the actions the programmer can take to implement the postcondition, and reveals which assumptions the programmer is invited to make on the environment atomic actions. If the system is deployed into a environment where the assumptions do not hold, there is no commitment of satisfiying the contract between the precondition and the postcondition.

The restrictions in rely and guarantee revolve around a notion of atomic actions. It is worth to highlight that rely-guarantee makes no assumption on the atomicity. All the information about the atomicity of the actions is taken from the operational semantics given by the designer to the targeted language. The concern here is that any observable interferece generated by the environment is bounded by the rely relation. Notice that atomic actions may be broken in intermediate states, as long as these states are are not externally visible by the environment.

Although rely-guarantee accounts for development of concurrent programs, it also extends to sequential process. In such case, the rely is regarded the identity relation (i.e. no interference is allowed), and the guarantee is the least restrictive relation, *true*, that does not limit the changes a process can make in the state.

Proof obligations are described in [19].

4 Case Study

An unbounded buffer is used to illustrate an intricate feature that can arise in specifications on rely-guarantee theory. We start from an sequential version, and then move for concurrent version. Data refinement itself is not the most important part of this Section. The objective here is to elicit that extensions are required to rely-guarantee overcome a "problem" that appears at the most abstract level. The problem is the explicit relationship between the postcondition and the rely. Refinement is used to illustrate that the "problem" may be overcomed if we change the level of abstraction of the data structure.

The unbounded buffer discussed in this session comes from [7] (reference to the original [8] is made in that document). This data structure is used by a consumer-producer pair operating in different extremities of it. The 4-tuple version of rely-guarantee [19] is used here instead of its algebraic version [3].

The reason for this choice is that we want to focus on the "problem" instead of being distracted by the particularities of the new algebraic notation.

For this discussion we also included also an abstract sequential version of the consumer-producer example. The purpose of the sequential version is to allows us to discuss later if possible values can reduce the gap between an sequential specification and a concurrent one.

4.1 Unbounded Buffer

Unbounded buffers are linear data structures without size restrictions. In our context, such a structure is a *fifo*-queue manipulated by a single producer-consumer pair that operate concurrently, each of them in a different extremity of the buffer. Consumption occurs always in the left extremity, and addition in the right extremity of the buffer. The consumer enables itself whenever the buffer is not empty, whereas the producer has no restrictions on its operation.

A sequence of elements of the type T is chosen to represent the abstract buffer. By definition its domain is finite. A small refinement step is to allow the concrete buffer to be a function from indexes to the type T, whose invariant states that there are two values, i and j, such that the domain is contained in the range $i..j$. Our interest is not the refinement itself, but investigate the suitability of rely-guarantee in keeping the postcondition self contained, i.e. without explicitely states the changes in the data-structure done by the environement (these changes are explictely in the rely).

$$
\begin{array}{l}
buf : T^* \\
cbuf : \mathbb{N} \pfun T \\
cbuf\_inv = ((i \leq j) \ \lor \ (i = j + 1)) \ \land \ (\mathrm{dom}\ cbuf \subseteq i \,..\, j)
\end{array}
$$

Data refinement The retrieve function *ret* establishes the relation between the concrete representation ($cbuf$) and the abstract one (buf).

$$
\begin{array}{l}
ret = (\lambda\, x : \mathbb{N} \pfun T \mid \mathrm{dom}\, x \subseteq i..j \bullet \\
\quad \textbf{if}\ i \leq j\ \textbf{then}\ (\lambda\, ind : 1..j - i + 1 \bullet x(i + ind - 1)) \\
\qquad \textbf{else}\ \langle\rangle)
\end{array}
$$

Sequential Specification. A consumer and a producer operate indefinitely on a non-deterministic order on the buffer, as show by the specification $System$.

$$
\begin{array}{l}
System = \textbf{do} \\
\qquad\quad true \rightarrow Consumer_{SEQ}(o) \\
\qquad [] \quad true \rightarrow Producer_{SEQ}(e) \\
\qquad \textbf{od}
\end{array}
$$

The consumer returns the head of the buffer ($o = hd\ buf$) and removes it ($buf' = tail\ buf$), whereas the producer adds an element (e) after the last element of the buffer ($buf' = buf \frown \langle e \rangle$).

$$
\begin{array}{l}
Consumer_{SEQ}(o : T) = \\
\quad \textbf{ext wr } buf : T^* \\
\quad \textbf{pre } buf \neq \langle\rangle \\
\quad \textbf{pos } (o = hd\ buf) \wedge (buf' = tail\ buf)
\end{array}
$$

$$
\begin{array}{l}
Producer_{SEQ}(e) = \\
\quad \textbf{ext wr } buf : T^* \\
\quad \textbf{pre } true \\
\quad \textbf{pos } buf' = buf \frown \langle e \rangle
\end{array}
$$

Abstract Concurrent Specification. Here consumer and producer run in parallel. One of the consequences of the parallelism is that now post-conditions need to account for the interference caused by the environment.

$$
\begin{array}{l}
System = \textbf{do} \\
\quad Consumer_{RG}(o) \parallel Producer_{RG}(o) \\
\textbf{od}
\end{array}
$$

The consumer can operate only if the buffer is not empty ($buf \neq \langle\rangle$), and it removes the head of the buffer. Its guarantee states that every atomic action used in the implementation can remove elements from the left extremity of the buffer (buf' suffix buf). The rely allows to new elements to be added to the right extremity of the buffer by the environment (buf prefix buf'). We use s in the postcondition to represent a sequence of elements that may be concatenated in the right extremity of the buffer by the environment. Operators suffix and prefix denote that the left argument is a suffix or a prefix (respectively) of the right argument.

$$
\begin{array}{l}
Consumer_{RG}(o : T) = \\
\quad \textbf{guar } buf' \ \mathsf{suffix}\ buf \\
\quad \textbf{rely } buf \ \mathsf{prefix}\ buf' \\
\quad \textbf{pre } buf \neq \langle\rangle \\
\quad \textbf{pos } \exists\, s : \mathrm{seq}\ T \cdot (o = hd\ buf) \wedge (buf' = tail\ buf \frown s)
\end{array}
$$

The producer can operate at any time (**pre** $= true$), and its atomic actions cannot remove elements from the buffer, just add elements (buf prefix buf'). The postcondition shows that at the termination, a new element is added to the right extremity of the buffer, and some elements may have been removed from the left extremity by the environment (buf' suffix buf). We use s to denote that the environment may modify the initial buffer during the process execution (s suffix buf).

$$
\begin{array}{l}
Producer_{RG}(e : T) = \\
\quad \textbf{guar} = buf \ \mathsf{prefix}\ buf' \\
\quad \textbf{rely } = buf' \ \mathsf{suffix}\ buf \\
\quad \textbf{pre } = true \\
\quad \textbf{pos } = \exists\, s : \mathrm{seq}\ T \mid s \ \mathsf{suffix}\ buf \cdot buf' = s \frown \langle e \rangle
\end{array}
$$

The proof obligations regarding the mutual existence of the rely and guarantee [19] in this case are logical implications assuming the form $P \Rightarrow Q$, where P is a guarantee of the producer (consumer), and Q is the rely of the consumer (producer). As it comes to the case that $P = Q$ in both cases, these implications assume the form $P \Rightarrow P$, which are tautologies.

Three extra proof obligations appear on [19]: **PR-ident, RQ-ident, QR-ident**. **Unfortunately**, this specification does not get passed by **QR-ident** proof obligation from [19][2]. We state this fact as:

$$\mathit{Producer\text{-}post} \fatsemi \mathit{Producer\text{-}rely} \not\Rightarrow \mathit{Producer\text{-}post}$$

By replacing the terms:

$$\exists\, s : \mathrm{seq}\ T \mid s\ \mathsf{suffix}\ \mathit{buf} \cdot \mathit{buf}' = s \frown \langle e \rangle \fatsemi (\mathit{buf}'\ \mathsf{suffix}\ \mathit{buf}) \not\Rightarrow \\ \exists\, s : \mathrm{seq}\ T \mid s\ \mathsf{suffix}\ \mathit{buf} \cdot \mathit{buf}' = s \frown \langle e \rangle$$

The non-implication is show by a counter-example. Let the first ocurrency of buf to the empty sequence:

$$\mathit{buf}' = \langle e \rangle \fatsemi \mathit{buf}'\ \mathsf{suffix}\ \mathit{buf} \not\Rightarrow \exists\, s : \mathrm{seq}\ T \mid s\ \mathsf{suffix}\ \mathit{buf} \cdot \mathit{buf}' = s \frown \langle e \rangle$$

Now choose buf' in the right side of the relation composition to be the empty buffer. This results in $\mathit{buf}' = \langle\rangle$ on the left side of the implication. This last predicate is false, as there is no sequence that when concatenated with an singleton sequence "consumes" the other argument of the concatenation.

$$\mathit{buf}' = \langle\rangle \Rightarrow (\exists\, s : \mathrm{seq}\ T \mid s\ \mathsf{suffix}\ \mathit{buf} \cdot \mathit{buf}' = s \frown \langle e \rangle)$$

Concrete Concurrent Specification This specification replaces the original buffer, $(\mathit{buf} : T^*)$, by a mapping from indexes to values of type T, $(\mathit{cbuf} : \mathbb{N} \pfun T)$. The new buffer comes with an invariant $(\mathit{cbuf\_inv} = ((i \leq j) \ \lor\ (i = j + 1)) \ \land\ (\mathrm{dom}\ \mathit{cbuf} \subseteq i \mathinner{..} j))$. The consumer uses $\lhd$ to denote domain restriction. Notice that the need for explicitly mention the interference in the postcondition disappears.

$$\begin{array}{l} \mathit{Consumer}_{RG}(o : T) = \\ \quad \mathbf{guar} = i' \geq i \ \land j' = j \ \land \\ \qquad \{i + 1 \mathinner{..} j\} \lhd \mathit{buf}' = \{i + 1 \mathinner{..} j\} \lhd \mathit{buf} \\ \quad \mathbf{rely} = j' \geq j \ \land i' = i \ \land \\ \qquad \{i \mathinner{..} j\} \lhd \mathit{buf}' = \{i \mathinner{..} j\} \lhd \mathit{buf} \\ \quad \mathbf{pre} \ = i \leq j \\ \quad \mathbf{pos} \ = \{i + 1 \mathinner{..} j\} \lhd \mathit{buf} = \{i + 1 \mathinner{..} j\} \lhd \mathit{buf}' \land \\ \qquad o = \mathit{buf}\ i \ \land \\ \qquad i' = i + 1 \end{array}$$

[2] I have not found a way of overcome this issue and fix the specification. But I have kept this specification to illustrate the applicability of *possible values*

$$
\begin{array}{l}
Producer_{RG}(e : T) = \\
\quad \textbf{guar} = j' \geq j \ \wedge i' = i \ \wedge \\
\qquad \{i \,..\, j\} \lhd buf' = \{i \,..\, j\} \lhd buf \\
\quad \textbf{rely} = i' \geq i \ \wedge j' = j \ \wedge \\
\qquad \{i+1 \,..\, j\} \lhd buf' = \{i+1 \,..\, j\} \lhd buf \\
\quad \textbf{pre} \ = true \\
\quad \textbf{pos} \ = \exists\, i \cdot \{i \,..\, j\} \lhd buf = \{i \,..\, j\} \lhd buf' \wedge \\
\qquad j' = j + 1 \wedge buf'(j') = e
\end{array}
$$

Separation Logic Approach. Separation Logic deals with this problem by stating each time, if a buffer cell is not owned by the environment, it is owned by either the consumer, or the producer. In what follows, the correctness proof is based on the resource invariant (RI) built on ls.

The buffer example discussed in Section 4 is taken from the O'Hearn's paper on CSL [7]. The CSL version of it uses a dumb node for efficiency purposes in the insertion. This has the consequence that the last node of the buffer is always owned by the producer, while a semaphore holds the ownership of nodes between the head the and the penultimate node.

$$
\begin{array}{l}
ls[x\ n\ z] \doteq (x = z \wedge n = 0 \wedge \texttt{emp}) \vee \\
\qquad\qquad (x \neq z \wedge n > 0 \wedge \exists\, y \cdot x \mapsto -, y * ls[y\ (n-1)\ z]) \\
RI \doteq ls[f\ number\ l]
\end{array}
$$

Discussion Both approaches are suitable for dealing with this example. The data races which are observable when looking to the whole data structure, disappears when we focus our attention to each position of the buffer. Thus, both formalisms fit to this example. To record designs, it might be useful to adopt rely-guarantee, whereas if the code is already provided, and the buffer is defined on the heap, concurrent separation logic would suit better the verification. Notice that the dummy node does not appear on the rely-guarantee specification, which is not concerned with performance of the implementation.

Regarding the rely-guarantee specification, it is worth to notice that the abstract (concurrent) version could benefit from the notion of possible values.

5 Extensions to Rely-Guarantee

Possible values is a *convention* [6] representing a set of expected values for a variable a process may observe as result from the interference. What follows is an attempt to formally define it. We denote by S the current state, by r the rely of a process, and by g the guarantee of a process. The predicates $rely(r, S_u, S_k)$ and $guar(g, S_u, S_k)$ relate the states S_u (before state) and S_k (after state) with the rely (r) or guarantee (g), respectively. The relation $closure_{rg}$ is the reflexive transitive closure of $rely \vee guar$.

One direction to define possible values is by including both environment updates as well the updates made by the process to a variable. In this understanding

a possible value for a variable var is achieved by any aleatory application of the guarantee and rely relations over the state.

$$\widehat{var}(S, r, g) = \{var_k \mid var_k \in S_k \cdot \\ (rely(r, S_u, S_k) \vee guar(g, S_u, S_k)) \wedge closure_{rg}(S, S_u)\}$$

Including the process steps leads to the problem of self-reference. For example, in the buffer example the expression $\exists\, b : \widehat{buf} \cdot buf' = b \frown (e)$ would allow for several copies of the same element to be added to the right extremity of the buffer. If we restrict possible values to the changes made by the environment, then we could make use of possible values in a postcondition to define the final value of the buffer. The next definition does this restriction:

$$\widehat{var}(S, r) = \{var_k \mid S_k(var) = var_k \wedge rely(r, S, S_k)\}$$

5.1 Possible Invariant Preservation

In addition to the original proof obligations of rely-guarantee [19], each value in a possible values set should satisfy the invariant. The idea here is to delimit the values in possible values by state's invariant.

$$\forall\, S \cdot INV(S) \Rightarrow \\ \exists\, S' \cdot INV(S') \wedge rely(r, S, S') \wedge (\forall\, var \in S \cdot \forall\, i \in \widehat{var} \cdot INV(S[var \mapsto i]))$$

6 Conclusions

Developing concurrent programs is difficult. In part, this is because of the numerous pitfalls that should be avoided to produce code that never deadlocks and continuously produces the expected result. To prove correctness of such programs, a method needs to be either based on mathematical proofs or model checking, i.e. concurrency generate forms of interleaving that are not tolerable and testing is not efficient to prove the absence of them.

In this technical report, a buffer example is developed in rely-guarantee to show how specifcations can benefit from the concept of possible values. We claim that when the postcondition needs to be defined in terms of the intermediate values that arise during the lifetime of an operation, possible values may simplify the description of the process.

Acknowledgement

This PhD research is supported by the School of Computing Science, Newcastle University. Thanks go for Cliff Jones for the avalability for inumerous discussions about the overall project, for Leo Freitas for the carefully guidance, for Matthew Parkinson for making me aware of four different concerns that were being put together by the Taming Concurrency Project, namely, abstraction, refinement,

rely-guarantee and ownership logics, for Ian Hayes for discussion about posit and prove and transformational approach, for Ilya Lopatkin for the discussion about the preliminary model of the unbounded buffer, and for Rosemeire Fiaccone for critics on the writing style adopted in a preliminary version of this report.

References

1. C. Jones, *Development Methods for Computer Programs Including a Notion of Interference*. Technical monograph, Oxford University Computing Laboratory, 1981.
2. V. Vafeiadis, "Modular fine-grained concurrency verification," Tech. Rep. UCAM-CL-TR-726, University of Cambridge, Computer Laboratory, July 2008.
3. R. J. C. Ian J. Hayes, Cliff B Jones, "Refining rely-guarantee thinking.".
4. J. Dingel, "A refinement calculus for shared-variable parallel and distributed programming," *Formal Aspects of Computing*, vol. 14, no. 2, pp. 123–197, 2002.
5. C. A. R. Hoare, "An axiomatic basis for computer programming," *Commun. ACM*, vol. 12, pp. 576–580, Oct. 1969.
6. C. B. Jones and K. G. Pierce, "Elucidating concurrent algorithms via layers of abstraction and reification," *Formal Asp. Comput.*, vol. 23, no. 3, pp. 289–306, 2011.
7. P. W. OHearn, "Resources, concurrency, and local reasoning," *Theor. Comput. Sci.*, vol. 375, pp. 271–307, Apr. 2007.
8. E. W. Dijkstra, "Cooperating sequential processes," in *Programming Languages: NATO Advanced Study Institute* (F. Genuys, ed.), pp. 43–112, Academic Press, 1968.
9. C. A. R. Hoare, "Towards a Theory of Parallel Programming," in *Operating Systems Techniques*, vol. 9 of *A.P.I.C. Studies in Data Processing*, pp. 61–71, Academic Press, 1972.
10. S. Owicki and D. Gries, "An axiomatic proof technique for parallel programs i," *Acta Informatica*, vol. 6, no. 4, pp. 319–340, 1976.
11. C. B. Jones, "Tentative steps toward a development method for interfering programs," *ACM Trans. Program. Lang. Syst.*, vol. 5, pp. 596–619, Oct. 1983.
12. P. W. O'Hearn, J. C. Reynolds, and H. Yang, "Local reasoning about programs that alter data structures," in *Proceedings of the 15th International Workshop on Computer Science Logic*, CSL '01, (London, UK, UK), pp. 1–19, Springer-Verlag, 2001.
13. J. Boyland, "Checking interference with fractional permissions," in *Proceedings of the 10th International Conference on Static Analysis*, SAS'03, (Berlin, Heidelberg), pp. 55–72, Springer-Verlag, 2003.
14. R. Bornat, C. Calcagno, P. O'Hearn, and M. Parkinson, "Permission accounting in separation logic," *SIGPLAN Not.*, vol. 40, pp. 259–270, Jan. 2005.
15. C. Morgan, *Programming from specifications (2nd ed.)*. Hertfordshire, UK, UK: Prentice Hall International (UK) Ltd., 1994.
16. E. W. Dijkstra, "The structure of the "THE"-multiprogramming system," *Comm. ACM*, vol. 11, no. 5, pp. 341–346, 1968.
17. P. B. Hansen, "Structured multiprogramming.," *Commun. ACM*, vol. 15, no. 7, pp. 574–578, 1972.
18. C. B. Jones, "Specification and design of (parallel) programs," in *IFIP Congress*, pp. 321–332, 1983.

19. J. W. Coleman and C. B. Jones, "A structural proof of the soundness of rely/guarantee rules," *J. Log. and Comput.*, vol. 17, pp. 807–841, Aug. 2007.
20. J. M. Spivey, *The Z Notation: A Reference Manual*. Upper Saddle River, NJ, USA: Prentice-Hall, Inc., 1989.