Using CSP to Develop Quality Concurrent Software

Derrick G. Kourie, Tinus Strauss, Loek Cleophas, and Bruce W. Watson

Abstract A method for developing concurrent software is advocated that centres on using CSP to specify the behaviour of the system. A small example problem is used to illustrate the method. The problem is to develop a simulation system that keeps track of and reports on the least unique bid of multiple streams of randomly generated incoming bids. The problem's required high-level behaviour is specified in CSP, refined down to the level of interacting processes and then verified for refinement and behavioural correctness using the FDR refinement checker. Heuristics are used to map the CSP processes to a Go implementation. Interpretive reflections are offered of the lessons learned as a result of the exercise.

1 Introduction

In software engineering, the functional requirements state *what* the system should do. Non-functional requirements are stated in terms of quality attributes that characterise *how* these functional requirements are to be attained. Examples of such attributes include maintainability, portability, security, etc. Many authors include correctness—the extent to which the software agrees with its specification—as a

D. G. Kourie (✉) · T. Strauss
Department of Information Science, Stellenbosch University, Stellenbosch, South Africa
e-mail: dkourie@fastar.org; tinus@fastar.org

L. Cleophas
Department of Information Science, Stellenbosch University, Stellenbosch, South Africa

Department of Mathematics and Computer Science, Eindhoven University of Technology, Eindhoven, The Netherlands
e-mail: loek@fastar.org

B. W. Watson
Department of Information Science, Stellenbosch University, Stellenbosch, South Africa

Centre for Artificial Intelligence Research, CSIR Meraka Institute, Pretoria, South Africa
e-mail: bruce@fastar.org

P. Müller, I. Schaefer (eds.), *Principled Software Development*,
https://doi.org/10.1007/978-3-319-98047-8_11

quality attribute. The overall *quality* of a given system is measured by the extent to which these quality attributes are attained.

A spectrum of relatively mature software development strategies has evolved over time that are aimed at ensuring high quality software systems. Some are programming-in-the-small strategies such as correctness-by-construction (CbC) and post-hoc verification. These aim principally at producing error-free code. Others may be thought of as programming-in-the-large strategies. They are represented by an array of software methodologies (such as Agile or RUP, etc). These are oriented towards ensuring that functional requirements are met in such a way that pre-specified levels of a broad range of quality attributes are met.

However, most of these strategies are aimed primarily, if not exclusively, at producing *sequential* software systems. One might have expected that the proliferation of multiprocessor hardware systems (such as multicore chips and graphical processing units) would have spawned an array of comparable strategies for developing *concurrent* systems, but this has not materialised to any significant extent.

The pioneering attempt of Owicki and Gries [8] to extend CbC to concurrent contexts has not gained traction. Neither have various proposals for concurrent software development methodologies[1] gained a significant foothold in industry. Examples of such proposals include [4, 7, 9, 11, 12, 16].

Ironically, quite a number of languages have emerged, both for *specifying* and *implementing* concurrent software systems. Specification languages include Estelle (ISO standardised); process algebras such as LOTOS (also ISO standardised), CCS and CSP; and finite automaton-based notations such as Harel's State Charts (used in UML). Programming languages with significant support for concurrency include Ada, Esterel, Occam, concurrent C++, concurrent Java, Erlang, GO, etc.

In this chapter, a small concurrent software system is formally specified, refined and then eventually implemented. The specification's development style is "iterative incremental". It starts with an abstract high-level CSP specification of the overall system whose behaviour very broadly conforms with the product requirements. FDR (a tool for analysing CSP specification refinements) is used to verify the correctness of each refinement step. When the iteratively refined specification has sufficiently exposed an architectural and logical structure for the problem's solution, the specification is mapped to a system of interacting *software processes*, expressed in the GO programming language.

The next section provides the background needed in respect of CSP and FDR, and suggests heuristics for mapping CSP to GO. Section 3 then describes the functional requirements for the system to be developed. Section 4 walks through the process of incrementally refining an initial abstract CSP specification and verifying the

[1]The notion of a "concurrent software development methodology" is ambiguous. It could refer to a methodology aimed at developing *concurrent software*, or to a methodology that advocates *concurrent execution of tasks* in the software development life cycle (sometimes referred to as concurrent engineering). Here the term is used in the former sense.

refinement steps. Section 5 outlines how the refined specification was mapped to an implementation in GO. Some interesting results obtained from the implemented system are displayed in Sect. 6. We conclude in Sect. 7 with some reflective comments about our experience.

2 Background

Since being open-sourced in 2009, the GO language has surged in popularity, climbing on the TIOBE index from 65th in 2015 to 14th at the time of writing.[2] Its developers aimed at a language that not only compiles efficiently to produce object code that runs efficiently, but that also supports concurrency. To achieve the latter objective, they based aspects of the language on CSP. They justified this choice as follows:

> Concurrency and multi-threaded programming have a reputation for difficulty. . . . One of the most successful models for providing high-level linguistic support for concurrency comes from . . . CSP. GO's concurrency primitives derive from . . . the powerful notion of channels as first class objects. Experience with several earlier languages has shown that the CSP model fits well into a procedural language framework.[3]

The foregoing confirms and motivates our choice of CSP, FDR and GO as a trilogy of technologies upon which to rely in our quest to evolve a concurrent software methodology.

Below we briefly introduce CSP and the notion of refinement in that context, we introduce relevant features of the FDR tool as well as the syntax of CSP variant used by the tool, indicated as CSP_M. Finally, features of GO that are pertinent to this study are introduced as well as the guidelines that were used to map CSP_M specifications to GO.

2.1 CSP and Refinement

CSP (Communicating Sequential Processes) is a process algebra that was originally proposed by Hoare as a language for specifying the behaviour of concurrent systems. Over time various modifications and extensions have been proposed (e.g. to model real time systems or asynchronous and shared memory systems). A comprehensive introduction and reference volume for CSP is available in [10].

CSP envisages that a process interacts with its environment (one or more other processes) in terms of *events* from an alphabet. It is essentially a notation for concisely describing the set of event sequences (called *traces*) that characterise a

[2]https://www.tiobe.com/tiobe-index/.

[3]https://golang.org/doc/faq.

process's interaction with its environment. Events are abstractions that are regarded as *atomic*—i.e. they have no duration and thus cannot occur simultaneously, but only before or after one another.

The trace set of process `P` is denoted by `traces(P)`. It includes all prefixes of all traces of `P`. Such a trace set can be regarded as the semantics of `P`. Process `Q` is said to trace-refine process `P` iff `traces(Q)` $\subseteq$ `traces(P)`.

CSP includes a more elaborate notion of refinement based on so-called *failures*. To briefly explain the notion of a failure, let `P/s` represent process `P` after trace `s`. A *refusal set* of `P/s` is *any* set of events, say `X`, that `P/s` can fail to accept from its environment, irrespective of how long one or more events in that set are offered. If `X` is a refusal set of `P/s`, then the pair `(s,X)` is called a *failure* of `P`.[4] The set of all failures of `P` over all of its traces is denoted by `failures(P)`. Process `Q` is said to failure-refine process `P` iff `failures(Q)` $\subseteq$ `failures(P)` and `traces(Q)` $\subseteq$ `traces(P)`. If Σ is the alphabet of `P` and `(s,`Σ`)` is a failure of `P`, then P has a deadlock after executing trace `s`.

There is yet another CSP notion of refinement related to so-called *divergences*. In order to remain concise, we will not discuss divergences in detail here. Nevertheless, we note that the definition for divergence-refinement is similar to that of failure-refinement. We furthermore note that divergences are associated with starvation (livelock).

2.2 FDR and CSP$_M$

In this text, a keyboard friendly syntactical variant of CSP known as CSP$_M$ will be used. This allows us to make use of FDR (version 4), an open source tool for analysing CSP specifications [2, 3, 10]. The tool can be used to automatically check for certain correctness attributes in a specification. It can also be used to generate the labelled transition diagram that visualises a specification.

Figure 1 summarises the CSP$_M$ operators that will be used in subsequent text. An informal explanation of these operators and their operands follows. Formal operator semantics and laws for their manipulation are available in various texts such as [5, 6, 10].

`a -> P` represents an **unnamed** process that does nothing until its environment offers event `a`, at which point the process engages with that event and thereafter behaves as the **named** process, `P`. The unnamed process may be named `Q` by writing `Q = a -> P`.

[4]Note that `X` does not have to be maximal to be a refusal set of `P/s`. There can therefore be many refusal sets of `P/s`. The set of such refusal sets at `P/s` is denoted by `refusals(P/s)`, each such refusal set giving rise to an additional failure of `P`.

`a -> P`	event *a* **then** process *P*
`P [] Q`	**External choice** between `P` and `Q`
`P |~| Q`	**Internal choice** between `P` and `Q`
`P [|A|] Q`	`P` in **shared parallel** with `Q`
	Synchronize on events in the set `A`
`P ||| Q`	`P` **interleaved** with `Q`
	Equivalent to `P |{}| Q`; `P` and `Q` run independently
`P ; Q`	`P` **followed by** `Q`
`C!x`	**Output** the value `x` to channel `C`
`C?t`	**Input** into variable `t` from channel `C`
`P|\{|C|}`	**Hide** events in `P` *except* those on channel `C`
`Op a:A @ P(a)`	`P(a1) Op P(a2) Op ... P(an)` where `Op` is `|~|` or `|||`
	and `A = {a1, a2, ..., an}`

Fig. 1 Selected CSP$_M$ operators and operands

Using the notation `Q/s` to reference process `Q` after engaging in trace `s`, the foregoing means that `Q/<a>` references process `P`. Note, however, that `Q/s` is not part of FDR syntax.

Named processes may have zero or more parameters (e.g. `Q` or `P(x,y)`). The **let...within** clause is used to assign a name and initialise the parameters of a subprocess that is only known locally to some exteriorly defined process.

`SKIP` designates a special process that simply terminates successfully without engaging in further events.

In `Rext = (a -> P) [] (b -> Q)` two unnamed processes serve as operands to the **external choice** operator.[5] If its environment offers event `a` then `Rext` engages with `a` and the behaves thereafter as `P`. Conversely, if its environment offers event `b` then it engages with `b` then behaves as `Q`. The behaviour thus depends on which event is offered first by other processes in the environment. If `P` and `Q` are both replaced by `SKIP`, then `traces(Rext) =` {`<>,<a>,<b>`} where `<>` is the empty trace.

In contrast, `Rint = (a -> P) |~| (b -> Q)` has the same two unnamed processes, but these serve as operands to the **internal choice** operator. `Rint` *may* engage with event `a` if offered by the environment and then behave as `P`, but may also refuse to engage with `a`. Likewise, `Rint` *may* engage with event `b` if offered by the environment and then behave as `Q`, but may also refuse to engage with `b`. Externally, the behaviour of `Rint` manifests as non-deterministic. Its behaviour is determined by internal considerations. Its traces correspond to those of `Rext` but its failures do not.

FDR provides for various ways of expressing parallel synchronisation between two (or more) processes. Synchronisation is with respect to an explicit or implied set of events. `R = P [|A|] Q` represents a process that synchronises on any element

[5]Equivalence relationships indicate what occurs if operands are unguarded. For example, if an operand is `SKIP` then `Rext` would be equivalent to the `SKIP` process.

in the set A. Thus, if the environment offers a ∈ A, and both P and Q are ready to engage with a, then R/<a> = P/<a> [|A|] Q/<a>. However, if P and/or Q is not ready to engage with a, then **deadlock** occurs. If the environment offers a ∉ A, and P is ready to engage with a but Q is not, then R/<a> = P<a> [|A|] Q, and similarly if Q is ready to engage with a, but not P.

The process R = P ||| Q represents the interleaving of P and Q and is simply another way of writing R = P [||] Q. Here, P and Q respond to what the environment offers entirely independently of one another. Normally, it is assumed that their alphabets are disjoint.

In the process specification R = P;Q, the **followed by** operator is used. R behaves as P until it terminates successfully (i.e. evolves into SKIP). Thereafter, process R behaves as Q.[6]

CSP relies on the notion of **channels** as part of a process's environment. Channels synchronously connect processes to one another. Each channel has a specific alphabet determining events (or messages) that are communicated across it. If C designates a channel and x is an element of its alphabet, then the notation C!x indicates that x is to be output on C (or equivalently, the event C.x is offered to the environment.) Dually, C?t indicates that any one of the events in the alphabet of C may be input (received) into variable t.

Process Q = P|\{|C|} is a process whose traces are the traces of P except that all events not in the alphabet of channel C are **hidden**. Thus, if t is a trace of P and t' is t but stripped of all events except those that occur on C, then t' is a trace of Q.

The following notation is used to express the external choice or interleaving over n parameterised processes: Op a:A @ P(a) = P(a1) Op ... OP P(an), where Op is either |~| or ||| and A = {a1, a2, ..., an}

The syntax **if** Bexp **then** P **else** Q specifies a process that behaves as process P if the Boolean expression Bexp is true and behaves as process Q otherwise.

The FDR assert statement takes an FDR predicate as argument and appropriately returns true or false (together with debugging information). We used assert with the following predicates, P and Q being process names.

P [T= Q	Q trace-refines P
P [FD= Q	Q failure-divergence-refines P
P : [deadlock-free]	P is deadlock free
P : [divergence-free]	P is divergence free
P : [deterministic]	P is deterministic

[6]Again, equivalence relationships indicate what occurs under various conditions. For example, if P does not terminate successfully (i.e. it does not terminate, or terminates in the deadlock process, STOP, then P;Q ≡ P.

2.3 Go

Although Go is in the tradition of the C/Java family of languages, there are several differences: for example, declaration syntax differs and features such as generics, pointer arithmetic and inheritance are absent. Below, a very brief account is given of Go features that are relevant in this text. For more information, refer to standard documentation sources.[7]

In mapping CSP$_M$ to Go, we use a number of heuristics and special Go features.

- CSP$_M$ processes are mapped to Go functions that are subsequently launched as so-called goroutines. Each time the command `go somefun()` is issued, a goroutine instance of the function called `somefun` is allocated to one of the available cores on a multicore machine and runs to completion on that core. A goroutine may therefore be viewed as a process that runs concurrently with other goroutines / process, the allocation of processes to processors being sorted out automatically by the runtime environment.
- In general, the state of a CSP$_M$ process at any given point is reflected by the process's actual (as opposed to formal) parameters. The corresponding Go process's state is naturally stored in variables that were declared as local in the Go function's definition.
- Since Go supports channels, we map CSP$_M$ channels to Go channels. An instance of a channel, say `c`, that is to convey messages of a given type, say `<T>`, is created by the call `c := make(chan <T>)`. In our implementation, all channels were of type `int`.
- The CSP$_M$ command `C!a` (i.e. transmit `a` on channel `C`) is mapped to Go as `C <- a`.
- Similarly, `C?x` (read from a channel `c` into variable `x`) is mapped to Go as `x := <-C`.
- Since Go channels optionally may be buffered, there is no need to explicitly map every CSP$_M$ FIFO buffer process to a goroutine. For example, the command `C := make(chan int, 10)` creates the channel `C` that stores up to 10 integers in a FIFO buffer.
- Additionally, multiple goroutines can write to and/or read from the same Go channel. This feature was relied upon in constructing the multiplexing FIFO buffer described below in CSP$_M$.
- The Go for-loop construct `for e := range c {...}`, repeatedly reads a value from channel `c` into variable `e`, and then executes the body of the loop. Once the channel `C` is closed using the call `close(C)`, attempts to read from the channel are discontinued. This convenient Go feature was used in our implementation to ensure graceful closure of processes.
- CSP$_M$'s external choice is mapped to Go's `select` statement.

[7]For example https://golang.org/doc/.

It should be emphasised that the foregoing mappings are merely heuristics. They should not be mechanistically applied.

3 A Least Unique Bid Game Simulation

The least unique bid (LUB) game is generally played for charity fund raising purposes. Bids for a prize are solicited. All bids have to be paid over to the fund raising entity. Once bidding is closed, the bidder who has submitted the *least unique* bid wins the prize. If there is no unique least bid, then no prize is awarded.

The algorithmic task of finding the LUB of a given set of bid values is relatively straightforward. One approach is to sort the values, and then traverse the sorted values in ascending order until the first entry is found that differs from both its predecessor and its successor entries.

However, a simulation of the LUB game is to be developed that is more elaborate than merely computing the LUB of a given set of bid values. This simulation serves as a small case-study to illustrate the proposed CSP-based strategy for developing quality concurrent software. The simulation system's functional description is as follows:

- Several streams of independent random bids (representing, for example, random bids coming into a call centre) serve as input to a LUB calculator.
- The LUB calculator keeps track of the LUB (if it exists) with respect to all the bids it has received to date.
- Upon request from the user of the system, the LUB calculator returns the LUB most recently computed. If no LUB is present in the bids offered to date, it returns a special signal instead.

The system may be viewed abstractly as three interacting processes as loosely described below. `GEN` generates random bids that are offered to `LUB`. `LUB` processes all the received bids and offers results to `USER`.

```
System = GEN || LUB || USER
```

4 Refining `System` Processes

The informal abstract of `System` given above glosses over several details. For example, it ignores the fact that the output from `GEN` derives from several streams of independent random bids. Neither does it indicate the specific operations that `LUB` has to carry out. Nor does it name channels connecting the processes.

Below, CSP$_M$ will be used to describe the behaviour of each of the interacting processes of `System`. The CSP$_M$ description will then be incrementally refined. The FDR system is used to ensure that the refinements are consistent. Once the refinements are sufficiently detailed to expose an overall system architecture and execution logic to allow for easy heuristic mapping to GO structures, then the system is further developed in GO.

4.1 Refining LUB

The CSP$_M$ defined process below, LUB1, is a first approximation description of LUB.

```
LUB1(in,out) =
 let
  RandOut(Bids) =  |~| b:Bids @ out!b -> P(Bids)
  P({}) = in?b -> P({b})
  P(Bids) = in?b -> P(union({b},Bids))
         [] RandOut(Bids)
 within
  P({})
```

Channel in acquires bids from its environment (represented above by GEN). Channel out offers bids to its environment (represented above by USER). The internal process P has a set parameter (initially the empty set) that describes the set of bids received to date. The built-in FDR function union is used to update this set parameter. Another internal process, RandOut, copies to the out channel a randomly selected member from the set of received Bids. To express this random selection, the internal choice operator, |~|, is distributed over multiple unnamed processes, each process outputting a different element of Bids.

Even though LUB1 is highly simplified—it does not even select a *unique* bid, let alone the *least* unique bid—it has the advantage of exposing the skeletal outline of what we hope will emerge as the final refined version of the CSP$_M$ specification for LUB.

LUB2 is our next iteration refinement of LUB1. It ensures that a bid emitted on the out channel is a random representative of the set of *unique* bids received to date on the in channel. It therefore accomplishes the task of isolating unique bids, but avoids the task of identifying the *least* unique bid.

```
LUB2(in, out) =
 let
  RandOut(UBids,Bids) =  |~|b:UBids @ out!b -> P(UBids,Bids)
  P({},{}) = in?b -> P({b},{b})
  P(Bids,{}) = in?b -> (if member(b,Bids) then P(Bids,{})
                      else P(union({b},Bids),{b}))
  P(Bids,UBids) =
       in?b -> (if member(b,Bids) then P(Bids,diff(UBids,{b}))
               else P(union({b},Bids), union({b},UBids)))
   [] RandOut(UBids,Bids)
  within
    P({},{})
```

In order to keep track of unique bids, an internal process of `LUB2`, namely `P`, maintains *two* set parameters. The first, called `Bids`, is the set of received bids. The second, called `UBids`, is the set of unique bids received to date. Again built-in FDR functions, namely `member`, `union` and `diff`, are used to update these sets. The internal process, `RandOut` is modified to offer randomly selected members from the set of *unique* bids on the `out` channel.

`LUB3` refines `LUB2` so that the *least* of all unique bids is emitted upon request. Again, the same CSP$_M$ structure as `LUB2` can be retained. However, `UBids` is now maintained as a *sorted sequence* instead of as a set.

```
LUB3(in, out) =
  let
   P({},<>) = in?b -> P({b},<b>)
   P(Bids,<>) =
      in?b -> (if member(b,Bids) then P(Bids,<>)
              else P(union({b}, Bids), <b>))
      [] out!Null -> P(Bids,<>)
   P(Bids,UBids) =
      in?b -> (if member(b,Bids) then P(Bids,remove(b,UBids))
              else P(union({b}, Bids), insert(b,UBids)))
      [] out!head(UBids) -> P(Bids,UBids)
  within
   P({},<>)
```

The definition of `P(Bids,UBids)` achieves this by relying on two user-defined sequence functions, `remove` and `insert` (not reproduced here), instead of using the built-in set functions `diff` and `union`, respectively.

1. `remove(b,UBids)` returns the largest subsequence of `UBids` that excludes `b`.
2. `insert(b,UBids)` returns the smallest sorted super-sequence of `UBids` that includes `b`.

Furthermore, `LUB3` no longer needs the internal function, `RandOut`. Instead, an FDR built-in sequence function, `head`, is used to offer the head of the sorted sequence `UBids` to a request on channel `out`.

Note that `LUB3` returns `Null` on channel `out` if no LUB is present in bids processed to date. In contrast, `LUB1` and `LUB2` never offer a `Null` signal on the `out` channel. In this respect, `LUB3` is not a strict refinement of `LUB2` because it relies on an enriched alphabet for channel `out`.

However, the FDR tool affirms that `LUB3` trace-refines `LUB2` provided that the `Null` event is excluded from `LUB3`'s traces. It also affirms that `LUB2` trace-refines `LUB1`. It does so by verifying the following assertions:

```
assert LUB2(in, out) [T= LUB3(in,out)\{out.Null}
assert LUB1(in, out) [T= LUB2(in,out)
```

Note that the LUB processes above do note terminate. The matter of graceful process termination was ignored in the interests of clarity and brevity. LUB1T below shows how LUB1 may be changed to specify graceful termination upon receipt of a sentinel signal, End, received on the in channel, that is then relayed to the out channel before shutting down.

```
LUB1T(in,out) =
 let
  RandOut(Bids) =  |~| b:Bids @ out!b -> P(Bids)
    P({}) = in?b -> if b != End then P({b})
                    else out!End -> SKIP
    P(Bids) = in?b -> if b != End then P(union({b},Bids))
                      else out!End -> SKIP
             [] RandOut(Bids)
  within
    P({})
```

Similar adaptations are easily made to the other LUB variants. These adaptations do not disturb any of the refinement arguments.

4.2 Refining **GEN**

A first approximation for a CSP$_M$ definition of GEN would be a process, GEN1, whose definition is substantially the same as that of RandOut that was defined locally in LUB1 above. However, instead of generating an infinite stream of random numbers from the interval Range, as does RandOut, GEN1 generates only Max such numbers and outputs them to the channel out. The process therefore has the parameters GEN1(Max,Range,out).

While such a process GEN1 produces what LUB3 expects from GEN—a stream of random bids on a single channel—the stated system requirement is that these bids should derive from *multiple independent streams*.

GEN2 is therefore defined as a refinement of GEN1. It can be modelled as a set of M independent instances of GEN1, collectively named as the process MGEN, each instance sending its output to a *multiplexing* buffer that we will call MBuff. Using Mchans to denote the set of channels connecting each instance of GEN1 with Buff, GEN2 and MGEN may be defined as follows:

```
GEN2 = MGEN [|{|Mchans|}|] MBuff(Size, Mchans, out)|\{|out|\}
MGEN = |||i:ID @ GEN1(Max, Range, Mchans.i)
```

Here ID = {1, ...M}, indicating that there are M instances of GEN1. The i th instance of GEN1 outputs on channel Mchans.i, an element of Mchans.

Note that the expression with the hide operator at the end of GEN2's definition, |\ {|out|}|, means that GEN2's traces only contain events that occur on channel out—events occurring on channels Mchans.1...Mchans.M are hidden when FDR generates a trace description of GEN2.

In the interests of brevity, the full definition of the multiplexing buffer, MBuff, is not given here. It is an adaptation of a CSP$_M$ specification of a FIFO buffer that may be found in [10]. The first parameter of MBuff indicates its size and the third parameter is the single channel feeding bids to LUB3.

The FDR system affirms that the following two assertions are true:

```
assert GEN1(Max * M, Range, out) [FD= GEN2
assert GEN2 [FD= GEN1(Max * M, Range, out)
```

Thus, if an instance of GEN1 outputs to the same output channel as GEN2, and generates Max * M random integers in the same range as the M subprocesses in GEN2, and if the M subprocesses in GEN2 each generate Max random integers, then those instances of GEN1 and GEN2 are trace-, failure- and divergent-equivalent processes. This is in line with expectations.

4.3 Checking the System

The USER process can be modelled at various levels of detail. At its simplest, it can be seen as a process that merely consumes everything sent to it. USER(in) defined below is a slightly more elaborate model. It relies on a subtly defined local process, P, consisting of the external choice between two unnamed processes. The first receives input on channel in and then recurses to P but with its parameter overwritten by the received input, y. The second engages in the event use.x, and then recurses to P with no change in P's parameter. To start off, P is parameterised as Null.

```
USER(in) =
  let
    P(x) = in?y:Range -> P(y) [] use.x -> P(x)
  within
    P(Null)
```

Rather than connecting the USER process directly to LUB3, a standard FIFO buffer, BUFF(Cap,in,out) (as defined in [10]) between them will temporarily store the output from LUB3 until USER is ready to consume it. The buffer's parameters indicate its capacity, its input channel and its output channel. The overall system can therefore be modelled as follows in FDR.

```
System =
   (((GEN2 [|{|in|}|] LUB3(in,out))
           [|{|out|}|] BUFF(UserMax,out,uin))
           [|{|uin|}|] USER(uin))
```

Here, GEN2's output channel, named out in its definition above, should be renamed to in and serve as the input channel to LUB3 whose output is placed on channel out. The buffer, whose capacity is designated UserMax, receives its input from channel out and offers its data to USER on the channel parameterised here as uin.

The first two assert statements below verified that the system is deadlock and divergence (livelock) free. Subsystem1 in the third assert statement refers to the concurrent interaction of the LUB3, BUFF and USER processes. This assert statement verifies that if we remove the nondeterministic bid generators in GEN2, then the remainder of the system behaves deterministically.

```
assert System :[deadlock-free]
assert System :[divergence-free]
assert Subsystem1 :[deterministic]
```

The foregoing has presented *some* of the CSP$_M$ specifications that we developed and tested. When we judged that the CSP$_M$ models had reached a sufficient level of refinement maturity, and given us sufficient insight into the solution approaches, we used the specifications together with the heuristics mentioned in Sect. 2.3 as a guide to evolve a GO implementation.

5 Implementation

The overall architecture of System as specified in CSP$_M$ is visualised in Fig. 2. It was carried over as the basic architecture for the GO implementation, making adaptations appropriate for the GO environment.

The CSP$_M$ LUB subprocess was mapped to a corresponding GO function and then launched as a goroutine from within GO's main program. Two different versions were considered, for reasons explained below. The CSP$_M$ USER subprocess was implemented as final code of GO's main process.

Instead of a separate GO process matching the CSP$_M$ BUFF process that buffers messages from the out channel to the uin channel, the GO processes implementing LUB and USER are linked by a single buffered GO channel.

In the GO implementation, GEN2 does not exist as an explicit goroutine. Its functionality was realised by mapping the CSP$_M$ subprocess GEN1 to a GO function,

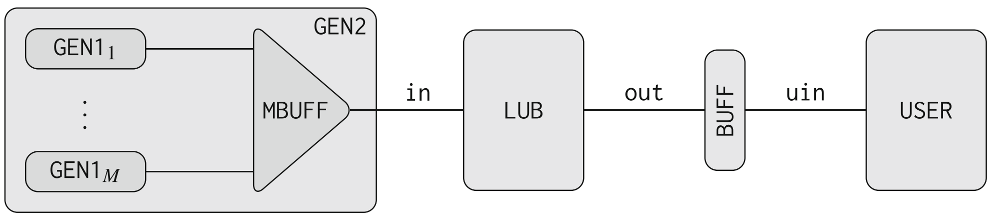

Fig. 2 Diagram of System

Gen1, and then launching multiple instances of Gen1 as separate goroutines from within a loop of GO's main program. All Gen1 goroutines output to the *same* buffered channel. That channel, in turn, is used for input by the GO implementation of the LUB process. The CSP_M multiplexing buffer process, MBUFF, is therefore redundant in the GO implementation.

The parameters of Gen1 specify, *inter alia*, the number of integers to be generated, the range within which they should fall, and the channel to which they should be sent. Based on these parameters, Gen1 generates and outputs random integer values from a given uniform distribution. Between every generation of a random number, the function sleeps for a short period.

The two different GO versions of the LUB process were Lub3 and Lub4 respectively. Lub3 corresponds to the previously given CSP_M process LUB3. Lub4 corresponds to a CSP_M process, LUB4, whose specification—though omitted above in the interests of brevity—had been verified to be a trace refinement of LUB3.

In accordance with our heuristics, Lub3 was implemented using a select statement that has two cases: one for receiving a new bid; and another for emitting the current LUB on the output channel. Note that the Lub3 implementation does indeed conform to Sect. 3's requirement that, upon request from the user of the system it is to return the most recently computed LUB. However, it would fail to meet a more stringent requirement, namely that the USER process should be able to record *every* LUB value as the sequence of bids evolves over time.

The Lub4 process meets this requirement by requesting a new bid from its inbound channel only *after* not only processing the previous bid, but also communicating the resulting LUB to its outbound channel. An outline of Lub4's implementation is given in lines 1–22 of Fig. 3. The for-loop header in line 3 reads a bid from the input channel, in, and lines 4–15 of the loop's body update local data structures (bids and ubids) according to the logic already worked out in the CSP_M specification. Lines 16–20 emit resulting LUB information on the out channel.

The for construct of line 3 allows for simple but graceful termination of Lub4. When all the GEN1 processes have terminated, an anonymous goroutine-(see below) is prompted to close the in channel. The for loop is exited (at line 22) if and only if the buffer of channel in is both empty *and* the channel has been closed. Code in the omitted sections (line 2) ensures that channel out will be closed before Lub4 terminates.

The main function of the GO program sets up the process network and dispatches the goroutines. It is shown in lines 24–44 of Fig. 3.

Lines 28–31 show the launching of multiple instances of the Gen1 function, all parameterised to send output to the same channel, namely bids.

Line 34 launches an anonymous function as a goroutine. Without showing the details of this code, we note that its task is to close the bids channel when all Gen1 processes have shut down. The Lub3 implementation requires a modification in this function so that it receives a sentinel from this process, sends the sentinel downstream and then terminates.

```
 1  func Lub4(in <-chan int, out chan<- int) {
 2     ...
 3     for bid := range in {
 4        if ubids.Len() <= 0 { // No unique bid
 5           if bids.Contains(bid) {// Change nothing
 6           } else {
 7              bids.Insert(bid); ubids.Insert(bid)
 8           }
 9        } else { // At least one unique bid
10           if bids.Contains(bid) {
11              ubids.Remove(bid)
12           } else {
13              bids.Insert(bid); ubids.Insert(bid)
14           }
15        }
16        if ubids.Len() <= 0 {
17           out <- NULL
18        } else {
19           out <- ubids.GetHead()
20        }
21     }
22  }
23
24  func main() {
25     // Instantiate channels.
26     bids := make(chan int, 10); clubs := make(chan int, 10)
27     ...
28     for i := 0; i < GENS; i++ { // Start all instances of Gen1.
29        wg.Add(1)
30        go Gen1(bids, BIDSPERGEN, LOW, HIGH, rng, &wg)
31     }
32
33     // Close channel bids when all Gen1 instances terminate
34     go func(w *sync.WaitGroup) {...}
35
36     go Lub4(bids, clubs) // Start the LUB process.
37
38     // The USER process.
39     for current := range clubs { // Receive a LUB
40        fmt.Println(current) // Use a LUB
41     }
42  }
```

Fig. 3 Extracts of GO code

Line 36 launches `Lub4` using `bids` and `club` as the actual parameters for `Lub4`'s formal parameters `in` and `out` respectively.

Lines 38–41 implement the CSP$_M$ `USER` process's functionality. The `for`-loop reads repeatedly from channel `clubs` and then sends out the received data to the console. When `Lub4` closes `clubs` and its buffer is cleared, then the loop, and consequently also `main`, terminate.

6 Results

Though the GO implementation described above is no more than a prototype for a more comprehensive simulator for the LUB game, it can already be deployed to discover characteristics of the game of potential interest to someone wanting to use it for fund generation.

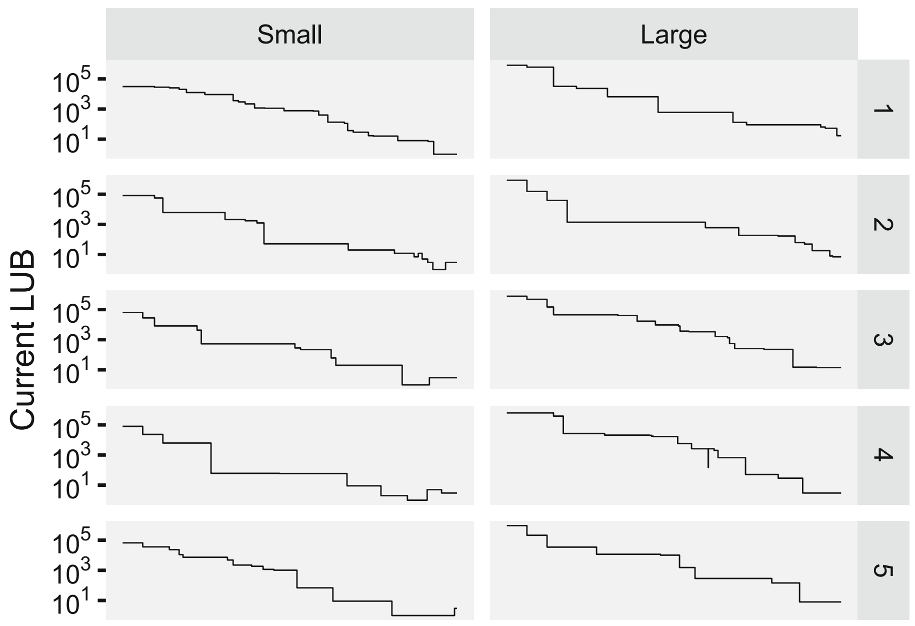

Fig. 4 LUBs over time for five runs for two scenarios of the simulation

As an example of its use, we carried out an experiment to find out how the LUB varies over time as the number of random bids increases. Ten `Gen1` goroutines, each generating 10^5 bids, were launched. Two scenarios dubbed 'Small' and 'Large', respectively, were considered. For the 'Small' scenario, random values were drawn from the interval $[1; 10^5)$ and for the 'Large' scenario the interval was $[1; 10^6)$. In each case, `Lub4` was used to determine the LUBs. Figure 4 shows graphs of the results of five runs of the implementation for the two respective scenarios.

The graphs neatly illustrate that, while the LUB tends to decline as increasingly more bids come in, the decline is not always monotonic. The graphs also show that the possibility of the LUB increasing is greater under the 'Small' scenario than under the 'Large'.

Clearly, similar experiments could investigate the impact of random data that is not drawn from a uniform distribution, different intervals could be used, expected income for different modes of charging for bids could be computed, etc. Such information could be used for decision support with respect to the size of the prize on offer, the expected amount of income to generate, etc.

7 Reflections

It is well-known that "...errors generate significant software development costs if their resolution requires system redesign ..." [17]. The risk of needing to redesign a system will be mitigated by using a specification language to articulate, analyse and explore different architectural solutions before committing to an implementation in code. That, at least, has been the assumption underlying the work described here and elsewhere.

In order to *effectively* support such analysis and exploration in the context of *concurrent* software development, the specification language has to be able to express the notions of concurrent processes and their interactions. For the language to do so *efficiently*, it has to allow for abstraction away from irrelevant details. But there is a certain tension between expressiveness and abstraction: too much abstraction might elide over relevant details; too much expression of detail risks bogging the user down in time-consuming matters better addressed during implementation. Language support for refinement is intended to provide a structured pathway from an efficiently produced abstract specification to a more effective concrete specification.

In previous work [13–15], we used CSP to derive appropriate software architectural specifications, and then mapped the specifications to GO implementations. However, in the interests of efficiency, we used CSP in an informal, *lightweight* fashion. This meant that we ignored specification details such as types and ranges; where needed, we simply assumed the existence of utility functions instead of specifying their details; refinements were informal and were made *in situ*, as it were, by simply overwriting parts of an earlier version of a specification until we deemed it to be sufficiently detailed; etc. Even so, this approach to *efficiently* (ab)using CSP proved to be sufficiently *effective* for our purposes in that context.

The present study represents our first practical exposure to FDR. For this reason we chose an illustrative problem of limited scope for study. Although this limits the inferences we can draw from using the tool, we nevertheless consider that some valuable lessons have been learned, both in regard to the advantages of specification prior to implementation in general, and in regard to perceived strengths and weaknesses of FDR.

Unsurprisingly, FDR requires attention to details that we previously ignored. For example, precise type and range specifications are needed for variables. In previous exercises, we would simply assume that utility functions such as `insert` and `remove` exist (for inserting a value into its sorted position in a sequence or removing a value from a sequence) and reference them in a specification; in FDR, we had to write (and therefore also debug) such functions. In return for this additional effort, FDR allowed us to automatically verify assertions about refinements, deadlock freedom and divergence freedom. While the value-add of being able to automate these verifications might not be particularly spectacular for our relatively small problem, being able to do this for larger problems could be very useful indeed.

An important prerequisite for successfully evaluating these FDR assertions was to ensure that the state space to be searched was kept sufficiently small. For example, we verified that the system is deadlock-free if there are three independent bid generators, each generating exactly two random bids, and we inductively infer that `SYSTEM` is generally deadlock free for any number of generators and bids. It is not self-evident that such inductive inferences are *always* valid. Furthermore, we found that FDR rapidly overloads as the problem size increases. The question of how well

FDR would scale up in the face of a much larger "industrial scale" problems is an issue that requires further investigation.

What has been our constant experience, though, is that the intellectual effort put into articulating a system in CSP (whether or not for subsequent FDR analysis) offers significant returns: relevant processes and their interactions are identified, leading to an inherently modular architecture; various potential problems and pitfalls in the actual implementation may be suggested; alternative architectural or algorithmic possibilities may suggest themselves; the need to reformulate the initial functional specification might be realised; etc. Here is a non-exhaustive list of some such fruitful ideas that arose while developing the CSP$_M$ specifications for the current problem:

- The need for a modular pipelined architectural structure having a multiplexed FIFO buffering function between GEN and LUB was apparent from the start. However, the need for a FIFO buffer between LUB and USER was unanticipated, emerging only when we revised the functional specification.
- The initial impulse for finding the LUB of an incoming data stream was to maintain a list of received values in ascending sorted order, and to traverse the list from the bottom upwards whenever a LUB is requested. While refining LUB1, an innovative notion came to mind: store on separate lists a *single exemplar* of each bid received, as well a list of each *unique* bid received. For each new bid, update both these lists appropriately. LUB3 then introduced a further refinement of maintaining the unique bid list in ascending order so that its head is, by default, the LUB. (Note that LUB3 left as an implementation detail the decision about whether or not to maintain the list of received bids in ascending order.)
- While contemplating the LUB3 solution, the question naturally arose: what if we wanted to capture *every* LUB generated during a simulation? This led to a change in functional requirements as well as to the formulation of LUB4 and implementation of its counterpart in Go, Lub4. The implementation example in Sect. 5 illustrated the interesting information that could be gleaned from this change in functional specification.
- While specifying LUB at various refinement levels, the matter of graceful termination also naturally arose. The specification of LUB1T and its refinements indicated one approach, subsequently implemented by relying on a sentinel. In the case of Lub4, a for...range loop provided a neat implementation solution.

Our collective experience in using CSP and FDR (only partially outlined above) provides *prima facie* evidence to support the incorporation of an early phase of abstract specification in a comprehensive concurrent software development methodology. In some circumstances, CSP used in lightweight fashion appears as a suitable candidate for such specification. Elsewhere there are claims that "FDR has made a significant impact ... across industrial domains, such as high-tech manufacturing, telecommunications, aerospace, and defence"[3]. Yet others report on the benefits of generating CSP traces for test cases in the software development life-cycle [1].

Notwithstanding such evidence, it would be difficult to prove empirically to a cynic that the benefits attributed to the use of CSP would not accrue in any case, even if a hack-and-attack approach has been taken to writing the software. A meta-study to settle such issues would be valuable.

References

1. Gustavo Carvalho et al. “NAT2TEST tool: From natural language requirements to test cases based on CSP”. In: *Software Engineering and Formal Methods* Springer, 2015, pp. 283–290.
2. Thomas Gibson-Robinson et al. “FDR: From Theory to Industrial Application”. In: *Concur rency Security and Puzzles: Essays Dedicated to Andrew William Roscoe on the Occasion of His 60th Birthday* Ed. by Thomas Gibson-Robinson, Philippa Hopcroft, and Ranko Lazicć Cham: Springer International Publishing, 2017, pp. 65–87. ISBN: 978-3-319-51046-0. DOI: 10.1007/978-3-319-51046-0_4. URL: https://doi.org/10.1007/978-3-319-51046-0_4
3. Thomas Gibson-Robinson et al. “FDR3 — A Modern Refinement Checker for CSP”. In: *Tools and Algorithms for the Construction and Analysis of Systems.* Ed. by Erika Ábrahám and Klaus Havelund. Vol. 8413. Lecture Notes in Computer Science. 2014, pp. 187–201.
4. Hassan Gommaa. *Software Design Methods for Concurrent and Real-Time Systems* Addison-Wesley Professional, 1993.
5. C. A. R. Hoare. *Communicating Sequential Processes*. Ed. by Jim Davis. (Electronic version). 2004. URL: http://www.usingcsp.com/cspbook.pdf (visited on 09/16/2016).
6. C. A. R. Hoare. “Communicating sequential processes”. In: *Communications of the ACM* 26.1 (1983), pp. 100–106.
7. J. Magee and J. Kramer. *Concurrency: State models and Java Programs*. 2nd ed. John Wiley, 2006.
8. Susan Owicki and David Gries. “An axiomatic proof technique for parallel programs I”. In: *Acta Informatica* 6.4 (Dec. 1976), pp. 319–340. ISSN: 1432-0525. DOI: 10.1007/BF00268134. URL: https://doi.org/10.1007/BF00268134.
9. Carl G. Ritson and Peter H. Welch. “A Process-Oriented Architecture for Complex System Modelling”. In: *Concurrency and Computation: Practice and Experience* 22 (Mar. 2010), pp. 182–196. DOI: 10.1002/cpe.1433 URL: http://wwwcs.kent.acuk/pubs/2010/3066.
10. A. W. Roscoe. *Understanding Concurrent Systems*. 1st. New York, NY, USA: Springer Verlag New York, Inc., 2010. ISBN: 9781848822573.
11. Marlene Maria Ross. “Unity-inspired object-oriented concurrent system development”. PhD thesis. University of Pretoria, 2001.
12. Adam T. Sampson. “Process-oriented Patterns for Concurrent Software Engineering”. D.Phil thesis. University of Kent, 2008.
13. Marthinus David Strauss. “Process-based Decomposition and Multicore Performance: Case Studies from Stringology”. PhD thesis. University of Pretoria, 2017.
14. Tinus Strauss et al. “A Process-Oriented Implementation of Brzozowski’s DFA Construction Algorithm”. In: *Proceedings of the Prague Stringology Conference 2014, Prague Czech Republic, September 1–3, 2014*. Ed. by Jan Holub and Jan Zdárek. Department of Theoretical Computer Science, Faculty of Information Technology, Czech Technical University in Prague, 2014, pp. 17–29. ISBN: 978-80-01-05547-2.
15. Tinus Strauss et al. “Process-Based Aho-Corasick Failure Function Construction”. In: *Communicating Process Architectures 2015. Proceedings of the 37th WoTUG Technical Meeting 23–26 August 2015, University of Kent, UK*. Ed. by Kevin Chalmers et al. Open Channel Publishing Ltd., 2015, pp. 183–206. ISBN: 0993438504. URL: http://wotug.org/cpa2015/programme.shtml.

16. Peter H. Welch and Jan B. Pedersen. “Santa Claus: Formal Analysis of a Process-oriented Solution”. In: *ACM Transactions on Programming Languages and Systems* 32.4 (Apr 2010), 14:1–14:37. ISSN: 0164-0925. DOI: 1.1145/1734206.1734211 URL: http://doi.acm.org/10.1145/1734206.1734211.
17. J. Christopher Westland. “The cost of errors in software development: evidence from industry”. In: *Journal of Systems and Software* 62 (2002), pp. 1–9.