CLEI electronic journal, Volume 26, Number 1, Paper 6, May 2023

On the specification and verification of the PCR parallel programming pattern in TLA$^+$

José E. Solsona[1*], Álvaro Tasistro[2], Sergio Yovine[1]
Universidad ORT Uruguay, Facultad de Ingeniería,
Montevideo, Uruguay, 11100,
{*solsona,yovine*}*@ort.edu.uy*[1]
alvaro.tasistro@fi365.ort.edu.uy[2]

Abstract

Physical limitations in processor design have made computer industry shift from improving the speed of a single processor to increasing the number of processing core units. But the design of software to exploit parallel processing power in a correct and cost-effective way is a challenging task requiring a high degree of expertise. In 2017, Pérez and Yovine proposed a platform-agnostic parallel programming pattern called PCR, that eases writing parallel code. In this work, we formalize the PCR pattern in terms of TLA$^+$ —a formal specification language for concurrent systems that is being used at places such as Intel, Amazon and Microsoft. We seek to provide a formal framework mainly for (1) expressing high level PCR designs and prove their functional correctness in the sense that their parallel computation computes a given mathematical function, and (2) being able to formally relate different PCR designs. In this way, we contribute to the state of the art in formal verification of parallel programs by leveraging TLA$^+$-related tools to proving properties about high-level PCR-based algorithms such as their functional correctness and refinement.

Keywords: Parallel algorithms, Design patterns, Formal Methods, TLA+

1 Introduction

Physical limitations in processor design have made computer industry move, since 2005, from improving the speed of a single processor to increasing the number of processing core units [1]. This paradigm shift raises for software engineering the challenge of providing appropriate tools for effectively building software that correctly and efficiently exploits parallel processing power. Besides taking care of the well-known pitfalls of concurrent programming, such as deadlocks, livelocks and data races, which are the cause of numerous bugs, developing software for multicore hardware demands considering different execution models, being aware of certain hardware characteristics, and integrating legacy code which cannot always be easily or simply rewritten from scratch. This complexity makes the engineering of correct and efficient parallel software require a high degree of expertise.

In [2], Pérez and Yovine argued that a high-level, platform-agnostic approach based on design patterns would help tackling those difficulties. However, they also noted that most of the existing approaches to pattern-based design lacked formal semantics, which undermined the possibility of adequately tackling the correctness aspect. Hence, they proposed the PCR parallel programming pattern, for which they gave a semantics based on FXML —a theoretical formal language that uses partial orders for interpreting parallel computations [3], and showed how it could be implemented in a concrete execution model. They were able to support through empirical evidence that it was possible to generate performant parallel code based on the mentioned approach. Regarding correctness, although they proved the transformations made by a prototype tool correct, they were not concerned with proving the correctness of the PCRs themselves. Besides, FXML has no associated tools that could be used to assist on formal verification.

*The research that gives rise to the results presented in this publication received funds from the National Research and Innovation Agency under the code POS_NAC_2018_1_152201.

In this thesis, we seek to provide a formal framework mainly for (1) expressing high level PCR designs and prove their functional correctness in the sense that their parallel computation computes a given mathematical function, and (2) being able to formally relate different PCR designs. In particular, that a PCR with "more parallelism" implements another PCR with "less parallelism", i.e., the former is a refinement of the latter. For practical reasons, we wish to use off-the-shelf tools for mechanical verification of the aforementioned properties, and we believe it especially valuable that these tools include some form of automated verification (e.g., model checking) which enable a more light-weight but arguably still rigorous methodology appropriate for industry practitioners.

The rest of this document is structured into four sections. They are: (2) where the PCR parallel programming pattern is informally introduced, together with an example of its use; (3) where we contribute an abstract model for general classes of PCR computations for which the expected input/output relation is established. Again, we discuss here some concrete instances; (4) where we formalize and verify our model in the TLA$^+$ framework, including input/output semantics (i.e., what the PCRs compute) as well as their concurrent semantics (i.e., how the computation is carried out), and also describe our formal verification methods concerning correctness (i.e., safety and liveness) and refinement of PCRs; finally, in (5) we conclude with some general remarks.

2 A primer on the PCR Pattern: Producer, Consumer and Reducer

Our main concern is the design of correct parallel algorithms, in order to actually attain the potential benefits of parallel computation. We follow a pattern-based approach, according to which parallel algorithms should be developed on the basis of some specific and well understood design patterns amenable to formal analysis. The PCR pattern [2] describes computations performed concurrently by communicating *Producers*, *Consumers* and *Reducers*, each one being either a basic function – i.e., a user provided function implemented in some host language – or a nested PCR. Here, we present an informal and illustrative explanation of the PCR pattern.

2.1 High level overview

The PCR pattern aims at expressing computations consisting of a *producer* consuming input data items and generating, for each one of them, possibly many outputs which are consumed by several *consumers* working in parallel. The outputs of them all are aggregated into a single result by a *reducer*.

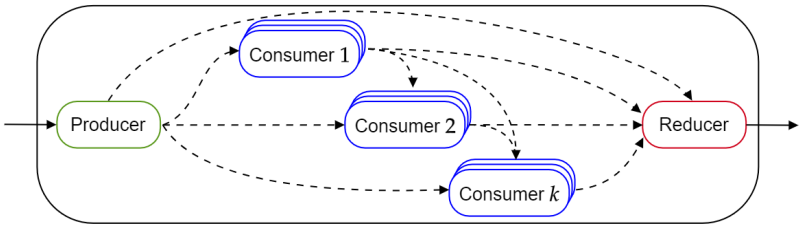

Figure 1: Pictorial view of the PCR pattern.

The goal of the pattern is to emphasize the independence between the different computations, in order to expose all parallelization opportunities. Figure 1 depicts the general form of a PCR. Arrows represent data connections in a PCR. The full ones model the external input source and the output channel to the external environment. The external input is actually available to any inner component. Dashed arrows denote internal data channels. Data cycles between internal components are not allowed: the network is itself a directed acyclic graph (DAG) of which any topological sorting has the producer and the reducer as the first and the last items, respectively. Reads in data channels are non destructive; the same value can be read by any consumer and by the reducer.

Information flow inside a PCR is as follows: (1) For each input data item, the *producer* component generates a set of output values, each one being immediately available for reading. (2) *Consumer* components read values from the outer scope and from the private data channels to perform their computations. (3) A *reducer* component combines values from one or more data sources coming from the producer and one or more consumers, generating a single output item for every external input item processed by the producer.

Producer, consumers, and reducer work in parallel subject to the existing data dependencies: all input items must be available for a producer, consumer or reducer instance in order to perform its calculation. Each producer, consumer and reducer can potentially spawn as many parallel execution instances as necessary for

any specific workload. Both the nature of an execution instance (local and/or remote thread or process) and the scheduling policy are assumed to be defined by the underlying implementation.

PCRs combine several parallel programming concepts. Both the PCR as a whole and each internal consumer as a unit can be regarded as a *parallel map* operation transforming the input stream. Putting several consumers in sequence conforms a *parallel pipeline* with each consumer as a worker. Depending on which implementation is chosen and on the nature of the computation, the reducer could perform a *parallel reduce* operation. Also, the reducer could finish its computation before all its input items are processed or even before the producer or the consumer instances finish their work, enabling for so called *eureka* computations. The sequence of first the producer output triggering the spawning of consumer instances, and then the aggregation of consumers' outputs by the reducer can be regarded as an instance of the *Fork/Join* model, having the reducer as the synchronization point. PCRs combine *collective* operations into a single composable high-level pattern: the distribution of the producer output to instances of the same consumer is a *scatter* operation; availability of the same produced item to all consumers is analogous to a *broadcast*; the combination of all inputs by the reducer is a *gather* operation.

2.2 Example: counting the first prime Fibonacci numbers

Let us consider the problem of counting the prime numbers among the first N Fibonacci numbers. Figure 2 illustrates pictorially how two PCRs may cooperate in a compositional fashion to resolve this problem. The

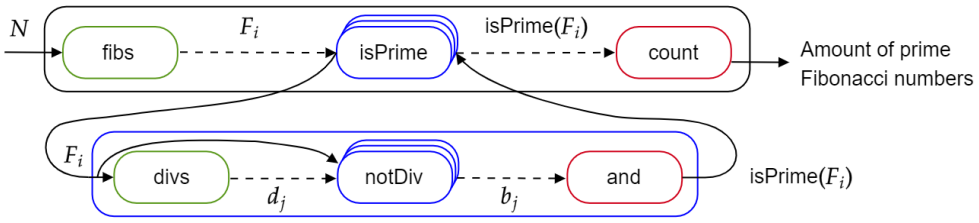

Figure 2: PCR solution for counting Fibonacci primes.

PCR solution is intended to work as follows:

1. At the *outer* PCR, the producer `fibs` generates the sequence F_1, F_2, ..., F_N of Fibonacci numbers.
2. For each F_i, an independent instance of consumer `isPrime` can check, in parallel, its primality generating a boolean result `isPrime(`F_i`)`. This constitutes a *nesting opportunity*, in which another *inner* PCR may paralellize the sub-task of primality testing as follows: (2.1) The producer `divs` generates all possible divisors d_j of F_i. (2.2) For each d_j, an independent instance of consumer `notDiv` can check, in parallel, the (non-)divisibility of F_i by d_j generating a boolean result b_j. This is an example of a consumer reading the producer output and the PCR input as well. (2.3) The reducer `and` computes the result `isPrime(`F_i`)` as the conjunction of all the consumer outputs b_j.
3. The reducer `count` counts the number of consumer outputs `isPrime(`F_i`)` which are true.

This solution admits parallel execution at several levels. Many instances of `isPrime` could be executed simultaneously as allowed by the available processing units and the F_i production rate. Besides, each instance may be computed by an inner PCR with its own parallel capabilities. Since the `count` and `and` reduce operations are associative (and commutative), they can also be parallelized. Indeed, the consumer `notDiv` can also be considered as a nesting opportunity where another PCR may check divisibility in parallel, although performance-wise this would possibly only be worthwhile for very large numbers.

3 Abstract model for the PCR pattern

We now introduce an abstract model for PCR, as well as an associated syntax. This model should be faithful to the informal description, serving as a rigorous basis to reason about PCRs but simple enough to abstract away from details like hardware architecture, interleaving vs no-interleaving, task scheduling, or reducer implementation strategies. Here we aim at clearly distinguishing, in the most abstract and elementary way possible, *how* a PCR computes (the concurrency) from *what* the PCR computes (the expected input/output).

3.1 The basic PCR

The PCR computation is governed by an *iteration space*, a possibly input dependent set of indices (natural numbers) on which the internal components should act. More precisely, the iteration space indexes data produced and consumed by the components of the PCR (e.g., for the Fibonacci example is $\{1, 2, \ldots, N\}$). The result of each internal component operation at every index is written to an associated *output variable*. Specifically, output variables for producer and consumers describe the *full history* of assignments over the iteration space, modeling the internal data channels of the PCR. We thus think of these variables as *streams* of values mathematically represented by partial functions $\mathbb{N} \to T$, abbr. $\vec{T}$, for some range type T. If v is a stream variable, we denote its i-th element by v^i and by $wrt(v^i)$ the condition that v^i has been *written*. Writing to a stream variable may happen in any order. The stream-like nature of variables can be leveraged into a syntactic mechanism which allows stream operations *look-ahead*/*look-behind* to be used on variables by indexing (e.g., to compute the i-th Fibonacci number it is performance-wise useful to have access to the previous computed values at $i-1$ and $i-2$). The following is the general scheme for a *basic* PCR.

Definition 3.1 (Basic PCR scheme)**.** Let $\mathcal{A}$ be a basic PCR with k consumers. The computation of $\mathcal{A}$ over input x consists in the following operations (assignments) for each $i \in I_x$:

$$\begin{array}{rl} \textbf{Producer}: & p^i := f_p(x, p, i) \\ \textbf{Consumer}\ 1: & c_1^i := f_{c_1}(x, p, i) \\ \vdots & \vdots \\ \textbf{Consumer}\ k: & c_k^i := f_{c_k}(x, p, c_1, \ldots, c_{k-1}, i) \\ \textbf{Reducer}: & r := r \otimes f_r(x, p, c_1, \ldots, c_k, i) \end{array}$$

where:

- $I_x \subseteq \mathbb{N}$ is an index set representing the *iteration space*. In general, it depends on input x and has the form $\{i \in lBnd(x)..uBnd(x) : prop(i)\}$ where *lBnd* and *uBnd* denote its lower bound and the upper bound respectively, and *prop* is a truth condition acting as filter.
- p, c_1, ..., c_{k-1} and c_k are variables representing the *histories* of values computed by producer and consumers. These constitute the output of producer and consumers.
- f_p, f_{c_1}, ..., f_{c_k} and f_r are *basic functions* associated with producer, consumers and reducer. In general, every basic function may consume values from previous output variables, which naturally induces a (strict) partial order representing *data dependencies* between PCR operations.
- $\otimes$ is a binary combiner operation. r is a variable representing the (partial) combined value, the output of the reducer. Its initial value is given by $r_0(x)$, to which we sometimes refer as r_0. Unlike the other components' output variables, r is a scalar (i.e., not a stream variable) and the reduction order with respect to I_x is unspecified.

All assignments, basic functions and reduce operations are assumed to be atomic. Except when explicitly stated otherwise, we assume (1) I_x is finite, (2) the combiner $\otimes$ is an associative and commutative operation, and (3) $\otimes$ has an identity element, namely $\texttt{id}_\otimes$, so that $r_0(x) = \texttt{id}_\otimes$ and this is also the default value in case the iteration space is empty.

Definition 3.2 (Basic PCR Termination)**.** A basic PCR $\mathcal{A}$ *terminates* at input x if the reduce operation has been performed for all $i \in I_x$ (i.e., a total of $\mid I_x \mid$ reductions have been made). We shall write $end_\mathcal{A}$ for this condition.

From the given definitions, it follows that the PCR $\mathcal{A}$'s *output* is the value of r when $end_\mathcal{A}$ holds.

3.2 Basic syntax: *produce*, *consume* and *reduce*

In Table 1, we introduce a simple syntax to describe PCRs. For any producer/consumer stream variable `v`, `v[0]` refers to the value at the current iteration index i (i.e., v^i) and this can be more conveniently written just as `v`.[1] Also, basic functions have optional access to the index parameter i if they need to act upon its value. We do not dwell on the host language's syntax/semantic specifics.

[1] Whenever needed, syntax `v[0]` can be used to make it clear we are specifically referring to the value v^i, as opposed to the complete history of values v.

Type	Syntax	
Basic function	**fun** $f(x_1, x_2, \ldots, x_n)$	
Iteration space	**lbnd** $\mathcal{A}$ = $\lambda x.\ e$ // **default:** 0 **ubnd** $\mathcal{A}$ = $\lambda x.\ e$ // **default:** ∞ **prop** $\mathcal{A}$ = $\lambda i.\ e$ // **default:** *True*	where $\mathcal{A}$ is the PCR's name and **lbnd**, **ubnd**, **prop** are the lower bound, upper bound and filter functions respectively.
Producer	p = **produce** f_p x p	where f_p is a basic function, and x is the PCR's input variable
Consumer	c_e = **consume** f_{c_e} x p c_1 ... c_{e-1}	where f_{c_e} is a basic function ($1 \leq e \leq k$), x is the PCR's input variable, p is the producer's output variable, and c_1, ..., c_{e-1} are the previous consumers' output variables.
Reducer	r = **reduce** $\otimes$ (r_0 x) (f_r x p c_1 ... c_{k-1})	where $\otimes$ is the combiner operation (with initial value r_0) and f_r is a basic function on PCR input and producer/consumer outputs.

Table 1: Basic PCR syntax.

Example 3.1. IsPrime1 is a *basic* PCR with a single consumer. For an input number N, IsPrime1 returns a boolean value indicating if N is a prime number. It works as follows: (1) The producer `divs` generates all the possible divisors of the input number N,[2] and for this it is enough to test divisibility in the range $2..\lfloor\sqrt{N}\rfloor$. (2) Each instance i of the `notDiv` consumer checks, in parallel, the divisibility of N by p^i, resulting in the output of indexed boolean values c^i. (3) The reducer `and` computes the conjunction of those outputs.

```
1  fun divs(i) = i
2  fun notDiv(N,p) = not (N % p == 0)
3  lbnd IsPrime1 = 2
4  ubnd IsPrime1 = λN.⌊√N⌋
5  PCR IsPrime1(N)
6    par
7      p = produce divs
8      c = consume notDiv N p
9      r = reduce and (N > 1) c
```

Listing 1: PCR IsPrime1 code.

In this example, we take the liberty of returning something different from the combiner's identity (i.e., *True*) for small inputs. Notice that, when $N \leq 1$ we have $r = N > 1 = \mathit{False}$, which makes sense for this PCR because numbers 0 and 1 are trivially not prime. This decision may seem a bit awkward, but helps to keep this particular introductory example simpler. □

Now we also add syntax to handle explicit data dependency. In general, for variables v_1, v_2 and indices i, j keyword **dep** $v_1(i) \rightarrow v_2(j)$ means the value of v_2 at j (i.e., v_2^j) depends on the value of v_1 at i (i.e., v_1^i). Recall, `v[0]` (or simply `v`) refers to v^i. To look behind on v at k (i.e., v^{i-k}) the syntax is `v[-k]`, and to look ahead on v at k (i.e., v^{i+k}) the syntax is `v[+k]`. No value can depend on present or future values of the *same* variable, as this situation may introduce deadlock.

Example 3.2. FibPrimes1 is a basic PCR with a single consumer and a producer that looks behind on previous values to compute. For an input number N, FibPrimes1 returns the number of primes among the first N Fibonacci numbers. It works as follows: (1) The producer `fib` generates the sequence F_1, F_2, ..., F_N of Fibonacci numbers. (2) Each instance $i \in 1..N$ of the `isPrime` consumer checks, in parallel, the primality of F_i, resulting in the output of indexed boolean values c^i.[3] (3) The reducer `count` counts the number of those outputs which are true. Notice when $N < 1$, vacuously: $r = 0$, the identity of $+$.

```
1  fun fibs(p,i)  = if i <= 2 then 1 else p[-1] + p[-2]
2  fun isPrime(p) = p > 1 and not (some (λm. divides(m,p)) [2..p-1])
3  fun count(r,c) = r + if c then 1 else 0
4  dep p(i-1) → p(i)
5  dep p(i-2) → p(i)1
6  lbnd FibPrimes1 = 1
7  ubnd FibPrimes1 = λN.N
8  PCR FibPrimes1(N)
```

[2] In fact, producer `divs` is trivial here, it's just the identity function on the iteration space. In practice, there would be no reason to write the producer in these cases.

[3] Here, `isPrime` is a basic function, not a PCR. We discuss PCR nesting later.

```
 9    par
10      p = produce fibs p
11      c = consume isPrime p
12      r = reduce count 0 c
```

Listing 2: PCR FibPrimes1 code.[4]

3.3 What does a PCR compute?

Here we turn our attention to *what* a PCR computes. We are interested in functional behaviour and shall look for an explicit form comprising the basic functions that constitute the PCR. First, let us make a general observation. All PCR operations, except reductions, are assignments of the form $v^i := f_v(x, u_1, u_2, \ldots, u_k, i)$ where $x \in T$, $v \in \vec{T}_v$, $u_j \in \vec{T}_j$ for $1 \leq j \leq k$, $i \in \mathbb{N}$, $f_v : T \times \vec{T}_1 \times \cdots \times \vec{T}_k \times \mathbb{N} \to T_v$, and $T_1, \ldots, T_k, T_v$ are the (range) types of the intervening variables. So, variables v and u_j's are streams while f_v is a function on the u_j's streams (as well as on input x and index i). We can treat f_v also as a stream if we define $\vec{f}_v : T \times \vec{T}_1 \times \cdots \times \vec{T}_k \to \vec{T}_v$ (essentially a curried version of f_v) as:

$$\vec{f}_v(x, u_1, u_2, \ldots, u_k) \;=\; i \in \mathbb{N} \mapsto f_v(x, u_1, u_2, \ldots, u_k, i) \tag{1}$$

so that $v^i = \vec{f}_v^{\,i}(x, u_1, u_2, \ldots, u_k)$ holds for any *written* assignment i, and we do similarly for the u_j's associated functions. This will allow us to express the effect of PCR operations as a functional stream composition originated from basic functions in the *index-free* style $\vec{f}_v(x, \vec{f}_{u_1}, \vec{f}_{u_2}, \ldots, \vec{f}_{u_k})$. Next, we apply this observation and further discuss related matters carrying out a revision of our previous examples.

Example 3.3 (Example 3.1 revisited)**.** Define the stream versions of *divs* and *notDiv* as follows:

$$\begin{aligned} \overrightarrow{divs} &= i \in \mathbb{N} \mapsto divs(i) \\ \overrightarrow{notDiv}(N, p) &= i \in \mathbb{N} \mapsto notDiv(N, p, i) \end{aligned}$$

Let $N \in \mathbb{N}$. Then $p^i = \overrightarrow{divs}^{\,i}$ and $c^i = \overrightarrow{notDiv}^{\,i}(N, p)$ for each $i \in 2..N$. Let $\wedge$ be the combiner **and**. In general, we have a piecewise function for $N \in \mathbb{N}$:

$$IsPrime1(N) \;=\; N > 1 \;\wedge \bigwedge_{i=2}^{\lfloor \sqrt{N} \rfloor} \overrightarrow{notDiv}^{\,i}(N, \overrightarrow{divs}) \tag{2}$$

□

Example 3.4 (Example 3.2 revisited)**.** The producer basic function *fibs* is defined as:

$$fibs(p, i) \;=\; \textbf{if } i \leq 2 \textbf{ then } 1 \textbf{ else } p^{i-1} + p^{i-2}$$

Unlike the previous example, this depends on p^{i-1} and p^{i-2} to compute the i-th value, which is safe due to the presence of the appropriate dependencies. Here, we propose to use instead of *fibs* either the mathematical recurrence or the closed form of the recurrence. The idea is to abstract away the *non-linear* and self-referential dependencies imposed on the producer. Of course, closed forms are not always known or may not even exist, so the second option is not always viable. We normally say this is a *sequential producer*, because the dependence on previous values.

Thus, in general, for any producer function f_p possibly depending on values of the form p^{i-k} for some $k > 0$, we define g_p equal to f_p but replacing every occurrence of the form p^{i-k} by the recursive call $g_p(x, i-k)$. The difference between f_p and g_p is essentially that the former is a *memorized* version of the latter, but they are the same function. For the purpose of deductive formal verification, rewriting the producer function as a pure recurrence seems more convenient.

Now, we define the stream versions of the basic functions *fibs* and *isPrime*, as well for the transformation before reduction: [5]

$$\begin{aligned} \overrightarrow{fibs} &= i \in \mathbb{N} \mapsto fibs(i) \\ \overrightarrow{isPrime}(p) &= i \in \mathbb{N} \mapsto isPrime(p, i) \\ [c] &= i \in \mathbb{N} \mapsto [c^i] \end{aligned}$$

[4] Alternatively, the reducer may be written as: `reduce + 0 (λc. if c then 1 else 0)`

[5] We use Iverson's bracket notation ($[P]$ where P is a predicate) for the characteristic function.

Let $N \in \mathbb{N}$, then we have $p^i = \overrightarrow{fibs}^{\,i}$ and $c^i = \overrightarrow{isPrime}^{\,i}(p)$ for each $i \in 1..N$. In general, for any $N \in \mathbb{N}$:

$$FibPrimes1(N) \;=\; \sum_{i=1}^{N} \Big[\overrightarrow{isPrime}(\overrightarrow{fibs})\Big]^i \tag{3}$$

□

In what follows, we generalize what we did on previous examples for arbitrary basic PCRs.

Proposition 3.1. Let $\mathcal{A}$ be a basic PCR with input parameter of type T and whose reducer variable is of type D. Then, for all $x \in T$, the output of $\mathcal{A}$ at x, to be written $\mathcal{A}(x)$, is determined if $\mathcal{A}$ terminates at input x, and it holds: $\mathcal{A}(x) = \bigotimes_{i \in I_x} \vec{f}_{\mathcal{A}}^{\,i}$ for some function $\vec{f}_{\mathcal{A}} \in \vec{D}$.

Proof. See [4, page 60]. □

In particular, for a single consumer PCR $\mathcal{A}$, Proposition 3.1 expands to:

$$\mathcal{A}(x) \;=\; \bigotimes_{i \in I_x} \vec{f}_r^{\,i}\big(x,\, \vec{g}_p(x),\, \vec{f}_c(x,\, \vec{g}_p(x))\big) \tag{4}$$

Equations 2 and 3 derived in previous examples are special cases of 4 as they have only one consumer. Further clarification is needed for 2, as this assertion strictly holds only for input $N > 1$, because when $N \leq 1$ we have $r = \mathit{False}$ which is not the combiner's identity.

3.4 Composition

PCRs can be composed by hierarchical nesting, which allows reusing components and controlling the desired grain of parallelism, and it can also be helpful in making more manageable the task of correctness verification. According to Proposition 3.1, a basic PCR behaves as a function, thus we intuitively expect nested PCRs to behave as a composition of functions. The definition of termination given before extends naturally to composed PCRs.

3.4.1 Composition through consumer

We start with a motivating example to introduce the basic ideas and then generalize. In PCR FibPrimes1, instead of using the `isPrime` basic function at the consumer we can use our PCR IsPrime1 taking advantage of parallel primality testing.[6] This idea is implemented in a new PCR FibPrimes2 shown in Listing 3. Notice this captures our first informal example illustrated in Figure 2.

```
1  PCR FibPrimes2(N)                    PCR IsPrime1(F)
2    par                                  par
3      p = produce fibs p                   d = produce divs
4      c = consume IsPrime1 p               b = consume notDiv F d
5      r = reduce count 0 c                 a = reduce and (F > 1) b
```

Listing 3: PCR FibPrimes2

Let N be any input for FibPrimes2. For each assignment $i \in 1..N$ there is the i-ith instance of IsPrime1, so there are in general N instances (and N inputs/outputs) of IsPrime1. Consequently, the producer and consumer variables of the inner PCR IsPrime1 are doubly indexed, i.e., by the outer scope assignment $i \in 1..N$ (the father instance) and the inner scope assignment $j \in 2..\lfloor\sqrt{F^i}\rfloor$ where F^i is the i-th Fibonacci number. We therefore denote by $d^{i,j}$ the j-th produced divisor at instance i of IsPrime1 and do similarly for the boolean variable b. Comparing with FibPrimes1 where the consumer is a basic function, we have that each dependence $p^i \to c^i$ is now expanded to the DAG of the i-ith instance of PCR IsPrime1.

Now, *what* does the PCR FibPrimes2 compute? We perform an analysis analogous to the one in example 3.4 for FibPrimes1. There is a stream version of PCR IsPrime1 just like for any basic function:

$$\overrightarrow{IsPrime1}(p) \;=\; i \in \mathbb{N} \mapsto \mathit{IsPrime}1(p, i)$$

where $\mathit{IsPrime}1$ is defined according to equation

$$\mathit{IsPrime}1(p, i) \;=\; p^i > 1 \;\wedge \bigwedge_{j=2}^{\lfloor\sqrt{p^i}\rfloor} \overrightarrow{notDiv}^{\,j}(p^i, \overrightarrow{divs})$$

[6] Here we are assuming that the basic function and the PCR are functionally equivalent.

Thus, in general, for any $N \in \mathbb{N}$:

$$FibPrimes2(N) \;=\; \sum_{i=1}^{N} \left[i \in \mathbb{N} \mapsto \overrightarrow{fibs}^{\,i} > 1 \wedge \bigwedge_{j=2}^{\left\lfloor \sqrt{\overrightarrow{fibs}^{\,i}} \right\rfloor} \overrightarrow{notDiv}^{\,j}(\overrightarrow{fibs}^{\,i}, \overrightarrow{divs}) \right]^{i} \tag{5}$$

In general, indices are sequences of natural numbers. Let I be the index of a particular execution of a PCR. Any child PCR inherits the index of the father and extends its dimension by writing to its producer variable, say p, the (I, i)-th value $p^{I,i}$, for every assignment i according to its iteration space. This multidimensional indexing allows for the concurrent execution of any two instances $I \neq J$ of the child PCR, each one generating its own set of p values, namely $p^{I,i}$ and $p^{J,j}$. The root index of execution is taken to be the empty sequence. Next, we introduce the general scheme for PCR composition through consumer. Elements of distinct PCRs will be identified by subscript notation.

Definition 3.3 (Consumer composition scheme)**.** Let $\mathcal{A}$ and $\mathcal{B}$ be PCRs with k_1 and k_2 consumers respectively. The computation of $\mathcal{A}$ composed with $\mathcal{B}$ through consumer $e \in 1..k_1$, on input x_1, consists in the following operations for each $i \in I_{x_1}$ and $j \in J_{x_2^i}$:

PCR $\mathcal{A}$:

$$\begin{aligned}
p_1^i &:= f_{p_1}(x_1, p_1, i)\\
c_{1,1}^i &:= f_{c_{1,1}}(x_1, p_1, i)\\
&\vdots\\
c_{1,e}^i &:= r_2^i \;\text{ if }\; end_{\mathcal{B}^i}\\
&\vdots\\
c_{1,k_1}^i &:= f_{c_{1,k_1}}(x_1, p_1, c_{1,1}, \ldots, c_{1,k_1-1}, i)\\
r_1 &:= r_1 \otimes f_{r_1}(x_1, p_1, c_{1,1}, \ldots, c_{1,k_1}, i)
\end{aligned}$$

PCR $\mathcal{B}$:

$$\begin{aligned}
x_2^i &:= (x_1, p_1, c_{1,1}, \ldots, c_{1,e-1}, i)\\
p_2^{i,j} &:= f_{p_2}(x_2^i, p_2^i, j)\\
c_{2,1}^{i,j} &:= f_{c_{2,1}}(x_2^i, p_2^i, j)\\
&\vdots\\
c_{2,k_2}^{i,j} &:= f_{c_{2,k_2}}(x_2^i, p_2^i, c_{2,1}^i, \ldots, c_{2,k_2-1}^i, j)\\
r_2^i &:= r_2^i \oplus f_{r_2}(x_2^i, p_2^i, c_{2,1}^i, \ldots, c_{2,k_2}^i, j)
\end{aligned}$$

where:

- $I_{x_1} = \{i \in lBnd_1(x_1)..uBnd_1(x_1) : prop_1(i)\}$ and $J_{x_2^i} = \{j \in lBnd_2(x_2^i)..uBnd_2(x_2^i) : prop_2(j)\}$.
- p_1, $c_{1,1}$, ..., c_{1,k_1}, r_1 are the variables of PCR $\mathcal{A}$, and f_{p_1}, $f_{c_{1,1}}$, ..., $f_{c_{1,k_1}}$, f_{r_1} are the *basic functions* associated to these variables. Similarly, p_2, $c_{2,1}$, ..., c_{2,k_2}, r_2 are the variables of PCR $\mathcal{B}$, and f_{p_2}, $f_{c_{2,1}}$, ..., $f_{c_{2,k_2}}$, f_{r_2} are the *basic functions* associated to these variables.
- $\otimes$ and $\oplus$ are the combiner operations of PCR $\mathcal{A}$ and $\mathcal{B}$ respectively.

Notice there is no basic function $f_{c_{1,e}}$ in PCR $\mathcal{A}$. The value of $c_{1,e}$ at each assignment i comes from the output of the i-th instance of PCR $\mathcal{B}$ (i.e., $\mathcal{B}^i$), which is the *final* value of r_2^i. Here, $\mathcal{B}$ is assumed to be a basic PCR, whose input is written by $\mathcal{A}$ in variable x_2. In general, any x_2^i is a tuple of the form $(x_1, p_1, c_{1,1}, \ldots, c_{1,e-1}, i)$. This means $\mathcal{B}$ have access to father's input, all father's variables till $c_{1,e-1}$ and outer assignment i. So, any basic function of $\mathcal{B}$ can look behind/ahead on the father stream variables subject to appropriate dependencies.

Proposition 3.2. Let $\mathcal{A}$ and $\mathcal{B}$ be PCRs with k_1 and k_2 consumers respectively, so that $\mathcal{A}$ with input parameter type T and reducer variable of type D_1 is composed with $\mathcal{B}$ through consumer $e \in 1..k_1$. Then, for all $x_1 \in T$, the output of $\mathcal{A}$ at x_1, to be written $\mathcal{A}(x_1)$, is determined if $\mathcal{A}$ terminates at input x_1 and it holds: $\mathcal{A}(x_1) = \bigotimes_{i \in I_{x_1}} \overrightarrow{f}_{\mathcal{A}}^{\,i}$ for some function $\overrightarrow{f}_{\mathcal{A}} \in \overrightarrow{D}_1$.

Proof. See [4, page 69]. □

In particular, for two single consumer PCRs $\mathcal{A}$ and $\mathcal{B}$, Proposition 3.2 expands to:

$$\mathcal{A}(x_1) \;=\; \bigotimes_{i \in I_{x_1}} \overrightarrow{f}_{r_1}^{\,i}\Big(x_1,\, \overrightarrow{g}_{p_1}(x_1),\, i \in \mathbb{N} \mapsto \bigoplus_{j \in J_{x_2^i}} \overrightarrow{f}_{r_2}^{\,j}\big(x_2^i,\, \overrightarrow{g}_{p_2}(x_2^i),\, \overrightarrow{f}_{c_2}(x_2^i,\, \overrightarrow{g}_{p_2}(x_2^i))\big)\Big) \tag{6}$$

where $x_2^i = (x_1, \overrightarrow{g}_{p_1}(x_1), i)$. Coming back to our previous example FibPrimes2, Equation 5 is *almost* a special case of 6.[7] Also, we have that $FibPrimes2(N) = FibPrimes1(N)$ for any $N \in \mathbb{N}$ if $IsPrime1(F_i) = isPrime(F_i)$ for all Fibonacci numbers F_i with $i \in 1..N$. In other words, if in the PCR FibPrimes1 we replace the basic function isPrime with the PCR IsPrime1 and both behave the same under the relevant inputs generated by FibPrimes1's producer, we obtain a new PCR which is functionally equivalent to FibPrimes1 but with "more paralellism".

Next we show a final relevant class of applications of the PCR pattern.

[7] The qualification "almost" is because, as was noted earlier, PCR IsPrime1 conforms to proposition 3.1 strictly for input $N > 1$, so there is a small deviation.

3.4.2 Divide and Conquer

Property 3.1 shows that a basic PCR behaves like a function. This allows calling a PCR from any *basic function* in a blocking way, where the caller holds until the call returns. Of course, even if the caller is blocked, the parallelism inside the callee is preserved. Calling a PCR as a function enables recursive parallelism. A prominent use case is *divide and conquer* (DC), which in turn is an special case of the *Fork/Join* pattern.

A PCR scheme for this technique is presented in Listing 4. The producer partitions the original problem into subproblems using the `iterDiv` function. Consumers process each subproblem, either using `base` or recursively calling PCR $\mathcal{DC}$, depending on the result of `isBase`. The reducer uses `conquer` to combine all the subproblems solutions. Here, r_0 is expected to compute the empty subproblem.

```
1  fun div(x), isBase(x,p,i), base(x,p,i), f_r(x,p,c,i), r_0(x)
2  fun iterDiv(x,i) = div(x)[i]
3  fun subproblem(x,p,i) = if isBase(x,p,i) then base(x,p,i) else DC(p)
4  fun conquer(r,x,p,c,i) = r ⊗ f_r(x,p,c,i)
5  lbnd DC = λx. 1
6  ubnd DC = λx. len(div(x))
7  PCR DC(x)
8    par
9      p = produce iterDiv x
10     c = consume subproblem x p
11     r = reduce conquer (r_0 x) x p c
```

Listing 4: Divide and conquer scheme in syntactic form.

Proposition 3.3. Let $\mathcal{DC}$ be a DC PCR with input parameter of type T and reducer variable of type D. Then, for all $x \in T$:

$$\mathcal{DC}(x) \;=\; \bigotimes_{i=1}^{len(div(x))} \overrightarrow{f_r}^{i}(x,\, \overrightarrow{div}(x),\, \overrightarrow{subproblem}(x,\, \overrightarrow{div}(x))) \tag{7}$$

where $\overrightarrow{subproblem}(x,p) \;=\; i \in \mathbb{N} \mapsto \big(isBase(x,p,i) \;\rightarrow\; base(x,p,i),\; \mathcal{DC}(p^i)\big)$.

The nature of the recursive problem decomposition process is captured in Proposition 3.3 using conditional notation mimicking the *subproblem* definition. PCR $\mathcal{DC}$ composes with itself through the consumer depending on the result of *isBase*. However, the child PCR does not have access to the father's stream variables as in the ordinary composition scheme presented before —only the value p^i is passed in the call. In this scheme, the producer component has no dependencies on itself, so all its assignments can be executed in parallel like the consumer.

Example 3.5. MergeSort1 is a DC PCR implementing the very well known *Merge Sort* algorithm for sorting lists. It works as follows: (1) The producer generates two halves, say L_1 and L_2, of the input list L. (2) Instances of the consumer work, in parallel, on each sub-list L_i. If $len(L_i) \leq 1$ then L_i is trivially sorted and is ready to be reduced (deemed a base case), otherwise the procedure starts again on input L_i. (3) The reducer merges the sorted halves of the input list.

```
1  fun div(L)    = let m = ⌊len(L)÷2⌋ in [L[1..m], L[m+1..len(L)]]
2  fun isBase(p) = len(p) <= 1
3  fun base(p)   = p
4  fun ⊎(x,y)
5  PCR MergeSort1(L)
6    par
7      p = produce iterDiv L
8      c = consume subproblem L p
9      r = reduce ⊎ [] c
```

Listing 5: PCR MergeSort1 code. ⊎ is the binary merge operation on lists.

MergeSort1 has a very regular behaviour, for any non base case input there are exactly two subproblems produced. In general, for any list L:

$$MergeSort1(L) \;=\; \biguplus_{i=1}^{2} \big(isBase(\overrightarrow{div}(L), i) \;\rightarrow\; base(\overrightarrow{div}(L), i),\; MergeSort1(\overrightarrow{div}^{\,i}(L))\big) \tag{8}$$

□

4 Formalizing the PCR pattern in TLA$^+$

TLA$^+$ is a specification language for computer systems developed by Lamport [5] that has been successfully used at places such as Intel, Amazon and Microsoft to specify and verify mostly concurrent systems. It can be analyzed into two fragments: (1) The Temporal Logic of Actions (TLA) [6], a variant of Pnueli's original temporal logic that makes it practical to write a specification as a single formula, and (2) a first order theory based on Zermelo–Fraenkel set theory with the axiom of Choice (ZFC) and a *choose* operator.

Within TLA$^+$'s framework, computation is understood as the discrete evolution of *state* described by a temporal logic formula, where the state is formed by logical structures described in set theory based mathematics. In TLA$^+$, both the system and its claimed properties are formulas in the same logic. Consequently, system S *satisfies* property P if and only if $S \Rightarrow P$ is a valid formula of the logic. Moreover, system S_2 is a *refinement* of system S_1 if and only if $S_2 \Rightarrow S_1$ is a valid formula of the logic.

TLA$^+$ possesses good tool support, a necessary feature for industrial use. Most notably, it supports automated verification (on finite models) by model checking with the TLC tool [7] and semi-automated verification by theorem proving using the TLAPS tool [8].[8] Also, there exists an IDE that is very well integrated with the tools and it is generally considered as the preferred way to work with TLA$^+$ [10].

We now present a formalization of the abstract PCR models and the functions they compute in TLA$^+$. There are mainly two themes to be addressed:

1. **The input/output semantics (aka *What?*)**
   PCR input/output behaviour was previously identified with certain functions under some algebraic assumptions. To deal with this formally, we organize the required mathematical concepts, properties and proofs across a collection of TLA$^+$ modules where only *classical* mathematics is used.

2. **The concurrent semantics (aka *How?*)**
   A formal concurrent semantics for PCRs is given in terms of temporal TLA$^+$ formulas. To this end, we carry out a straightforward translation of the PCR operations as *actions* in the TLA$^+$ sense so that a PCR execution can be seen as a (fair) labeled transition system.

These two items are connected in TLA$^+$ by the notion of *partial correctness* (a *safety* property) and *termination* (a *liveness* property), i.e., the concurrent semantics should compute an appropriate function. Additionally, specifications of different PCR models are connected by the notion of *refinement* under appropriate substitutions. The specifications are parameterized for the *basic functions* and other concrete elements that should be provided by the user under normal circumstances. In this way, classes of concrete PCRs (*models* in the formal logic sense) can be represented with a single specification (a *theory*), so that instantiating the specification parameters gives a particular concrete PCR.

4.1 Initial assumptions

We fix hereinafter some ground assumptions about PCRs and their execution. These assumptions are made mostly in the interest of simplicity but some of them are also due to limitations on the tooling support.

We will assume an interleaving execution semantics, which is the most common approach adopted to model concurrency and a reasonable abstraction for most practical purposes.

According to the definitions given in Section 3.1 we allowed the basic functions to read from all the previous variables. A simplification we make now is to read only from the immediately preceding variable. There is no loss of expressiveness, as each basic function could forward the previous values together with its output if it was necessary. Also, we will work only with single consumer PCRs. Again, there is no loss of expressiveness, as we can express the computation of multiple consumers with a single one.

For the sake of uniformity in our specifications, we never assume that the main PCR being specified is located at the root of the execution hierarchy. Recall that, in general, indexes are sequences of natural numbers, formally objects of the form $I \circ \langle i \rangle \in Seq(Nat)$ (which we informally write as I, i) where I is the index of the instance (also the index of the father's instance) and i is the current assignment at the inner scope. So, the point of view we adopt is that the main PCR being specified operates starting from some base index $I_0 \in Seq(Nat)$, which *may* be the empty index $\langle\rangle$ (in which case it could be considered as the root of execution) but otherwise it would mean is located deeper in the execution hierarchy, thus reflecting a bit more general setting.

[8] A more recent alternative is the APALACHE model checker which is based on symbolic techniques instead of the classic explicit enumeration of states. [9]

4.2 Formalizing the mathematics underlying the input/output semantics

Recall Proposition 3.1 states the result of a basic PCR $\mathcal{A}$ on input x as given by the expression $\bigotimes_{i \in I_x} \vec{f}_{\mathcal{A}}^{\,i}$ which in general expands to $\mathtt{id}_{\otimes} \otimes \vec{f}_{\mathcal{A}}^{\,m} \otimes \vec{f}_{\mathcal{A}}^{\,m'} \otimes \cdots \otimes \vec{f}_{\mathcal{A}}^{\,n}$ for some function $\vec{f}_{\mathcal{A}} : \mathbb{N} \to D$ built from the PCR basic functions and assuming $(D, \mathtt{id}_{\otimes}, \otimes)$ is an abelian monoid (or just a monoid when a fixed reduction order is imposed).

Now we need a suitable *formal* definition for this kind of expression in the TLA$^+$ language that should also be manageable by its tools. To this end, we formalize the extension of a binary operation $\otimes$ to finite (not necessarily consecutive) intervals of $\mathbb{N}$ under certain algebraic properties. We call this generalization the *big operation* and denote it by $\bigotimes$.[9] Currently, the standard library of mathematics offered by TLAPS is not very large. However, there is (more than) enough material for our needs.

Our formalization is completely independent of the PCR concept and is organized in several modules that can be found in PCR/BaseModules (GitHub). Some of them are summarized in Table 2.

Module Name	Description
`AbstractAlgebra`	Definitions from elementary abstract algebra.
`MonoidBigOp`	The big operator assuming a monoid structure.
`AbelianMonoidBigOp`	The big operator assuming an abelian monoid structure.

Table 2: Modules of the TLA$^+$ specification for algebraic concepts and their associated theorems.

4.3 Formalizing the abstract PCR models

In this section we present the formalization in TLA$^+$ of the abstract PCR models introduced in Section 3.1. Table 3 summarizes some of the TLA$^+$ modules representing the abstract PCR models. They can be found at PCR/AbstractModels (GitHub). Each module includes the corresponding input/output and concurrent specification. In what follows, we briefly discuss some of this modules, starting with the basic PCR. The other modules can be considered extensions or variations of the basic PCR.

Module Name	Description	Concrete instance
`PCR_A`	Basic PCR. See def. 3.1.	FibPrimes1, IsPrime2
`PCR_ArLeft`	Basic PCR with left reducer over a consecutive iteration space.	
`PCR_A1step`	Basic PCR as a one step computation.	
`PCR_A_c_B`	PCR composed through consumer with basic PCR. See def. 3.3.	FibPrimes2
`PCR_A_r_B`	PCR composed through reducer with basic PCR.	
`PCR_DC`	DC PCR. See section 3.4.2.	MergeSort1, NQueensDC
`PCR_DCrLeft`	DC PCR with left reducer.	Merge
`PCR_DC_r_DCrLeft`	DC PCR composed through reducer with other DC PCR with left reducer.	MergeSort2

Table 3: Modules of the TLA$^+$ specification for the abstract PCR models and concrete instances.

4.3.1 The basic PCR

The main reference for this formalization is the definition of a basic PCR given in 3.1. Elements such as basic functions, types and data dependencies are parameterized by constant symbols. In particular *Dep_pp*, *Dep_pc* and *Dep_cr* are pairs of look behind/ahead sets of the form $\langle \{b_1, b_2, \dots\}, \{a_1, a_2, \dots\} \rangle$.

The state of a basic PCR is represented by the following variables: (1) X : The input of the PCR, a (partial) function from indexes to input type T.[10] (2) p : The output variable of the produce, a function mapping indexes to partial functions from *Nat* to producer type T_p. (3) c : The output variable of the

[9] There is nothing special about symbols $\otimes$ and $\bigotimes$, we may use the pair $\oplus$ and $\bigoplus$ as well. However, it should be noted that $\oplus$ is already taken for the multiset (bag) union operation in the standard TLA$^+$ modules.

[10] We model partial functions in TLA$^+$ as total functions adding a special undefined value.

consumer, a function mapping indexes to partial functions from *Nat* to consumer type T_c. (4) r : The output variable of the reducer, a function from indexes to output type D. (5) *red* : An auxiliary variable to track the reductions done at any moment, a function mapping indexes to functions from *Nat* to *BOOLEAN*. The variables p, c and rs represent the internal behaviour of the PCR, whereas the variables X and r represent the external visible behaviour. Other PCR may want to write on input X or to read from output r. For convenience, the internal variables are curried functions separating the father index $I \in Seq(Nat)$ from the assignment $i \in Nat$. In TLA$^+$ notation, all these variables are accessed with (multiple) square brackets as they are functions (of functions). For internal variables we have, for example, that $p[I]$ denotes the producer stream at index I whereas $p[I][i]$ denotes the value of the i-th assignment of the producer stream at index I. Sometimes we will write them in super-index notation like was customary in Section 3, e.g., p^I and $p^{I,i}$ respectively.

The specification of the input/output semantics provides a formula characterizing the mathematical function associated to the basic PCR. This formula is used in to assert correctness, and is also used to represent the PCR as one step computation in module `PCR_A1step` to prove correctness by way of refinement. Module `AbelianMonoidBigOp` is instantiated with the signature $(D, \texttt{id}_\otimes, \_ \otimes \_)$, and assumption *H_Algebra* asserts it satisfies the laws of an abelian monoid which gives us access to the corresponding theorems for theorem proving. For concrete PCRs, this should be model checked or deductively proved.

$$M \triangleq \text{INSTANCE } AbelianMonoidBigOp$$
$$\text{AXIOM } H\_Algebra \triangleq AbelianMonoid(D, \texttt{id}_\otimes, \_ \otimes \_)$$

The stream versions of the basic functions, noted as $\vec{g}_p$, $\vec{f}_c$ and $\vec{f}_r$ in Section 3.3, are defined as follows:

$$Gp(x) \triangleq [i \in Nat \mapsto g_p(x, i)]$$
$$Fc(x, p) \triangleq [i \in Nat \mapsto f_c(x, p, i)]$$
$$Fr(x, c) \triangleq [i \in Nat \mapsto f_r(x, c, i)]$$

Thus, the formalization of the Function 4 associated to the PCR $\mathcal{A}$ is:

$$A(x) \triangleq M!BigOpP(lBnd(x), uBnd(x), prop, \text{LAMBDA } i : Fr(x, Fc(x, Gp(x)))[i])$$

The specification of the concurrent semantics provides a temporal formula in the so-called canonical form — a conjunction of Safety and Liveness formulae. A main baseline unfair specification, named *Spec* as customary, and its fair version are defined according to the canonical form:

$$Spec \triangleq Init \wedge \Box[Next]_{\langle in, vs\rangle}$$
$$FairSpec \triangleq Spec \wedge WF_{vs}(Step)$$

where $vs = \langle X, p, c, r, red\rangle$. The initial conditions are established by the *Init* state predicate. The reducer output is initially by default the monoid identity $\texttt{id}_\otimes$. Producer and consumer outputs are initially undefined, and the reduction history is initially entirely false.

The *Next* action governs the evolution of the state, and is the disjunction of two possible and mutually exclusive sub actions *Step* and *Done*. Action *Done* detects termination when all well defined instances of the PCR (given by set *WDIndex*) have terminated and system state is unchanged: [11]

$$Done \triangleq (\forall I \in WDIndex : end(I)) \wedge \text{UNCHANGED } vs$$

Action *Step* represents any possible operation of the basic PCR components. For this, operations are modeled by atomic actions, namely P for the producer, C for the consumer and R for the reducer, which are parameterized by the instance index I and the current assignment i. So, *Step* is defined as an (double) existential quantifier which introduces non determinism to model the concurrent execution of possible multiple instances of the basic PCR and their component actions:

$$Step \triangleq \exists I \in WDIndex : \exists i \in It(X[I]) : P(I, i) \vee C(I, i) \vee R(I, i)$$

Table 4 presents the PCR operations as syntax primitives and their corresponding formalization as TLA$^+$ actions. Note that all these actions have in common that they are enabled if they haven't already happened. and their dependencies with respect to a previous component are meet.

Partial correctness and termination is stated as follows:

$$Correctness \triangleq end(I_0) \Rightarrow r^{I_0} = \mathcal{A}(X^{I_0})$$
$$Termination \triangleq \Diamond end(I_0)$$

Formula *Correctness* is the formalization of Proposition 3.1.

[11] For the main PCR being specified, this just means termination at index I_0.

PCR Syntax	**TLA$^+$ action**
p = **produce** f_p x p	$P(I,\, i) \triangleq$ $\land\ \neg wrt(p[I][i])$ $\land\ wrts(p[I],\, deps(X[I],\, Dep\_pp,\, i) \setminus \{i\})$ $\land\ p' = [p \text{ EXCEPT } ![I][i] = f_p(X[I],\, p[I],\, i)]$ $\land$ UNCHANGED $\langle X,\, c,\, r,\, red\rangle$
c = **consume** f_c x p	$C(I,\, i) \triangleq$ $\land\ \neg wrt(c[I][i])$ $\land\ wrts(p[I],\, deps(X[I],\, Dep\_pc,\, i))$ $\land\ c' = [c \text{ EXCEPT } ![I][i] = f_c(X[I],\, p[I],\, i)]$ $\land$ UNCHANGED $\langle X,\, p,\, r,\, red\rangle$
r = **reduce** $\otimes$ $\mathtt{id}_\otimes$ $(f_r$ x $c)$	$R(I,\, i) \triangleq$ $\land\ \neg red[I][i]$ $\land\ wrts(c[I],\, deps(X[I],\, Dep\_cr,\, i))$ $\land\ r' = [r \text{ EXCEPT } ![I] = r[I] \otimes f_r(X[I],\, c[I],\, i))]$ $\land\ red' = [red \text{ EXCEPT } ![I][i] = \text{TRUE}]$ $\land$ UNCHANGED $\langle X,\, p,\, c\rangle$

Table 4: TLA$^+$ specification of the PCR primitives over basic function names.

4.3.2 Composition through consumer

For composition through consumer as given in 3.3, we have two PCRs in one specification: the main PCR $\mathcal{A}$ and the nested basic PCR $\mathcal{B}$. There is a set of constants and variables symbols for each one, and also there are basic definitions associated with each one. To distinguish them, we suffix variables and constants names with numbers 1 and 2 for $\mathcal{A}$ and $\mathcal{B}$ respectively (much like in 3.3), and suffix definitions with the corresponding PCR name.

The concurrent and input/output specification of $\mathcal{B}$ is essentially the same as was explained previously for a basic PCR, so what we call $\mathcal{B}$ in module `PCR_A_c_B` is what we call $\mathcal{A}$ in module `PCR_A`. The only important thing to note about $\mathcal{B}$ here is that its input is controlled by $\mathcal{A}$ and there can be multiple instances of $\mathcal{B}$. Inputs for $\mathcal{B}$ are triplets of type $T2 \triangleq T1 \times StA(Tp1) \times Nat$. Initially, there is no input defined for $\mathcal{B}$ and consequently there is no initial instance of $\mathcal{B}$.

Module `AbelianMonoidBigOp` is instantiated twice, namely $M1$ and $M2$, with the signatures $(D1, \mathtt{id}_\otimes, \_\otimes \_)$ and $(D2, \mathtt{id}_\oplus, \_ \oplus \_)$ for $\mathcal{A}$ and $\mathcal{B}$ respectively. Algebraic assumptions of an abelian monoid are asserted for both. Consequently there is an associated function defined for each PCR:

$$B(x2) \triangleq M2!BigOpP(lBnd2(x2),\, uBnd2(x2),\, prop2,\, \text{LAMBDA } i : Fr2(x2, Fc2(x2, Gp2(x2)))[i])$$
$$A(x1) \triangleq M1!BigOpP(lBnd1(x1),\, uBnd1(x1),\, prop1,\, \text{LAMBDA } i : Fr1(x1, Fc1(x1, Gp1(x1)))[i])$$

where in particular we have $Fc1(x1, p1) = [i \in Nat \mapsto B(\langle x1, p1, i\rangle)]$ so that the function of $\mathcal{A}$ is composed with the function of $\mathcal{B}$. This formalizes the function in Equation 6.

The concurrent specification of $\mathcal{A}$ differs from a basic PCR only in what concerns the consumer component, as his values are computed by the nested PCR $\mathcal{B}$ and not by an atomic basic function. Table 5 presents the formalization of the consumer syntax over a PCR name as a pair of actions $C1ini$ and $C1end$ that can be thought as a call and a return with respect to $\mathcal{B}$.

$C1ini(I,\, i) \triangleq$ $\land\ \neg wrt(X2[I \circ \langle i\rangle])$ $\land\ wrts(p1[I],\, depsA(X1[I],\, Dep\_pc1,\, i))$ $\land\ X2' = [X2 \text{ EXCEPT } ![I \circ \langle i\rangle] = \langle X1[I],\, p1[I],\, i\rangle]$ $\land$ UNCHANGED $\langle X1,\, p1,\, c1,\, r1,\, red1\rangle$	$C1end(I,\, i) \triangleq$ $\land\ \neg wrt(c1[I][i])$ $\land\ wrt(X2[I \circ \langle i\rangle])$ $\land\ endB(I \circ \langle i\rangle)$ $\land\ c1' = [c1 \text{ EXCEPT } ![I][i] = r2[I \circ \langle i\rangle]]$ $\land$ UNCHANGED $\langle X1,\, p1,\, r1,\, red1,\, X2\rangle$

Table 5: TLA$^+$ specification for the **consume** primitive over a PCR.

As the main PCR $\mathcal{A}$ is not basic, its $StepA$ action is accommodated as follows:

$$StepA \triangleq \exists I \in WDIndexA : \exists i \in ItA(X1[I]) : P1(I, i) \lor C1ini(I, i) \lor C1end(I, i) \lor R1(I, i)$$

In general, we could have two (fair) specifications for $\mathcal{A}$ and $\mathcal{B}$ defined as:

$$\begin{aligned} \mathit{FairSpecA} &\triangleq \mathit{InitA} \wedge \Box[\mathit{StepA} \vee \mathit{DoneA}]_{\langle in, vs1\rangle} \wedge \mathit{WF}_{vs1}(\mathit{StepA}) \\ \mathit{FairSpecB} &\triangleq \mathit{InitB} \wedge \Box[\mathit{StepB} \vee \mathit{DoneB}]_{vs2} \wedge \mathit{WF}_{vs2}(\mathit{StepB}) \end{aligned}$$

where $vs1 = \langle X1, p1, c1, r1, red1, X2\rangle$ and $vs2 = \langle p2, c2, r2, red2\rangle$, so that their conjoint specification is their conjunction taking interleaving into account:

$$\mathit{FairSpec} \triangleq \mathit{FairSpecA} \wedge \mathit{FairSpecB} \wedge \Box[vs1' = vs1 \vee vs2' = vs2]_{\langle vs1, vs2\rangle}$$

In order to work with TLC, we transform the conjoint specification to a single canonical formula using logical equivalences which produces a single system formula in the canonical form:

$$\begin{aligned} \mathit{Spec} &\triangleq \mathit{Init} \wedge \Box[\mathit{Next}]_{\langle in, vs1, vs2\rangle} \\ \mathit{FairSpec} &\triangleq \mathit{Spec} \wedge \mathit{WF}_{vs1}(\mathit{StepA}) \wedge \mathit{WF}_{vs2}(\mathit{StepB}) \end{aligned}$$

where *Init* is written in terms of *InitA* and *InitB*, and *Next* is written in terms of *StepA* and *StepB*.

Partial correctness and termination is stated for both PCRs as follows:

$$\begin{aligned} \mathit{CorrectnessA} &\triangleq \mathit{endA}(I_0) \Rightarrow r_1^{I_0} = \mathcal{A}(X_1^{I_0}) \\ \mathit{TerminationA} &\triangleq \Diamond \mathit{endA}(I_0) \\ \mathit{CorrectnessB} &\triangleq \forall\, I \in \mathit{WDIndexB} : \mathit{endB}(I) \Rightarrow r_2^I = \mathcal{B}(X_2^I) \\ \mathit{TerminationB} &\triangleq \Diamond(\forall\, I \in \mathit{WDIndexB} : \mathit{endB}(I)) \end{aligned}$$

The correctness of the overall specification is what we state for the main PCR $\mathcal{A}$, as always relative to I_0, which in turns depends on the correctness of possibly multiple instances of $\mathcal{B}$. However, the proof of the refinement of a basic PCR, only relies on the correctness of $\mathcal{B}$ (and some basic invariants of $\mathcal{A}$). Formula *CorrectnessA* is the formalization of Proposition 3.2.

4.3.3 Divide and conquer PCR (DC)

For the input/output semantics, module `AbelianMonoidBigOp` is instantiated with signature $(D, id, Op(\_,\_))$, and the function 3.3 associated to PCR $\mathcal{DC}$ can be expressed as the recursive function:

$$\begin{aligned} \mathit{DC}[x \in T] \triangleq\ & M!\mathit{BigOp}(1,\ \mathit{uBnd}(x),\ \text{LAMBDA}\ i : \mathit{Fr}(x, [k \in \mathit{Nat} \mapsto \text{IF}\ \mathit{isBase}(x, \mathit{div}(x), k) \\ & \qquad \text{THEN}\ \mathit{base}(x, \mathit{div}(x), k) \\ & \qquad \text{ELSE}\ \mathit{DC}[\mathit{div}(x)[k]]])[i]) \end{aligned}$$

For the concurrent semantics, the consumer component differs from the ordinary consumer composition because here the consumer is writing and reading on the same PCR variables (although in different indexes) and we need to take into account when the *isBase* condition holds. However, the overall idea is pretty much the same. Table 6 presents the formalization of the consumer syntax in a DC PCR in terms of three actions.

The *Step* action for the DC PCR is defined as follows:

$$\mathit{Step} \triangleq \exists I \in \mathit{WDIndex} : \exists i \in \mathit{It}(X[I]) : P(I, i) \vee \mathit{Cbase}(I, i) \vee \mathit{Cini}(I, i) \vee \mathit{Cend}(I, i) \vee R(I, i)$$

4.4 Verification of properties

Each module has a section stating the safety and liveness properties concerning the PCRs being specified in the module and possibly also refinement properties relating to PCRs specified in other modules. Here, safety properties include the partial correctness of the PCR computation and other invariant properties like type correctness, whereas liveness properties are, more specifically, termination properties. In the TLA$^+$ framework, mechanical verification can be done in either of the following ways:

- Model checking with TLC. Modules in Table 3 corresponds to abstract PCR models. Constant symbol parameters need to be instantiated with concrete definitions that can be evaluated, which result in concrete PCRs. This means that model checking can be used as a method of verification for concrete PCRs over finite data types. TLC allows to verify safety, liveness and refinement.[12]

[12] By default, TLC searches for deadlocks. However, the absence of deadlocks does not rule out infinite pathological behaviours like livelock or starvation.

PCR Syntax	**TLA$^+$ actions**

fun subproblem(*x*,*p*,*i*) =
  if isBase(*x*,*p*,*i*)
  then base(*x*,*p*,*i*)
  else $\mathcal{DC}(p)$
...
c = **consume** subproblem *x p*
	$Cbase(I, i) \triangleq$
	$\wedge\ \neg wrt(c[I][i])$
	$\wedge\ wrts(p[I], deps(X[I], Dep\_pc, i))$
	$\wedge\ isBase(X[I], p[I], i)$
	$\wedge\ c' = [c \text{ EXCEPT } ![I][i] = base(X[I], p[I], i)]$
	$\wedge$ UNCHANGED $\langle X, p, r, red\rangle$
	$Cini(I, i) \triangleq$
	$\wedge\ \neg wrt(X[I \circ \langle i\rangle])$
	$\wedge\ wrts(p[I], deps(X[I], Dep\_pc, i))$
	$\wedge\ \neg isBase(X[I], p[I], i)$
	$\wedge\ X' = [X \text{ EXCEPT } ![I \circ \langle i\rangle] = p[I][i]]$
	$\wedge$ UNCHANGED $\langle p, c, r, red\rangle$
	$Cend(I, i) \triangleq$
	$\wedge\ \neg wrt(c[I][i])$
	$\wedge\ wrt(X[I \circ \langle i\rangle])$
	$\wedge\ end(I \circ \langle i\rangle)$
	$\wedge\ c' = [c \text{ EXCEPT } ![I][i] = r[I \circ \langle i\rangle]]$
	$\wedge$ UNCHANGED $\langle X, p, r, red\rangle$

Table 6: TLA$^+$ specification for the **consume** primitive in a DC PCR named $\mathcal{DC}$.

- Assisted theorem proving with TLAPS. For the deductive approach we don't need to instantiate constant symbol parameters. This means verification can be actually done for the abstract PCR model without finiteness restrictions. TLAPS currently allows to verify safety properties and the safety component of refinement properties (i.e., without fairness), liveness properties are ruled out.

We have done theorem proving for the following: (1) The safety properties of the basic PCR. (2) The safety properties of the basic PCR with left reducer. (3) A basic PCR is a refinement of a basic PCR expressed as a one step computation. (4) A basic PCR with left reducer is a refinement of a basic PCR. (5) The composition through the consumer is a refinement of a basic PCR. (6) Most assumptions for the concrete elements of PCR FibPrimes1.

All of the above and everything else (e.g., termination) has been verified by model checking on concrete PCRs. In particular, correctness and termination of the divide and conquer PCR was verified only by model checking. The complete specification of the concrete PCRs are in PCR/Concrete (GitHub), they are just modules instantiating the abstract PCR modules with the concrete elements that characterize them (e.g., types, data dependencies, basic functions, etc.).

Figure 3 illustrates some of the refinements that have been established between the modules summarised in Table 3. It should be added that refinements are not always just direct implications between the specification formulas. In general, refinements exists under appropriate substitutions (i.e., refinement mappings).

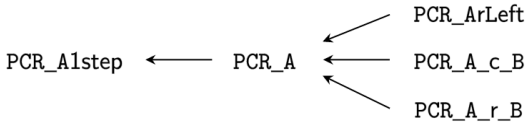

Figure 3: Chains of refinements between some of the abstract PCR models.

5 Conclusions

In future work, we expect to continue doing formal proofs. In particular, we know in next releases of TLAPS it will be possible to deal formally with termination. Besides, we envision an automatic translator tool for PCRs much in the same spirit of PlusCal — an algorithmic language that is translated to TLA$^+$ [11]. Moreover, as executable code extraction from PlusCal is currently being investigated [12], we believe this could benefit us and therefore is not unrealistic to aim for executable and correct PCR programs.

In summary, we think this thesis contributes to the state of the art in formal refinement of parallel programs from abstract models, especially starting off from an alternative characterization of the general PCR pattern, and utilizing the TLA$^+$ framework. We hope to have shown this as an interesting line of research within the general quest for quality programming.

References

[1] K. Asanovic *et al.*, "A view of the parallel computing landscape," *Commun. ACM*, vol. 52, no. 10, p. 56–67, oct 2009. doi: 10.1145/1562764.1562783

[2] G. Pérez and S. Yovine, "Formal specification and implementation of an automated pattern-based parallel-code generation framework," *STTT*, vol. 21, no. 2, p. 183–202, 2017. doi: 10.1007/s10009-017-0465-2

[3] S. Yovine *et al.*, "Formal approach to derivation of concurrent implementations in software product lines," *Algebra for Parallel and Distributed Processing*, pp. 359–401, 2008. doi: 10.1201/9781420064872.CH11

[4] J. E. Solsona, "On the specification and verification of the PCR parallel programming pattern in TLA$^+$," Master's thesis, Universidad ORT, Facultad de Ingeniería, 2021. [Online]. Available: https://dspace.ort.edu.uy/handle/20.500.11968/4555

[5] L. Lamport, *Specifying Systems: The TLA$^+$ Language and Tools for Hardware and Software Engineers*. Redwood City, CA, USA: Addison-Wesley Longman Publishing Co., Inc., 2002. ISBN 032114306X

[6] L. Lamport, "The Temporal Logic of actions," *ACM Trans. Program. Lang. Syst.*, p. 872–923, 1994. doi: 10.1145/177492.177726

[7] Y. Yu *et al.*, "Model Checking TLA$^+$ Specifications," in *Correct Hardware Design and Verification Methods*, 1999. doi: 10.5555/646704.702012 pp. 54–66.

[8] D. Cousineau *et al.*, "TLA$^+$ Proofs," in *FM 2012: Formal Methods*, vol. 7436, 2012. doi: 10.1007/978-3-642-32759-9_14 pp. 147–154.

[9] I. Konnov, J. Kukovec, and T.-H. Tran, "TLA$^+$ Model Checking made symbolic," *Proc. ACM Program. Lang.*, vol. 3, p. 1–30, 2019. doi: 10.1145/3360549

[10] M. Kuppe *et al.*, "The TLA$^+$ Toolbox," *ArXiv*, vol. abs/1912.10633, pp. 50–62, 2019. doi: doi.org/10.4204/eptcs.310.6

[11] L. Lamport, "The pluscal algorithm language," *Theoretical Aspects of Computing-ICTAC*, vol. 5684, pp. 36–60, January 2009. doi: 10.1007/978-3-642-03466-4_2

[12] GitHub, "PGo is a source to source compiler from Modular PlusCal specs into Go programs," [Online]. Available: https://github.com/UBC-NSS/pgo. Accessed on: Oct. 3, 2020.