TLA$^+$ specification of *PCR* parallel programming pattern

José Solsona* Sergio Yovine†

Universidad ORT Uruguay

August 31, 2020

Programming correct parallel software in a cost-effective way is a challenging task requiring a high degree of expertise. In [1], it is proposed a pattern-based formally grounded tool that eases writing parallel code. In particular, the tool is based on a platform-agnostic parallel programming pattern called *PCR*. The *PCR* pattern aims at expressing computations consisting of a *producer* consuming input data items and generating for each of them, a data set to be consumed by several *consumers* working in parallel. Their outputs are finally aggregated back into a single result by a *reducer*. *PCR* emphasize the independence between different computations in order to expose all opportunities for parallelism.

The semantics of *PCR* is given in terms of the formal language *FXML* [2]. However, *FXML* has no associated verification tool. Therefore, our current research goal is to formalize the semantics of *PCR* in terms of TLA$^+$. In this way, we can leverage TLA$^+$ related tools to prove properties. Besides correctness and termination, we are particularly interested in proving *refinement*. Moreover, we will envisage to develop a translator from *PCR* into TLA$^+$ to make the integration seamless. To start up with, we have been working on the formalization of some concrete examples of *PCR* specifications from [1] in the TLA$^+$ specification language. In this presentation, we will discuss our work in progress.

We defined a TLA$^+$ base module that specifies the common skeleton of a *PCR*, that is, all constant definitions. In a *PCR*, variables are streams indexed with multidimensional indexes which are automatically generated by the underlying runtime system. To capture the semantics of this behavior in TLA$^+$ we define *contexts* and *context mappings*. A *context* contains input, output and state variables in the inner scope of the *PCR*. Multidimensional indexes are modeled by sequences of Nat. A *context mapping* maps indexes to contexts. Field v denotes the value of a variable, say x, at index i, with $NULL$ meaning the ith assignment to x has not occured so far. Field r is used to keep track of the number of times it has been read.

$$
\begin{array}{lcl}
VarP & \triangleq & [Nat \rightarrow [v : VarPType \cup \{NULL\},\ r : Nat]] \quad \text{Producer variable type} \\
VarC & \triangleq & [Nat \rightarrow [v : VarCType \cup \{NULL\},\ r : Nat]] \quad \text{Consumer variable type} \\
VarR & \triangleq & VarRType \quad \text{Reducer variable type, i.e, PCR output type}
\end{array}
$$

$$
\begin{array}{lcll}
CtxType & \triangleq & [in\ \ : InType \cup \{NULL\}, & \text{Input} \\
 & & \ \ i\_p : Nat \ \cup \{NULL\}, & \text{Iterator index} \\
 & & \ \ v\_p : VarP, & \text{Producer history} \\
 & & \ \ v\_c : VarC, & \text{Consumer history} \\
 & & \ \ ret\ : VarR, & \text{Reducer result and PCR output} \\
 & & \ \ ste\ : \{OFF,\ RUN,\ END\}] & \text{Discrete state}
\end{array}
$$

A *PCR* has associated an *iteration space* which defines the indexes generated by the *PCR* and appended to the dynamic outer scope index. To cope with iteration spaces, in the TLA$^+$ base module we define an *Iterator* operator which resorts to higher-order operators, namely *Step*, *LowerBnd*, and *UpperBnd*, which have to be explicitly defined when a concrete *PCR* is instantiated.

$$
\begin{array}{lcl}
Iterator(id) & \triangleq & AllFromTo(Step,\ LowerBnd(in(id)),\ UpperBnd(in(id))) \\
Bound(id) & \triangleq & i\_p(id) \in Iterator(id)
\end{array}
$$

A TLA$^+$ specification of a concrete *PCR*, extends a base module. In particular, it must define the actual behavior of the *PCR* *produce*, *consume*, and *reduce* actions, in terms of parameterized TLA$^+$ actions $P(id)$, $C(id)$, and $R(id)$, respectively, where id is the index that identifies the context. The following snippets correspond to part of the Fibonacci Prime Counter *PCR* in [1]: (1) *PCRFibPrimes* generates Fibonacci numbers up to input N, and (2) *PCRIsPrime* acts as a consumer which is dynamically invoked for each Fibonacci number to check its primality.

*solsona@fi365.ort.edu.uy

†yovine@fi365.ort.edu.uy

PCRFibPrimes producer $P(id)$ action generates Fibonacci numbers while $Bound(id)$ holds.

$$
\begin{array}{lll}
P(id) \triangleq & \land\ Bound(id) & \text{// } i\_p(id) \leq N \\
& \land\ map' = [map \text{ EXCEPT} & \text{// Update } PCRFibPrimes \text{ context mapping at index } id. \\
& \quad ![id].v\_p[i\_p(id)] = [v \mapsto fib(v\_p(id),\ i\_p(id)),\ r \mapsto 0], & \text{// } fib(v, i) \triangleq v_{i-2} + v_{i-1}. \\
& \quad ![id].i\_p \qquad = Step(@)] & \text{// } i\_p(id)' = i\_p(id) + 1.
\end{array}
$$

PCRIsPrime consumer $C(id)$ action checks divisor does not divide input (Fibonacci number of index id).

$$
\begin{array}{lll}
C(id) \triangleq & \exists j \in Iterator(id) : & \\
& \quad \land\ Written(v\_p(id),\ j) & \text{// } v\_p(id) \text{ at index } j \text{ has been written, i.e, } v\_p(id)[j] \neq NULL. \\
& \quad \land\ \neg Read(v\_p(id),\ j) & \text{// Producer variable at } j \text{ has not been read.} \\
& \quad \land\ \neg Written(v\_c(id),\ j) & \text{// Cons var at } j \text{ has not been written, i.e, } v\_c(id)[j] = NULL. \\
& \quad \land\ map' = [map \text{ EXCEPT} & \text{// Update } PCRIsPrime \text{ context mapping at index } id. \\
& \qquad ![id].v\_p[j].r = @ + 1, & \text{// Increment read counts of producer variable at index } j. \\
& \qquad ![id].v\_c[j] \quad = [v \mapsto notDivides(v\_p(id)[j].v,\ in(id)),\ r \mapsto 0]] &
\end{array}
$$

There is a *Main* module that instantiates all *PCR* involved in the specification, together with predicate *Init* and action *Next*. The flexible variables map_1 and map_2 are the context mappings for *PCRFibPrimes* and *PCRIsPrime*, respectively.

$$
\begin{array}{ll}
CtxMap1 \triangleq [Seq(Nat) \rightarrow PCRFibPrimes!CtxType \cup \{NULL\}] & \text{// Mapping for } PCRFibPrimes \\
CtxMap2 \triangleq [Seq(Nat) \rightarrow PCRIsPrime!CtxType \quad \cup \{NULL\}] & \text{// Mapping for } PCRIsPrime
\end{array}
$$

$$
\begin{array}{lll}
Init \triangleq & \land\ N \in InType1 & \\
& \land\ map1 = [id \in Seq(Nat) \mapsto & \text{// Computation starts with } PCRFibPrimes \\
& \qquad \text{IF} \quad id = \langle 0 \rangle & \text{// in root context } \langle 0 \rangle \text{ and input } N \\
& \qquad \text{THEN } PCRFibPrimes!InitCtx(N) & \\
& \qquad \text{ELSE } NULL] & \\
& \land\ map2 = [id \in Seq(Nat) \mapsto NULL] &
\end{array}
$$

$$
\begin{array}{lll}
Next1(id) \triangleq & \land\ map1[id] \neq NULL & \\
& \land\ \lor\ \land\ PCRFibPrimes!Off(id) & \\
& \qquad\ \land\ map1' = [map1 \text{ EXCEPT } ![id].ste = RUN] & \\
& \quad\ \lor\ \land\ PCRFibPrimes!Running(id) & \\
& \qquad\ \land\ PCRFibPrimes!Next(id) & \text{// } P(\_),\ C(\_) \text{ or } R(\_) \text{ action} \\
& \land\ \text{UNCHANGED } N &
\end{array}
$$

$$
Next2(id) \triangleq \ \ldots \qquad \text{// Analog to } Next1
$$

$$
\begin{array}{lll}
Next \triangleq & \lor\ \exists\, id \in Seq(Nat) : Next1(id) & \text{// } PCRFibPrimes \text{ step} \\
& \lor\ \exists\, id \in Seq(Nat) : Next2(id) & \text{// } PCRIsPrime \text{ step} \\
& \lor\ Done &
\end{array}
$$

Correctness, termination, and refinement theorems are specified in the *Main* module.

$$
\begin{array}{ll}
Solution(in) \triangleq & \text{LET } fibValues \triangleq \{Fibonacci(n) : n \in \{m \in Nat : m \leq in\}\} \\
& \text{IN} \quad Cardinality(\{f \in fibValues : isPrime(f)\})
\end{array}
$$

$$
\begin{array}{l}
Correctness \triangleq \Box(PCRFibPrimes!Finished(\langle 0 \rangle) \Rightarrow PCRFibPrimes!Out(\langle 0 \rangle) = Solution(N)) \\
Termination \triangleq \Diamond PCRFibPrimes!Finished(\langle 0 \rangle)
\end{array}
$$

In the presentation we will provide further details of the specification, give insights on the approach, discuss stating and model-checking refinements, and sketch future work towards automatically generating TLA$^+$ specifications from general *PCR*.

References

[1] G. Pérez and S. Yovine. Formal specification and implementation of an automated pattern-based parallel-code generation framework. *STTT*, 2017.

[2] S. Yovine, I. Assayad, F. Defaut, M. Zanconi, and A. Basu. Formal approach to derivation of concurrent implementations in software product lines. *Algebra for Parallel and Distributed Processing*, pages 359–401, 2008.