Thread-Local, Step-Local Proof Obligations for Refinement of State-Based Concurrent Systems *

Gerhard Schellhorn, Stefan Bodenmüller, and Wolfgang Reif

Institute for Software and Systems Engineering
University of Augsburg, Germany
{schellhorn,stefan.bodenmueller,reif}@informatik.uni-augsburg.de

Abstract. This paper presents a proof technique for proving refinements for general state-based models of concurrent systems that reduces proving forward simulations to thread-local, step-local proof obligations. Instances of this proof technique should be applicable to systems specified with ASM rules, B events, or Z operations. To exemplify the proof technique, we demonstrate it with a simple case study that verifies linearizability of a lock-free implementation of concurrent hash sets by showing that it refines an abstract concurrent system with atomic operations. Our theorem prover KIV translates programs to a set of transition rules and generates proof obligations according to the technique.

Keywords: Refinement, State-Based Concurrent Systems, Thread-Local Proof Obligations, Interactive Verification

1 Introduction

Refinement-based development is a successful approach to the development of algorithms and software systems. An important subcase is the development of efficient, thread-safe concurrent implementations, where the abstract specification is often given as simple atomic operations.

We have developed two approaches for verifying such refinements. One is based on a program calculus, and the other on which we focus in this paper relies on translating programs to a state-based description. This approach requires just predicate logic for verification.

We have done case studies with algorithms that are hard to verify. In particular, some require backward simulation or were hard to reduce to thread-local reasoning [12]. Most cases, however, like the one we consider in this paper, are simpler. We noted that their verification still results in much overhead when one tries to verify standard forward simulation conditions. There is much potential to reduce complex reasoning to simple verification conditions local to threads, exploiting symmetry (all threads execute the same operations). Furthermore,

* Supported by the Deutsche Forschungsgemeinschaft (DFG), "Correct translation of abstract specifications to C-Code (VeriCode) " (grant RE828/26-1).

giving assertions reduces proofs to individual conditions for each step, which are easy to understand.

This paper develops an approach to prove forward simulations with proof obligations that are local to individual threads and steps of the programs. Generating these proof obligations has been implemented in our KIV theorem prover. It makes use of earlier work that developed a translation from programs to transition systems and defined local proof obligations for verifying invariants. We extend the approach to refinements by specifying local proof obligations for forward simulations.

We exemplify the approach by proving the correctness of a simple, concurrent implementation of hash sets. Proving the case study was presented as a challenge at last year's VerifyThis competition [21] for theorem provers. However, the case study turned out to be far too complex to verify in a 90-minute time frame (none of the participants got further than to verify just termination of a simplified sequential version). We define the algorithms in Section 2 and sketch their translation to a transition system. Section 3 defines the main invariant and summarizes the local proof obligations that are needed to establish it.

Section 4 defines the strategy for generating local proof obligations based on three mappings: one establishes a mapping between the control states of each thread in the concrete and the abstract system. The second provides a mapping of steps that has some resemblance to the mapping used in Event-B refinements [1]. The third defines a relation between the local states of threads.

For our case study, we achieve the desired effect: the reasoning is reduced to the essential arguments that show that the programs have an atomic effect at one specific instruction.

Finally, Section 5 gives related work and Section 6 concludes.

2 Case Study: Concurrent Hash Sets

We use a challenge of the 2022 VerifyThis competition [21] held at ETAPS as a case study to illustrate our approach. The tasks of the challenge [22] revolved around verifying the correctness of a simple but thread-safe and lock-free implementation of hash sets. The implementation produces hash sets with a fixed capacity and only provides functionality for insertions and membership queries.

Implementation of the Algorithms in KIV

The two main operations of the given algorithms can be executed concurrently by an arbitrary number of threads, and were translated into KIV programs using algebraic data types as a basis. For concurrent executions, we assume an *interleaving semantics* where each program statement (such as assignments or evaluations of conditionals) is executed atomically, but atomic steps of different threads can interleave. The implementation uses a fixed-sized array $ar : Array(Elem)$ storing keys of a generic type $Elem$ as a state variable. Each slot of ar is initialized with a designated key $\perp : Elem$, used as a placeholder for empty slots.

Algorithm 1 Hash Set Insertion Operation in KIV.

```
idle: Insert(e; ; b)
  precondition: e ≠ ⊥
  postcondition: b ↔ ∃ n. n < #ar ∧ ar[n] = e
I01:   let sz = #ar in
I02:   let n0 = get_hash(e, sz) in
I03:   let n = n0 in {
I04:       b := false;
I05:       while ¬ b do {
I06 with (ar[n] = e ⊃ doInsert(t, true); τ):
             let e0 = ar[n] in { // atomic load
I07 /* e0 ≠ ⊥ → e0 = ar[n] */:
               if e = e0 then {
I08 /* e0 = e ∧ e = ar[n] */:
                 b := true; // return true if the element is already there
I09:             return idle;
               } else
I10:             if e0 ≠ ⊥ then
I11:               n := (n + 1) mod sz // slot is occupied, try next slot
                 else {
I12 with (ar[n] = ⊥ ∨ ar[n] = e ⊃ doInsert(t, true); τ):
                   if* ar[n] = ⊥ // CAS (returns the new value in e0)
                   then e0 := e, ar[n] := e else e0 := ar[n];
I13:               if e0 = e then {
I14:                 b := true; // return true if the element was inserted
I15:                 return idle;
                   } else
I16:                 n := (n + 1) mod sz // slot is occupied, try next slot
             } };
I17:         if n = n0 then {
I18 with doInsert(t, false):
               b := false; // return false if the array is full
I19:           return idle;
             } else
I20:           skip; // continue with next loop iteration
       } };
I21:   return idle; // never reached

assertions
  I03 → I20 :  n0 = get_hash(e, #ar);
  I04 → I16 :  allslotsfull(ar, n0, n, e, false);
  I17 :  allslotsfull(ar, n0, n, e, true);
  ...
```

Algorithm 1 lists the KIV implementation (ignore the **with** clauses and **assertions** for the moment) of the **Insert** operation for adding keys to the set. The operation takes a key e : *Elem* as input and signals via the output b : *Bool* whether the requested key was inserted (or was already included in the set).[1] First, the algorithm calculates the hash value n_0 for the key e using the function `get_hash` (line `I02`). The function returns a value in the range $[0, sz)$, where sz is set to the size of ar (written $\#ar$). Then, the algorithm uses linear probing to find a free slot in ar, i.e., it searches for the closest following unoccupied location in ar starting from n_0. For this, the **while** loop (`I05` - `I20`) incrementally checks the entries of ar (accessing a location n of an array ar is written $ar[n]$).

Depending on the value e_0 of the slot currently considered, different situations must be handled. If the slot already contains the requested key e, nothing has to be inserted and the operation returns `true` (`I07` - `I09`). When the slot is occupied, i.e., e_0 is neither e nor $\perp$, the search must be continued at the next slot (`I10` - `I11`). For this, the current index n is incremented for the next loop iteration (note that the search continues at index 0 when the upper bound of the array is reached). If a free slot was found ($e_0 = \perp$), the algorithm tries to insert the element atomically using a CAS (compare-and-swap) operation (`I12`). In KIV, this is modeled using the **if*** construct, which performs the evaluation of its condition and the first statement of the chosen branch as one atomic step. In case the CAS was successful, the element was successfully added and operation returns with `true` (`I13` - `I15`). Otherwise, another thread interfered and occupied the slot, so the search must be continued (`I16`). Finally, insertion is aborted if the search went one full round and no free slot was found. Then the array is full, and the operation returns `false` (`I17` - `I19`).

Analogously, Algorithm 2 shows the implementation of the **Member** operation for checking whether a key e has been inserted into the set. The result b is again determined by traversing ar using linear probing (`M05` - `M17`) until the searched element was found (`M07` - `M09`). The search is aborted and the operation returns `false` when either the complete array was checked (`M14` - `M16`) or a $\perp$ was reached (`M10` - `M12`).

Note that the KIV implementations of both operations slightly differ from the pseudo-code given in the challenge description as it uses **do-while** loops, which are currently not supported by the programming language of KIV.

Translation to a State-Based Transition System

KIV provides functionality to automatically translate algorithms like the one given above to state-based transition systems. More precisely, the framework of *Input/Output Automata (IOA)* [15] is used.

[1] KIV procedures currently do not have return values. Instead, the parameters of a procedure are partitioned into input, reference, and output parameters, which are separated by semicolons.

Algorithm 2 Hash Set Member Operation in KIV.

```
idle: Member(e; ; b)
  precondition: e ≠ ⊥
  postcondition: b → ∃ n. n < #ar ∧ ar[n] = e
M01:  let sz = #ar in
M02:  let n0 = get_hash(e, sz) in
M03:  let n = n0 in {
M04:    b := false;
M05:    while ¬ b do {
M06 with (ar[n] = e ∨ ar[n] = ⊥ ∨ (n + 1) mod sz = n0 ⊃ doMember(t); τ):
          let e0 = ar[n] in // atomic load
M07:        if e = e0 then {
M08:          b := true; // return true if the element was found
M09:          return idle;
            } else
M10:          if e0 = ⊥ then {
M11:            b := false; // return false if empty entry was found
M12:            return idle;
              } else {
M13:            n := (n + 1) mod sz; // slot is occupied, try next slot
M14:            if n = n0 then {
M15:              b := false; // return false if array is full and element not in
M16:              return idle;
                } else
M17:              skip; // continue with next loop iteration
      } } };
M18:  return idle; // never reached
```

Definition 1. *An* Input/Output Automaton *(IOA) is a labeled transition system A with*

- *a type State of states,*
- *a predicate* `init`(s) *that fixes a subset of initial states s,*
- *a type Action of actions, and*
- *a step (or transition) predicate* `step`(s, a, s') *defining steps of the automaton from states s to states s', labeled by actions a.*

Actions can be viewed as parameterized ASM rules [3], as the names of Event-B events [1] parameterized by the values chosen in `ANY` ... `WHERE` clauses, or as Z operations [5] with inputs/outputs. The carrier set of *Action* is partitioned into internal actions a satisfying `internal`(a), which represent events of the system that are not visible to the environment, and external actions a satisfying `external`(a), which represent interactions of A with its environment. The set of external actions typically comprises *invoke* and *return* actions for each non-atomic operation, representing their invoking and returning steps and fixing the calling thread as well as the inputs and outputs. For example, the actions `invInsert`(t, e) and `retInsert`(t, b) represent the respective steps for the **Insert** operation (analogously, `invMember` and `retMember` for **Member**).

An *execution fragment* $\texttt{frag}(s_0\, a_1\, s_1\, a_2\, s_2\, a_3 \ldots)$ is a (finite or infinite) sequence of alternating states and actions such that $\texttt{step}(s_i, a_{i+1}, s_{i+1})$. An *execution* $\texttt{exec}(s_0\, a_1\, s_1\, a_2\, s_2\, a_3 \ldots)$ is additionally required to start with an initial state s_0 satisfying $\texttt{init}(s_0)$. The set of all executions or fragments of an automaton A is denoted $exec(A)$ and $frag(A)$, respectively. The *trace* of an execution is the projection of all its actions to the external ones, formally $\texttt{trace}(s_0\, a_1\, s_1\, a_2\, s_2\, a_3 \ldots) = a_1\, a_2\, a_3 \ldots \mid \{a_i \mid \texttt{external}(a_i)\}$. The set $traces(A)$ of all *traces* of an automaton A represents its visible behavior to a client. A trace shows concurrency by having several operations pending, e.g., the trace

$$\texttt{invInsert}(t_1, e_1)\ \texttt{invInsert}(t_2, e_2)\ \texttt{retInsert}(t_1, \texttt{true})\ \texttt{invMember}(t_1, e_2)$$

shows a situation where thread t_1 has inserted element e_1 successfully and is currently running a test for membership of e_2, while another thread t_2 is concurrently running an insertion of the same element e_2. Concurrent execution might add both $\texttt{retMember}(t_1, \texttt{true})$ or $\texttt{retMember}(t_1, \texttt{false})$ as the next action, depending on whether thread t_2 manages to insert the element before the check of thread t_1 or not.

In the following, we outline how the translation is performed for the hash set implementation; a more detailed description is given in [7].

The states of the automaton are constructed from three components: the global state $gs : GS$, the local state function $lsf : Tid \to LS$, and the program counter function $pcf : Tid \to PC$. The combined state is written as the tuple $\texttt{mkstate}(gs, lsf, pcf)$ of type $State$.

In KIV, states are given by (the values of) one or several (typed) state variables. The global state gs is the tuple of the state variables that can be accessed by all threads. For the hash set case study, this only includes the array ar, which can be accessed via the selector $gs.\texttt{ar}$. The local state function lsf stores local variables used by threads in the programs of the system. This includes all locally introduced variables in operations, e.g., sz or n in Alg. 1, as well as the parameters of operations, e.g., e and b in Alg. 1. The function stores a local state tuple $ls : LS$ for each thread $t : Tid$, where selectors for the individual fields are defined again. For example, the value of sz for a thread t is selected via $lsf(t).\texttt{sz}$.

The function pcf stores the program counter (control state) for each thread, which defines the current step of a thread within a program. For this, each atomic step in a KIV program is augmented with a unique *label* (`I01`, `I02`, …, `I21` for **Insert**, and `M01`, `M02`, …, `M18` for **Member**). The type PC is defined as an enumeration type containing a constant for each program label together with `idle` for a thread that is in between operation calls (of **Insert** or **Member**).

For the $\texttt{step}$ predicate, a generic axiom definition is generated.

$$\begin{aligned}
&\texttt{step}(\texttt{mkstate}(gs, lsf, pcf), a, \texttt{mkstate}(gs', lsf', pcf')) \\
\leftrightarrow \exists\, t.\quad &\texttt{pre}(gs, lsf(t), pcf(t), a) \wedge gs' = \texttt{gstepf}(gs, lsf(t), pcf(t), a) \\
&\wedge lsf' = lsf(t := \texttt{lstepf}(gs, lsf(t), pcf(t), a)) \\
&\wedge pcf' = pcf(t := \texttt{pcstepf}(gs, lsf(t), pcf(t), a))
\end{aligned}$$

The definition breaks down a system step to a step of one thread t by restricting changes of lsf and pcf to affect the parts of t only (the term $f(k := v)$ yields the function f where the value of $f(k)$ is updated to v). The three step functions `gstepf`, `lstepf`, and `pcstepf` calculate the next global and local state and the next program counter of this thread from the previous ones if the precondition predicate `pre` holds. These step functions and the precondition predicate are defined by axioms for each individual program counter.

The `pre` predicate fixes the actions a a program counter pc maps to, potentially depending on the current states gs and ls. The *Action* type contains values for all invoke and return steps of the automaton. Internal steps of non-atomic programs are typically mapped to the *default action* τ. However, internal steps can also be mapped to user-defined actions using a with-clause. We will assign actions representing (potential) *linearization points*, i.e., steps where an operation "takes effect" (cf. Sec. 4). For example, the steps I06, I12, and I18 of Alg. 1 are specified with the action `doInsert`, recording the current thread t and a boolean value determining whether the operation successfully inserted the element. The assignment of these actions can be conditional: the action of I06 is $\mathtt{doInsert}(t, \mathtt{true})$ only if $ar[n] = e$ holds at that point, otherwise it is τ. In the algorithm, the notation $\varphi \supset a_0; a_1$ is used as an abbreviation for an expression that computes a_0 if φ is true and a_1 otherwise. Thus, the precondition of I06 is specified by the following axiom, using the respective selectors to access the global and local state vars. [2]

$$\mathtt{pre}(gs, ls, \mathtt{I06}, a) \leftrightarrow a = (gs.\mathtt{ar}[ls.\mathtt{n}] = ls.\mathtt{e} \supset \mathtt{doInsert}(ls.\mathtt{tid});\ \tau)$$

State updates are also specified by individual axioms for the functions `gstepf` and `lstepf` for each program counter. For example, the **let**-statement at I06 introduces a new local variable e_0 and thus updates the corresponding field of the local state. On the other hand, the global state is not modified.

$$\mathtt{lstepf}(gs, ls, \mathtt{I06}, a) = (ls.\mathtt{e0} := gs.\mathtt{ar}[ls.\mathtt{n}])$$

$$\mathtt{gstepf}(gs, ls, \mathtt{I06}, a) = gs$$

Finally, the program counter step function `pcstepf` is defined based on the algorithm's control flow, e.g., the program counter of a thread is moved to I07 after the statement at I06 was executed. If the control flow can take different branches, the result of `pcstepf` is conditional. For example, after evaluating the **if**-condition at I07, the program counter is either set to I08 or I10.

$$\mathtt{pcstepf}(gs, ls, \mathtt{I06}, a) = \mathtt{I07}$$

$$\mathtt{pcstepf}(gs, ls, \mathtt{I07}, a) = (ls.\mathtt{e} = ls.\mathtt{e0} \supset \mathtt{I08};\ \mathtt{I10})$$

3 Local Proof Obligations for Invariants

For proving the refinement of the hash set implementation (see Sec. 4), an invariant restricting the reachable states of the automaton is necessary. This invariant

[2] To access the identifier of thread t, it is stored as a `tid`-field in its local state. An invariant ensures that threads store the correct identifier, i.e., $lsf(t).\mathtt{tid} = t$.

typically contains general consistency properties of the global state (independent of the local states of any thread, thus called *global invariants*) as well as various assertions for different control points of the algorithm (called *local invariants* as they also refer to the local states of threads).

The global invariant is given as a predicate $\mathtt{GInv}(gs)$. For the case study, it ensures that the array ar, in which the elements of the set are stored, has a valid size (it can store at least one element) and that its slots are filled correctly.

$$\mathtt{GInv}(ar) \leftrightarrow \#ar \neq 0 \wedge \mathtt{htok}(ar)$$

The latter property is expressed by the predicate $\mathtt{htok}$, which is defined using the auxiliary predicates $\mathtt{allslotsfull}$ and $\mathtt{between}$.

$$\begin{aligned}\mathtt{htok}(ar) \leftrightarrow\ \forall\, n.\quad & n < \#ar \wedge ar[n] \neq \bot \\ & \rightarrow \mathtt{allslotsfull}(ar, \mathtt{get\_hash}(ar[n], \#ar), n, ar[n], \mathtt{false})\end{aligned}$$

$$\begin{aligned}\mathtt{allslotsfull}(ar, n_0, n, e, b) \leftrightarrow\ \forall\, m.\quad & \mathtt{between}(n_0, m, n, b) \wedge m < \#ar \\ & \rightarrow ar[m] \neq e \wedge ar[m] \neq \bot\end{aligned}$$

$$\begin{aligned}\mathtt{between}(n_0, m, n, b) \leftrightarrow\quad & n_0 = n \wedge b \\ & \vee (n < n_0 \supset m < n \vee n_0 \leq m;\ n_0 \leq m \wedge m < n)\end{aligned}$$

The predicates encode that ar was filled by linear probing: it must hold for any non-$\bot$ element $ar[n]$ that all slots m between the element's hash value (calculated by $\mathtt{get\_hash}$) and the slot n it is stored in are "full", i.e., are occupied by other non-$\bot$ elements. Since the search for a free slot continues at the first slot when the end of the array is reached (cf. Alg. 1), the definition of $\mathtt{between}$ must consider both the case of $n_0 \leq n$ and the case of $n < n_0$ (expressed using the $\varphi \supset t_0; t_1$ notation). Note that the definitions just consider slots $m \in [n_0, n)$ when the flag b is $\mathtt{false}$, which is the case for the global invariant $\mathtt{htok}$. The predicates are used with $b \leftrightarrow \mathtt{true}$ only in local invariants to express that the array is filled completely (when all slots are considered, i.e., $n_0 = n$).

Instead of giving a local invariant formula directly, KIV generates a predicate definition from thread-local assertions for the individual program points. This approach facilitates tackling larger algorithms as the resulting formula becomes vast quite quickly (typically several pages of text, even for small case studies like the one presented in this paper). Thus, manually defining and maintaining this formula is very error-prone.

An assertion $\mathtt{LInv}_{\mathrm{pcval}}(gs, ls)$ can be given for every label $pcval \in PC$. In KIV, assertions can be encoded as a comment $/*\ \varphi\ */$ at the respective label (cf. lines I07 and I08 of Alg. 1). Since typically assertions hold for ranges in the code, they can also be given separately. For example, the assertions given at the bottom of Alg. 1 encode the progress of linear probing: in every iteration of the loop, all slots between the hash value $\mathtt{get\_hash}(e, \#ar)$ of the element and the current index n are occupied (I04 $\rightarrow$ I16 is a shorthand for the range I04, I05, $\ldots$, I15, I16). The critical step here is from I16 to I17, where the index n is incremented. At this point, the boolean flag of $\mathtt{allslotsfull}$ is toggled from false to true because n may have been incremented to n_0 when ar has been fully searched.

From the given assertions, KIV generates the definition of a local invariant predicate $\mathtt{LInv}(gs, ls, pc)$, which is then lifted to a full invariant definition $\mathtt{Inv}(gs, lsf, pcf)$ for the automaton.

$$\mathtt{LInv}(gs, ls, pc) \leftrightarrow \bigwedge_{pcval \in PC} (pc = pcval \rightarrow \mathtt{LInv}_{\mathrm{pcval}}(gs, ls))$$

$$\mathtt{Inv}(gs, lsf, pcf) \leftrightarrow \mathtt{GInv}(gs) \wedge \forall\ t.\ \mathtt{LInv}(gs, lsf(t), pcf(t))$$

Since the steps of threads can interleave, the given thread-local assertions must be *stable* over the steps of other threads for the invariant to hold. In order to avoid the combinatorial explosion of explicitly reasoning over all possible interleavings, a rely predicate $\mathtt{rely}(t, gs, gs')$ is used to abstract from the concrete modifications other threads can make. All steps that are *not* executed by thread t should satisfy this predicate when they start in global state gs and end with gs'. Thread t *relies* on other threads to change the global state according to rely. For the case study, the following rely predicate is sufficient, enforcing that no thread resizes the array and that no thread overwrites a slot at which an element has been inserted before.

$$\begin{aligned} &\mathtt{rely}(t, ar_0, ar_1) \\ \leftrightarrow\ &\#ar_0 = \#ar_1 \wedge \forall\ n.\ n < \#ar_0 \wedge ar_0[n] \neq \bot \rightarrow ar_1[n] = ar_0[n] \end{aligned}$$

With these definitions, proof obligations (POs) are generated that ensure that the predicate $\mathtt{Inv}(gs, lsf, pcf)$ is actually an invariant of the automaton. The obligations are formulated in sequent notion: a sequent $\Gamma \vdash \Delta$ abbreviates the formula $\forall \underline{x}.\ \bigwedge \Gamma \rightarrow \bigvee \Delta$ where Γ (the antecedent) and Δ (the succedent) are lists of formulas, and $\underline{x}$ is the list of all free variables in Δ and Γ.

step-pcval-pcval′: For every step from label *pcval* to *pcval′* with action a

$$\begin{aligned} &\mathtt{LInv}_{\mathrm{pcval}}(gs, ls),\ \mathtt{GInv}(gs),\ \mathtt{pre}(gs, ls, pcval, a) \\ \vdash\ \ &\mathtt{LInv}_{\mathrm{pcval'}}(\mathtt{gstepf}(gs, ls, \mathtt{LInv}_{\mathrm{pcval}}, a), \mathtt{lstepf}(gs, ls, \mathtt{LInv}_{\mathrm{pcval}}, a)) \\ &\wedge \mathtt{GInv}(\mathtt{gstepf}(gs, ls, pcval, a)) \end{aligned}$$

rely-pcval: For every step from label *pcval*

$$\begin{aligned} &\mathtt{LInv}_{\mathrm{pcval}}(gs, ls),\ \mathtt{GInv}(gs),\ \mathtt{pre}(gs, ls, pcval, a),\ ls.\mathtt{tid} \neq t \\ &\vdash \mathtt{rely}(t, gs, \mathtt{gstepf}(gs, ls, pcval, a)) \end{aligned}$$

stable-pcval: For every label *pcval*

$$\mathtt{LInv}_{\mathrm{pcval}}(gs, ls),\ \mathtt{GInv}(gs),\ \mathtt{rely}(t, gs, gs') \vdash \mathtt{LInv}_{\mathrm{pcval}}(gs', ls)$$

The first PO (**step-pcval-pcval′**) guarantees that each step of a thread establishes the thread-local assertion at the following statement and preserves the global invariant. The other two POs ensure that steps of other threads do not invalidate assertions. This is split into showing that all such steps are rely steps (**rely-pcval**) and that all assertions are stable over the rely (**stable-pcval**).

Note that often a significant amount of the generated obligations can be omitted. Many steps do not update the global state (when $\mathtt{gstepf}(gs, ls, pcval, a) =$

gs), and so the **rely-pcval** POs can be dropped for these steps as it is enforced that the `rely` predicate is reflexive. In fact, only the **rely-I12** PO is generated for the case study since the CAS at `I12` is the only step of the algorithm that modifies *ar*. Furthermore, if two assertions $\texttt{LInv}_{\text{pcval}}$ and $\texttt{LInv}_{\text{pcval}'}$ of different labels $pcval \neq pcval'$ are syntactically the same formula, the obligations **stable-pcval** and **stable-pcval′** are identical, so only one is generated.

In summary, 28 **stable** and 48 **step** proof obligations were verified with 65 interactions (including lemmas). Together they establish the invariant `Inv` of the IOA. A proof of the soundness of this thread-local proof technique is given in [7].

4 Local Proof Obligations for Refinement

While the invariants ensure that the array is always in a consistent state, they do not ensure that each operation has a desired effect, e.g. that insert adds at most the element given as input and deletes nothing. In a sequential setting simply augmenting the proof with suitable postconditions would be sufficient. In a concurrent setting this is not possible, as the postcondition can be invalidated by other threads. Instead one must show that the program behaves like an atomic operation. This is typically verified by giving abstract atomic descriptions of program behavior. A standard notion is *serializability* [18], which requires that programs behave like transactions: either they have an atomic effect or none at all when failing. *Opacity* [9] additionally requires that even failing transactions never read from states that result from partially executed transactions.

For concurrent libraries like the one we consider here, the standard correctness notion is *linearizability* [12], which in addition to atomicity requires that the effect of each operation happens between its invocation and its return. In contrast to other criteria, linearizability has the advantage that it is compositional: using several linearizable libraries is correct already if each library is correct.

The effect of a linearizable operation can be expressed directly as the whole code of each operation executing sequentially without any interleaving. This is done in model checking approaches, which automatically check that all possible interleavings of a fixed (usually very small) number of threads and operations has the same effect than executing them in some suitable sequential order. A more common approach in interactive proofs is to express the effect using simple operations of an abstract data type, like we do here.

Many of the atomicity criteria can be expressed as refinement correctness with respect to an abstract automaton (e.g., TMS2 for opacity, see [8]). A correct refinement from an automaton A (with states *as* of type *AState*, step relation `astep`, etc.) to an automaton C in general requires that the externally visible invoking and returning steps (i.e., the external actions of A and C that show their inputs/outputs) must be preserved, formally $traces(C) \subseteq traces(A)$.

Refinement can be verified using either a forward or a backward simulation. Together the approach is complete: if backward simulation is necessary, it is always possible to give an intermediate automaton, such that the upper refinement (often a simple one) can be verified using backward simulation, while the lower

`idle` : **InvInsert**(e)
atomic `invInsert`(t, e)
precondition : $e \neq \bot$
{ le := e;
  return `invIns` }
	`invIns` : **DoInsert**(do)
	atomic `doInsert`(t)
	{ lb := do;
	  if do **then**
	    set := $set \cup \{le\}$;
	  return `retIns` }
		`retIns` : **RetInsert**$(;;b)$
		atomic `retInsert`(t, b)
		{ b := lb;
		  return `idle` }

`idle` : **InvMember**(e)
atomic `invMember`(t, e)
precondition : $e \neq \bot$
{ le := e;
  return `invMem` }
	`invMem` : **DoMember**$()$
	atomic `doMember`(t)
	{ lb := $le \in set$;
	  return `retMem` }
		`retMem` : **RetMember**$(;;b)$
		atomic `retMember`(t, b)
		{ b := lb;
		  return `idle` }

Fig. 1: Canonical automaton for set operations.

one (usually the difficult one) is verified with a forward simulation. Therefore we will focus on forward simulations only, and on deriving thread-local proof obligations for this case. A forward simulation is defined as follows.

Definition 2. *A* forward simulation *from a concrete IOA C to an abstract IOA A is a relation* `abs` $\subseteq State \times AState$ *such that each of the following holds.*

Initialisation

$$\texttt{init}(s) \vdash \exists\ as.\ \texttt{ainit}(as) \land \texttt{abs}(s, as) \tag{1}$$

External step correspondence

$$\texttt{abs}(s, as),\ \texttt{step}(s, a, s'),\ \texttt{external}(a) \vdash \exists\ as'.\ \texttt{abs}(s', as') \land \texttt{astep}(as, a, as') \tag{2}$$

Internal step correspondence

$$\texttt{abs}(s, as),\ \texttt{step}(s, a, s'),\ \texttt{internal}(a) \vdash \exists\ \mathit{frag}(A)(as\ a_1\ as_1\ \ldots\ a_n\ as_n).\ \texttt{abs}(s', as_n) \land \forall\ i \leq n.\ \texttt{ainternal}(a_i) \tag{3}$$

It requires that the visible behavior represented by the actions of external steps to be preserved, i.e., one has to verify a commuting 1:1 diagram for each invoking or returning step, where equality of the action implies that the thread executing the step as well as its input/output are the same. In contrast, an internal step can refine an arbitrary number n of abstract internal steps. Often this number is one or zero, and we will focus on this case. If the number of steps is zero, the step is said to "refine skip" and $as = as_n$ holds.

For linearizability, the abstract specification A that has to be refined by the automaton C constructed from the algorithms is particularly simple and called the canonical automaton. The automaton has a state consisting of a data structure, here a *set* of elements (all different from $\bot$). For each operation available for the abstract data type (here: checking for membership and adding an element), it has three atomic steps.

The three steps for each operations are shown in Fig. 1 using KIV's general specifications of atomic steps of threads, indicated by the keyword **atomic** followed by the action of the step. These can in general be arbitrary programs again, although we here need simple assignments only.

The first of the three steps for each operation is an invoking step, that changes the program counter *apc* of the thread from `idle` to an invoked state (given after the **return** keyword). This step just copies the input to a local variable (here: *le*). The second step is a *Do* step that executes the operation, modifies the data structure and computes its result in a local variable (here: *lb*). The *Do* step changes the *apc* of the thread to a returning state, from which the *Return* step returns a result (by making it visible in its action) resetting the *pc* to `idle`. For the insert operation, the *Do* is nondeterministic, it can either insert the element, or refuse to do so, abstracting from the two possibilities of the insert algorithm. The nondeterminism is resolved by an additional boolean input that is also present in the action executed.

Like for the algorithms of Sec. 2, thread-local atomic steps accessing a global (here: *set*) and a thread-local state (here: the variables *le* and *lb*) are translated to predicate logic with preconditions `apre` and step functions `agstepf`, `alstepf`, `apcstepf`. The resulting canonical automaton A still allows operations of different threads to run concurrently, but insists that all operations have a simple, atomic effect described by the *Do* step that happens while the operation runs.

Finding a forward simulation between A and C requires finding the specific internal step of C where the effect of the operation happens. In general, finding a correct *linearization point* (LP) can be very difficult, e.g., it is possible that the LP of an operation is *not* a step of the thread executing it, but a step of another thread: one case is that thread t makes an offer, and another thread t' in a step that accepts the offer executes the LP of both threads (the elimination stack [11] and queue [17] are two instances). This case requires a forward simulation where one concrete step matches two *Do*-steps of the abstract specification.

The local proof obligations we give in this paper are tailored towards the most common case, which is that a specific step in the code of the thread executing an algorithm is its LP, which corresponds to the abstract *Do* step of the running operation. All other steps of an operation "refine skip", i.e., their proof obligation reduces to a 1:0 diagram.

For this case, we give a mapping that singles out the step, and gives the matching abstract *Do* step. This is done efficiently by exploiting that we can fix actions using the **with** clauses in the algorithms. For the **Insert** algorithm, see Alg. 1, there are three steps which can be the LPs: the obvious one is a successful CAS at line `I12`. However, a failed CAS at this line can also be a linearization point when the algorithm recognizes that the element is already present. For the same reason, the step at `I06` that loads $ar[n]$ is another LP when the loaded value is the element e that should be inserted. Finally, `I18` is an LP for the case where no element is inserted, since the array is full.

For the **Member** algorithm, only loading a value at `M06` can be an LP. It is one in three cases: First, when the element e checked to be in the set is loaded (**Member** will return `true`). Second, when $\perp$ is loaded: then **Member** will return `false`. Note that while there is often some freedom to choose an LP between several program steps, in this case the loading step is the only one that is correct. Any step executed later will not work, since in between executing the load and this step, another thread might have inserted e, and the abstract *Do* step would already return `true` rather than `false` as the algorithm does. Finally, the step is also an LP when the array slot checked is the last one, i.e., when $(n+1) \bmod sz = n_0$. In this case **Member** will return `false`.

To allow the definition of thread-local and step-local proof obligations, the abstraction relation is again split into a global part, and a thread-local part.

- The global abstraction relation $\mathtt{GAbs}(gs, ags)$ specifies how global states correspond. For the case study $\mathtt{absset}(gs.\mathtt{ar}, ags.\mathtt{set})$ is used, defined as $\forall\, e.\ e \in set \leftrightarrow \exists\, n.\ n < \#ar \wedge e = ar[n] \wedge e \neq \bot$.
- a local abstraction relation $\mathtt{LAbs}(gs, ls, pc, ags, als, apc)$ that gives the correspondence between program counters and local input and output values stored in ls, pc and als, apc, respectively (the relation may depend on the global states gs and ags). Like for the assertions used in invariants, we give these as assertions for certain ranges of program counters of the concrete algorithm. An example is $\mathtt{I5} : apc = (b \supset \mathtt{retIns};\ \mathtt{invIns}) \wedge (b \rightarrow \neg\, lb)$ which states that at `I5`, the abstract pc apc is before/after the *Do*-step, depending on the value of b, and that the local variable lb of the abstract specification is true when variable b used in the algorithm is true. In the proof obligations below, we refer to the formula that holds at a specific pc value $pcval$ as $\mathtt{LAbs}_{\mathrm{pcval}}(gs, ls, als, apc)$. The full `LAbs`-formula is defined as the conjunction of implications $pc = pcval \rightarrow \mathtt{LAbs}_{\mathrm{pcval}}(gs, ls, als, apc)$ for all pc values $pcval$, similar to the local invariant.

The full simulation relation includes the both global and local invariants as well as the global and local abstractions.

$$
\begin{aligned}
& \mathtt{abs}(gs, lsf, pcf, ags, alsf, apcf) \\
\leftrightarrow\ & \mathtt{GInv}(gs) \wedge \mathtt{AGInv}(ags) \wedge \mathtt{GAbs}(gs, ags) \\
& \wedge \forall\, t.\quad \mathtt{LAbs}(gs, lsf(t), pcf(t), alsf(t), apcf(t)) \\
& \qquad\quad \wedge \mathtt{LInv}(gs, lsf(t), pcf(t)) \wedge \mathtt{ALInv}(ags, alsf(t), apcf(t))
\end{aligned}
\tag{4}
$$

Assuming we have already proved invariants `LInv`, `GInv` and `ALInv`, `AGInv` for the concrete resp. abstract specification, we can now define thread-local, step-local proof obligations (POs) for a refinement. All POs share a number of common preconditions.

$$
\begin{aligned}
Prec\ = \ & \mathtt{GInv}(gs),\ \mathtt{AGInv}(ags),\ \mathtt{GAbs}(gs, ags), \\
& \mathtt{pre}(gs, lsf(t), pcf(t), a),\ \ gs' = \mathtt{gstepf}(gs, lsf(t), pcval, a), \\
& ls' = \mathtt{lstepf}(gs, lsf(t), pcval, a),\ \ pc' = \mathtt{pcstepf}(gs, lsf(t), pcval, a), \\
& \mathtt{LInv}_{\mathrm{pcval}}(gs, lsf(t)),\ \mathtt{ALInv}(ags, alsf(t)), \\
& \mathtt{LAbs}_{\mathrm{pcval}}(gs, lsf(t), alsf(t), apcf(t)), \\
& \forall\, t'.\ t' \neq t \rightarrow \quad \mathtt{LInv}(gs, lsf(t')) \wedge \mathtt{ALInv}(ags, alsf(t')) \\
& \qquad\qquad\qquad \wedge\ \mathtt{LAbs}(gs, lsf(t'), pcf(t'), ags, alsf(t'), apcf(t'))
\end{aligned}
$$

These refer to a concrete and an abstract state consisting of $gs, lsf, pcval$ and $ags, alsf, apcf$ related by `abs`, and to a thread t, that modifies the global state, the local state and the pc to gs', ls', and $pcval'$. The preconditions include a quantified formula that asserts the local invariants and local abstraction for other threads. For this case study, this quantified precondition is not required for the verification of the POs defined below. There are however case studies where a specific thread (e.g., a thread that has set a lock) influences another, where instantiating the quantifier is necessary.

Definition 3 (Thread-local, step-local proof obligations). *Each step from pcval to pcval′ of the concrete algorithm that executes action a under condition* φ *has two proof obligations. These depend on whether the action of the step is matched to an abstract action or not.*

Case 1 *The action a is also executed by the abstract system.*

PO-pcval-pcval′-same

$$Prec,\ \varphi,\ ags' = \mathtt{agstepf}(ags, alsf(t), apc, a),$$
$$als' = \mathtt{alstepf}(gs, lsf(t), pcval, a),\ \ apc' = \mathtt{apcstepf}(gs, lsf(t), pcval, a)$$
$$\vdash\quad \mathtt{apre}(ags, alsf(t), apcf(t)) \wedge \mathtt{GAbs}(gs', ags')$$
$$\wedge\ \mathtt{LAbs}_{pcval'}(gs', ls', pc', ags', als', apc')$$

PO-pcval-pcval′-other

$$Prec,\ \varphi,\ t \neq t',\ \mathtt{LInv}(gs, lsf(t')),\ \mathtt{ALInv}(ags, alsf(t')),$$
$$\mathtt{LAbs}(gs, lsf(t'), pcf(t'), ags, alsf(t'), apcf(t')),$$
$$gs' = \mathtt{gstepf}(gs, lsf(t), pcval, a),\ \ ags' = \mathtt{agstepf}(gs, lsf(t), pcval, a),$$
$$\vdash \mathtt{LAbs}(gs', lsf(t'), pcf(t'), ags', lsf(t'), pcf(t'), alsf(t'), apcf(t'))$$

Case 2 *The action a is not an abstract action.*

PO-pcval-pcval′-same

$$Prec,\ \varphi \vdash \mathtt{GAbs}(gs', ags) \wedge \mathtt{LAbs}_{pcval'}(gs', ls', ags, als, apc)$$

PO-pcval-pcval′-other

$$Prec,\ \varphi,\ t \neq t',\ \mathtt{LInv}(gs, lsf(t')),\ \mathtt{ALInv}(ags, alsf(t'))$$
$$\mathtt{LAbs}(gs, lsf(t'), pcf(t'), ags, alsf(t'), apcf(t'))$$
$$\vdash \mathtt{LAbs}(gs', lsf(t'), pcf(t'), ags', lsf(t'), pcf(t'), alsf(t'), apcf(t'))$$

Note that **with** clauses in the algorithms fix the condition φ under which a step is a linearization point, and therefore executes a specific abstract action. The two POs of each case distinguish preserving the global abstraction and the local abstraction of thread t that executes the step itself (**same**-POs), and preserving the local abstraction of some other thread $t' \neq t$ (**other**-POs).

The **other**-POs are trivial and dropped by the proof obligation generator when steps do not change the global state. When the global state changes, then the two `LAbs`-formulas must be expanded by their definition (and the proof obligation generator already does this), which results in quite large conjunctions over all assertions given. It is easy to prove that

Theorem 1. *The local proof obligations together with the initialization condition of forward simulation imply that* `abs` *as defined by (4) is a forward simulation between the concrete and the abstract system.*

by just noting that the assumption that `abs` holds for the initial states in the forward simulation conditions (2) and (3) implies all the preconditions of the thread local POs, except for the specific choice of `pre`, φ and a, which fixes one of the possible steps the concrete system has available. That `abs` in the postcondition of (2) and (3) is implied follows by looking at each individual predicate it consists of: that the global and local invariants hold again was already verified for each of the two automata C and A individually. Predicate `GAbs` is established by the **same**-PO. Finally, `LAbs` is established by the **same**-PO for thread t itself, and by the **other**-PO for all other threads.

The main reduction in effort is that doing all the case splits over available steps, the relevant quantifier reasoning for threads, the reduction of `LInv` and `LAbs` to the assertions $\texttt{LInv}_{\text{pcval}}$ and $\texttt{LAbs}_{\text{pcval}}$ that hold at a specific *pcval* has already been done, as well as dropping all trivial proof obligations. For our case study, the proof obligation generator results in 49 proof obligations of type **same**, and 15 of the **other** type. All but 5 are proven automatically by the simplifier.

The main difficult proof obligation is the one for the step that linearizes the member operation at M6. It requires showing that, based on the invariant `htok` and the assertion `allslotsfull` that holds at this point, linearization is correct for all three possible cases: the first is that the value loaded is $\perp$. In this case, we need the lemma

$$\begin{aligned}&\texttt{htok}(ar),\ ar[n] = \perp,\ e \neq \perp,\ \texttt{allslotsfull}(ar, \texttt{get\_hash}(e, \#ar), n, e, \texttt{false})\\ &\vdash (\forall\ m.\ m < \#ar \rightarrow ar[m] \neq e)\end{aligned}$$

The second case is that the last slot is loaded ($(n+1) \bmod sz = \texttt{get\_hash}(e, \#ar)$ holds) and is not e. This needs some quantifier reasoning for the `allslotsfull`-predicate to assert that the `between` range encompasses all array elements, implying the element e cannot be in the array. The third case, where e itself is loaded, is simple.

The other step that needs a lemma is the CAS step when inserting an element at I12. For the successful case a lemma is needed that asserts that updating both the array and the set preserves `absset`. Formulated as a rewrite rule

$$\begin{aligned}&\ \ n < \#ar \wedge ar[n] = \perp \wedge \texttt{absset}(ar, set)\\ &\rightarrow (\texttt{absset}(ar[n := e], set\ \cup \{e\}) \leftrightarrow e \neq \perp)\end{aligned}$$

the lemma is applied automatically, and just one interaction is needed that does a case split on whether the CAS succeeds.

Most of the effort in verifying the simulation now lies in fixing linearization points, and in defining suitable assertions based on this choice. Only 12 interactions were needed to prove the thread-local proof obligations. Verifying these was significantly simpler than proving the invariant of the concrete system. Development of thread local proof obligations was motivated and first tested with an earlier case study [6] on opacity. There, using thread local POs instead of the standard forward simulation conditions reduced the proof effort from 245 to 42 interactions. The online presentation [19] for this case study has been enhanced to include the new refinement proofs.

5 Related Work

Our approach is based on standard interleaving semantics used by many other formalisms. The more general semantics of concurrent ASMs [2] allows several threads (called agents) to make steps at the same time at the cost of considering clashes. Using a weak memory model would make reasoning more realistic but also more complex.

Our translation from programs to state-based transitions is influenced by Manna-Pnueli's work [16] and the translation of plusCAL [14] to TLA+. The thread-local proof obligations for invariants are influenced by rely-guarantee calculus [4,13]. However, because of symmetry, we need a `rely` predicate only, while the guarantee could be inferred as the conjunction of the `rely`'s for all other threads.

Our systems are usually step-deterministic, i.e., for a state s and the action a there is usually at most one state s' with $\mathtt{step}(s, a, s')$. The mapping between actions therefore allows to mimic a useful feature of the simulation conditions of Event-B refinement: these fix the choice of parameters for the `ANY`-clause of an abstract event (cf. [1], p. 251) avoiding the need for instantiation in the proof.

Most interactive theorem provers (Event-B is an exception) instantiate verified refinement theories and prove a simulation based on this, and we also follow that approach (a theory of IO Automata refinement is part of the web presentation [10]). Our work here resulted from the observation that for concurrent algorithms, the proof that shows sufficiency of thread-local proof obligations often constitutes a significant part of the work that can be avoided.

Our approach to thread-local proof obligations has some similarities to [20]. There, the proof obligations are specialized to linearizability and inferred on paper. An algorithm infers and verifies intermediate assertions automatically. The definition of a rely condition is avoided, instead the approach weakens assertions minimally (using decidable fragments of Separation Logic) to be stable over all the transitions of other threads.

6 Conclusion

We have defined an approach to the verification of concurrent threaded systems that reduces simulation proofs to thread-local, step-local proof obligations for a forward simulation. We found that this reduces the effort for verification significantly and allows us to focus on the core predicates and assertions needed for verification of the hash set implementation. All KIV specifications and proofs for the hash set case study can be found online [10].

In this paper we could not discuss various extensions that we either have already done (e.g., global system transitions that model crashes or flushing memory from volatile to persistent memory) or are future work (e.g., progress conditions). A comparison to the program calculus we alternatively use is also beyond the scope of this paper. Finally, it would also be interesting to see how incremental development of concurrent algorithms using several refinements could benefit.

References

1. J.-R. Abrial. *Modeling in Event-B: System and Software Engineering*. Cambridge University Press, 2010.
2. E. Börger and K.-D. Schewe. Concurrent abstract state machines. *Acta Informatica*, 53:469–492, 2016.
3. E. Börger and R. F. Stärk. *Abstract State Machines — A Method for High-Level System Design and Analysis*. Springer, 2003.
4. W.-P. de Roever, F. de Boer, U. Hannemann, J. Hooman, Y. Lakhnech, M. Poel, and J. Zwiers. *Concurrency Verification: Introduction to Compositional and Noncompositional Methods*. Number 54 in Cambridge Tracts in Theoretical Computer Science. Cambridge University Press, 2001.
5. J. Derrick and E. Boiten. *Refinement in Z and in Object-Z : Foundations and Advanced Applications*. Formal Approaches to Computing and Information Technology (FACIT). Springer, 2001. second, revised edition 2014.
6. J. Derrick, S. Doherty, B. Dongol, G. Schellhorn, O. Travkin, and H. Wehrheim. Mechanized Proofs of Opacity: a Comparison of two Techniques. *Formal Aspects of Computing (FAC)*, 30(5):597–625, 2018.
7. J. Derrick, S. Doherty, B. Dongol, G. Schellhorn, and H. Wehrheim. Verifying Correctness of Persistent Concurrent Data Structures: a Sound and Complete Method. *Formal Aspects of Computing (FAC)*, 33(4-5):547–573, 2021.
8. S. Doherty, L. Groves, V. Luchangco, and M. Moir. Towards Formally Specifying and Verifying Transactional Memory. *Formal Aspects of Computing (FAC)*, 25(5):769–799, 2013.
9. R. Guerraoui and M. Kapalka. On the Correctness of Transactional Memory. In *Proc. of Principles and Practice of Parallel Programming (PPOPP)*, pages 175–184, 2008.
10. Verification of Linearizability of Hash Sets with Local Proof Obligations with KIV. `http://www.informatik.uni-augsburg.de/swt/projects/HashSets.html`, 2023.
11. D. Hendler, N. Shavit, and L. Yerushalmi. A Scalable Lock-Free Stack Algorithm. In *Proc. of Parallelism in Algorithms and Architectures (SPAA)*, pages 206–215. ACM, 2004.
12. M. Herlihy and J. M. Wing. Linearizability: A Correctness Condition for Concurrent Objects. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 12(3):463–492, 1990.
13. C. B. Jones. Tentative Steps Toward a Development Method for Interfering Programs. *Transactions on Programming Languages and Systems*, 5(4):596–619, 1983.
14. L. Lamport. The PlusCal Algorithm Language. In *Proc. of Theoretical Aspects of Computing (ICTAC)*, pages 36–60. Springer, 2009.
15. N. A. Lynch and M. R. Tuttle. Hierarchical Correctness Proofs for Distributed Algorithms. In *Proc. of ACM Symposium on Principles of Distributed Programming (PODC)*, pages 137–151. ACM, 1987.
16. Z. Manna and A. Pnueli. *Temporal Verification of Reactive Systems – Safety*. Springer, 1995.
17. M. Moir, D. Nussbaum, O. Shalev, and N. Shavit. Using Elimination to Implement Scalable and Lock-Free FIFO Queues. In *Proc. of Parallelism in Algorithms and Architectures (SPAA)*, pages 253–262. ACM, 2005.
18. C. H. Papadimitriou. The Serializability of Concurrent Database Updates. *Journal of the ACM*, 26(4):631–653, 1979.

19. Verification of Opacity of a Transactional Mutex Lock with KIV and Isabelle. `http://www.informatik.uni-augsburg.de/swt/projects/Opacity-TML.html`, 2016.
20. V. Vafeiadis. Automatically Proving Linearisability. In *Proc. of Computer Aided Verification (CAV)*, volume 6174 of *LNCS*, pages 450–464. Springer, 2010.
21. VerifyThis Program Verification Competition Series. `https://www.pm.inf.ethz.ch/research/verifythis.html`.
22. VerifyThis 2022: Challenge 3 - The World's Simplest Lock-Free Hash Set. `https://ethz.ch/content/dam/ethz/special-interest/infk/chair-program-method/pm/documents/Verify%20This/Challenges2022/verifyThis2022-challenge3.pdf`, 2022.