Science of Computer Programming 241 (2025) 103227

Contents lists available at ScienceDirect

Science of Computer Programming

journal homepage: www.elsevier.com/locate/scico

Verification of forward simulations with thread-local, step-local proof obligations [☆]

Gerhard Schellhorn, Stefan Bodenmüller, Wolfgang Reif

Institute for Software and Systems Engineering, University of Augsburg, Universitätsstraße 6a, Augsburg, 86159, Germany

A R T I C L E   I N F O

Keywords:
Refinement
State-based concurrent systems
Thread-local proof obligations
Linearizability
Opacity
Interactive verification
KIV

A B S T R A C T

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. The approach has been implemented in our theorem prover KIV, which translates imperative programs to a set of transition rules and generates proof obligations accordingly. Instances of this proof technique should also be applicable to systems specified with ASM rules, B events, or Z operations. To exemplify the proof methodology, we demonstrate it with two case studies. The first verifies linearizability of a lock-free implementation of concurrent hash sets by showing that it refines an abstract concurrent system with atomic operations. The second applies the proof technique to the verification of opacity of Transactional Mutex Locks (TML), a Software Transactional Memory algorithm. Compared to the standard approach of proving a forward simulation directly, both case studies show a significant reduction in proof effort.

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 [1]. Most cases, however, like the ones 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, giving assertions reduces proofs to individual conditions for each step, which are easy to understand.

This paper presents 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

[☆] Supported by the Deutsche Forschungsgemeinschaft (DFG), “Correct translation of abstract specifications to C-Code (VeriCode)” (grant RE828/26-1).
E-mail addresses: schellhorn@informatik.uni-augsburg.de (G. Schellhorn), stefan.bodenmueller@informatik.uni-augsburg.de (S. Bodenmüller), reif@informatik.uni-augsburg.de (W. Reif).

https://doi.org/10.1016/j.scico.2024.103227
Received 7 December 2023; Received in revised form 21 October 2024; Accepted 1 November 2024

Available online 12 November 2024

[2] that developed a translation from programs to transition systems and defined local proof obligations for verifying invariants. We extend the technique to refinements by specifying local proof obligations for forward simulations.

We present two case studies to illustrate the proof technique. First, 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 [3] 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). After we give an overview of the refinement-based approach we pursue in this paper in Section 2, we define the hash set algorithms in Section 3 and sketch their translation to a transition system. Section 4 defines the main invariant and summarizes the local proof obligations that are needed to establish it.

Section 5 defines the strategy for generating local proof obligations for a concurrent refinement showing *linearizability* [1] of the hash set implementation. The obligations are based on three mappings: one establishes a correspondence 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 [4]. The third defines a relation between the local states of threads.

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

As a second example, we show how the technique can be applied to the Transactional Mutex Lock (TML) algorithm [5], an implementation of Software Transactional Memory (STM). This case study was already proven correct in earlier work [6], not using step-local proof obligations. It has been adapted to use the technique presented in this paper, which resulted in a significant reduction of proof effort. The general idea of STMs and the TML algorithm as an instance of STM are introduced in Section 6. Section 7 describes the correctness criterion *opacity* [7] capturing the atomicity property of STM transactions.

Furthermore, the existing abstract automaton TMS2 [8], which is proven to be opaque, is given as an abstraction of transactional memory implementations. Thus, showing correctness of TML can be done by proving it to be a refinement of TMS2. Section 8 outlines this refinement proof, demonstrating how thread-local, step-local proof obligation simplify verification compared to non-local reasoning. Finally, Section 9 gives related work and Section 10 concludes.

This paper extends the conference publication [9] by putting the work into context (Section 2) and by presenting TML as another case study (Section 6) to which the proposed methodology was applied successfully (Section 8). While the original paper focused on using refinement to prove linearizability, the additional case study demonstrates that the approach generalizes to other correctness criteria, such as *opacity* (Section 7).

2. Overview

This section gives an overview over the refinement-based approach used for the verification of concurrent algorithms that are executed by several threads and its specific realization in our theorem prover KIV.

Concurrent executions of such algorithms should typically satisfy certain atomicity constraints. These atomicity constraints are always specified as an abstract automaton. The abstract automaton can have internal steps, but it always has externally visible steps for a) the invocations of algorithms with information about inputs and for b) the responses of the algorithms, with return values. Invocations and responses (with inputs and outputs) are usually called *events* or *actions*.

The set of action sequences that are possible for the abstract automaton (its *traces*) then fixes the atomicity constraint that is imposed on the concrete operations: a set of algorithms satisfies the atomicity constraint iff its traces, i.e., sequences consisting of the invocations and responses when calling the concrete algorithms, are a subset of the ones allowed by the abstract automaton. This is formally specified as the criterion for refinement correctness.

There are abstract automata for various atomicity constraints (and one challenge is to find good ones for a new atomicity constraint). We will show two examples for the atomicity criteria of *linearizability* [1] (the standard criterion for concurrent libraries) and for *opacity* [7] (a strong form of *serializability* [10]) that is often used for implementations of Software Transactional Memory.

The approach requires to record the traces to be able to compare them. This can either be done using specific auxiliary state variables (often called history variables) that record a list of such actions/events. We prefer the I/O automaton approach which uses a labeled transition system, where the actions are modeled as labels. Then, traces are implicitly collected as the sequence of external labels.

The next sections will show two abstract automata that formalize the requirement of linearizability for concurrent libraries, and the requirement of opacity for implementations of STM.

It should be noted that refinement correctness is a pure safety criterion: An empty implementation that immediately deadlocks by having no transitions at all is correct, since it produces the empty set of traces, which is a subset of any set. Liveness properties, which imply that algorithms make progress, are not required for a correct refinement. They vary depending on the scenario, see [11] for an overview of common conditions. They are a separate problem that is independent of the atomicity constraints specified by the abstract automaton.

We ignore liveness conditions in this paper since they are trivial for the case studies we look at. To prove deadlock-freedom (some thread must always be able to make progress, i.e., be able to finish its currently running operation, assuming fairness), we use a rely-guarantee calculus (see [12,13]) similar to the one described in [14]. For lock-freedom (some thread must be able to make progress, no fairness assumed) temporal logic conditions have been defined [15,16]. For starvation-freedom (all threads must be able to make progress, assuming fairness) we have derived conditions in [17].

This paper focuses solely on proving refinement and on reducing the proof effort as best as possible. Ideal are proof obligations that talk about single steps of one algorithm (are *step-local*) and care about the global state, which is shared by all threads, and about the single local state of the one thread executing a step (are *thread-local*) only. With such conditions, one can focus on the specific step where a proof obligation fails, to understand whether assertions are missing or too strong (since they are invalidated by other threads).

KIV implements two approaches: one is to use a temporal logic calculus for programs, the other is to encode programs as an automaton. The first approach has the advantage that liveness properties are easier to express. However, it – like Hoare's calculus [18] or Rely-Guarantee calculus [19] – has to verify an algorithm as a whole (no step-locality).

The second approach is to translate to an automaton directly. This has the drawback that fairness and other liveness conditions are no longer part of program semantics (KIV's programming language has a (weak) fair interleaving operator as well as a non-fair one), but must be explicitly stated. On the other hand, it has the advantage that refinement proofs can be reduced to step-local conditions. Both approaches can be combined since both start with the same abstract programs, although there are some restrictions when they can be used: the automata based approach currently only supports non-recursive programs, and labels have to be added to programs, which the direct program calculus does not need. On the other hand, the approach based on a program calculus is less flexible for refinement. It can not handle some difficult cases (in particular none that require backward simulation), while the automaton based approach is universal.

Formally, an I/O automaton [20] is defined as follows.

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

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

Actions can be viewed as parameterized ASM rules [21], as the names of Event-B events [4] parameterized by the values chosen in `ANY ... WHERE` clauses, or as Z operations [22] with inputs/outputs. The type *Action* is partitioned into internal actions a satisfying $\mathtt{internal}(a)$, which represent events of the system that are not visible to the environment, and external actions a satisfying $\mathtt{external}(a)$, which represent interactions of A with its environment. In our scenario, the set of external actions comprises *invoke* and *return* actions for each algorithm, representing their invoking and returning steps. Such events fix the calling thread as well as inputs and outputs.

An *execution fragment* $\mathtt{frag}(s_0 a_1 s_1 a_2 s_2 a_3 \dots)$ is a (finite or infinite) sequence of alternating states and actions such that $\mathtt{step}(s_i, a_{i+1}, s_{i+1})$. An *execution* $\mathtt{exec}(s_0 a_1 s_1 a_2 s_2 a_3 \dots)$ is additionally required to start with an initial state s_0 satisfying $\mathtt{init}(s_0)$. 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 $\mathtt{trace}(s_0 a_1 s_1 a_2 s_2 a_3 \dots) = a_1 a_2 a_3 \dots \mid \{a_i \mid \mathtt{external}(a_i)\}$. The set $traces(A)$ of all *traces* of an automaton A represents its visible behavior to a client.

A correct refinement of an abstract automaton A to a concrete automaton C (written $C \leq A$) formally requires $traces(C) \subseteq traces(A)$. Refinement correctness can be shown by verifying that a *forward* or a *backward simulation* exists (and together the approach is complete [23]). Formally, a forward simulation requires to prove:

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

Initialisation

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

External step correspondence

$$\begin{gathered}\mathtt{abs}(s, as),\ \mathtt{step}(s, a, s'),\ \mathtt{external}(a) \\ \vdash \exists\, as'.\ \mathtt{abs}(s', as') \wedge \mathtt{astep}(as, a, as')\end{gathered} \tag{2}$$

Internal step correspondence

$$\begin{gathered}\mathtt{abs}(s, as),\ \mathtt{step}(s, a, s'),\ \mathtt{internal}(a) \\ \vdash \exists\, frag(A)(as\ a_1\ as_1\ \dots\ a_n\ as_n).\ \mathtt{abs}(s', as_n) \wedge \forall\, i \leq n.\ \mathtt{ainternal}(a_i)\end{gathered} \tag{3}$$

Backward simulation has a similar definition. 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 one (usually the difficult one) is verified with a forward simulation. Therefore, we focus on forward simulation. The step correspondence conditions are step-local already since they focus on one step of the algorithm, but are not thread-local since they consider the whole state of the automaton. For algorithms translated to an automaton, the state consists of the global (shared) state as well as of all the local states and program counter values of all threads, as we will see in the example given in the next section, where we detail the translation.

The proof obligations for forward simulations are not fully modular: the step correspondence proof obligation can be restricted to consider only states s and as that are reachable from initial states. These can be characterized by invariants. We prefer to prove invariants of individual automata separately, although formally they can simply be added as conjuncts to the abstraction relation.

The next section will demonstrate the approach by giving a simple concurrent algorithm that implements hash sets. We will show the algorithms as specified in KIV and demonstrate how they are translated to an automaton. The algorithms are linearizable, so we have to prove invariants and prove a forward simulation to an abstract specification that specifies the constraint of linearizability. This essentially requires that, to an observer, the inputs and outputs of concurrently executed operations look like if they were executed sequentially. To do this proof, Section 4 first shows invariants for the algorithms, and how proving them can be reduced to thread-local, step-local conditions. Section 5 shows an abstract automaton (called the *canonical automaton*) that characterizes linearizability. Again, we show how the global proof obligation of a forward simulation can be reduced to proof obligations that are thread-local.

3. Case study: concurrent hash sets

We use a challenge of the 2022 VerifyThis competition [3] held at ETAPS as a case study to illustrate our approach. The tasks of the challenge [24] 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.

3.1. 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. Note that the examples in this paper all use natural numbers (the type Nat in KIV) for numerical values, e.g., array indices or sizes (thus, values of variables $n, n_0, \ldots$ are always non-negative).

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 $n_0 : Nat$ for the key e using the function `gethash` (line I02).[2] 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 $n : Nat$ 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. Since the version of CAS used in the challenge description returns the value stored in the target (here $ar[n]$) *after* the operation, this value is assigned to the local variable e_0 in both the **then** and the **else** branches. In case the CAS was successful, the element was successfully added and operation returns with `true` (I13 - I15). Note that insertion is also successful if another thread has inserted the same element e into this slot (then the **else** branch of I12 is executed but the condition $e_0 = e$ of I13 is true nevertheless). Otherwise, some thread interfered and occupied the slot with an element other than e, 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 (see [24]) as it uses **do-while** loops, which are currently not supported by the programming language of KIV.

[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.

[2] The program construct **let** $x = t$ **in** α introduces a local variable x that is initialized with t and has scope α.

Algorithm 1 Hash Set Insertion Operation in KIV.

```
idle: Insert($e$; ; $b$)
  precondition: $e \neq \bot$
  postcondition: $b \leftrightarrow \exists\, n.\ n < \#ar \wedge ar[n] = e$
I01:  let $sz = \#ar$ in
I02:  let $n_0 =$ gethash($e, sz$) in
I03:  let $n = n_0$ in {
I04:    $b$ := false;
I05:    while $\neg\, b$ do {
I06 with ($ar[n] = e \supset$ doInsert($t$, true) ; $\tau$):
          let $e_0 = ar[n]$ in { // atomic load
I07 /* $e_0 \neq \bot \rightarrow e_0 = ar[n]$ */:
            if $e = e_0$ then {
I08 /* $e_0 = e \wedge e = ar[n]$ */:
              $b$ := true; // return true if the element is already there
I09:          returnidle;
            } else
I10:          if $e_0 \neq \bot$ then
I11:            $n$ := $(n + 1)$ mod $sz$ // slot is occupied, try next slot
              else {
I12 with ($ar[n] = \bot \vee ar[n] = e \supset$ doInsert($t$, true) ; $\tau$):
                if* $ar[n] = \bot$ // CAS (returns the new value in $e_0$)
                then $e_0$ := $e$, $ar[n]$ := $e$ else $e_0$ := $ar[n]$;
I13:            if $e_0 = e$ then {
I14:              $b$ := true; // return true if the element was inserted
I15:              returnidle;
                } else
I16:              $n$ := $(n + 1)$ mod $sz$ // slot is occupied, try next slot
          } };
I17:      if $n = n_0$ then {
I18 with doInsert($t$, false):
            $b$ := false; // return false if the array is full
I19:        returnidle;
          } else
I20:        skip; // continue with next loop iteration
      } };
I21:  returnidle; // never reached

assertions
  I03 → I20 : $n_0 =$ gethash($e, \#ar$);
  I04 → I16 : allslotsfull($ar, n_0, n, e$, false);
  I17 : allslotsfull($ar, n_0, n, e$, true);
  ...
```

3.2. Translation to a state-based transition system

KIV provides functionality to automatically translate algorithms like the one given above to an IO-Automaton.

The set of external actions is constructed as *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**).

The set of traces for the resulting now shows the visible behavior to a client. The example 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 retMember$(t_1, \texttt{true})$ or 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 [2].

The states of the automaton are constructed from three components: the global state $gs : GS$, the local state function $lsf : Tid \rightarrow LS$, and the program counter function $pcf : Tid \rightarrow PC$. The combined state is written as the tuple 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.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 Algorithm 1, as well as the parameters of operations, e.g., e and b in

Algorithm 2 Hash Set Member Operation in KIV.

```
idle: Member($e$; ; $b$)
  precondition: $e \neq \bot$
  postcondition: $b \to \exists\, n.\ n < \#ar \wedge ar[n] = e$
M01:  let $sz = \#ar$ in
M02:  let $n_0 = \mathtt{gethash}(e, sz)$ in
M03:  let $n = n_0$ in {
M04:    $b$ := false;
M05:    while $\neg\, b$ do {
M06 with $(ar[n] = e \vee ar[n] = \bot \vee (n+1) \bmod sz = n_0 \supset \mathtt{doMember}(t)\,;\,\tau)$:
          let $e_0 = ar[n]$ in// atomic load
M07:        if $e = e_0$ then {
M08:          $b$ := true; // return true if the element was found
M09:          return idle;
            } else
M10:          if $e_0 = \bot$ then {
M11:            $b$ := false; // return false if empty entry was found
M12:            return idle;
              } else {
M13:            $n$ := $(n+1) \bmod sz$; // slot is occupied, try next slot
M14:            if $n = n_0$ 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
```

Algorithm 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 $lsf(t)$.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 step predicate, a generic axiomatic definition is generated.

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

The definition breaks down a step of the full automaton to a local step lstep 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). Steps of one thread are further split into three step functions gstepf, lstepf, and pcstepf that 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, which are generated directly from algorithms specified in KIV, like Algorithm 1 and Algorithm 2. Note that this axiomatization scheme was chosen because it is beneficial for proving the *step-local* proof obligations we generate, where an obligation is specific to one concrete program counter (cf. Sec. 4 and Sec. 5).

The pre predicate determines whether a step with a certain action a can be executed. The *Action* type contains values for all invoke and return steps of the automaton (the *external actions*, see above). 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. This is necessary to correlate concrete and abstract steps when proving refinement: we will assign actions representing (potential) *linearization points*, i.e., steps where an operation "takes effect" (cf. Sec. 5).

For example, in Algorithm 1, no action is assigned to step I05, so it is mapped to τ, while steps 06 and I18 of 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, i.e., depend on the current states gs and ls: 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 following axioms specify the cases for steps I05, I06, I18 and idle of pre, using the respective selectors to access the global and local state vars.[3]

$$\mathtt{pre}(gs, ls, \mathtt{I05}, a) \leftrightarrow a = \tau$$

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

$$\mathtt{pre}(gs, ls, \mathtt{I18}, a) \leftrightarrow a = \mathtt{doInsert}(ls.\mathtt{tid}, \mathtt{false})$$

$$\begin{aligned}\mathtt{pre}(gs, ls, \mathtt{idle}, a) \leftrightarrow \exists e. \quad & a = \mathtt{invInsert}(ls.\mathtt{tid}, e) \wedge e \neq \bot \\ \vee\, & a = \mathtt{invMember}(ls.\mathtt{tid}, e) \wedge e \neq \bot\end{aligned}$$

The last example is when the thread is idle and can invoke both the **Insert** as well as the **Member** program. The predicate is defined in this case to include the preconditions of the algorithms. 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, while the global state is not modified.[4]

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

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

In the case study, the global state is only updated by a successful CAS statement at I12.[5]

$$\mathtt{gstepf}(gs, ls, \mathtt{I12}, a) = (gs.\mathtt{ar}[ls.\mathtt{n}] = \bot \supset gs.\mathtt{ar} := gs.\mathtt{ar}[ls.\mathtt{n} := ls.\mathtt{e}]\,; gs)$$

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.

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

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

While the action a is irrelevant for the axioms for internal steps of pcstepf, lstepf, and gstepf (as shown in the examples above), it is required for defining the steps of external actions. For example, a step with action $\mathtt{invInsert}(t, e_1)$ determines the thread t it changes the program counter of (from idle to I01) as well as the input e_1 that is written to the local state of t with the invocation. Thus, the corresponding axioms for pcstepf and lstepf are generated as follows.

$$ls.\mathtt{tid} = t \to \mathtt{pcstepf}(gs, ls, \mathtt{idle}, \mathtt{invInsert}(t, e_1)) = \mathtt{I01}$$

$$ls.\mathtt{tid} = t \to \mathtt{lstepf}(gs, ls, \mathtt{idle}, \mathtt{invInsert}(t, e_1)) = (ls.\mathtt{e} := e_1)$$

4. Local proof obligations for invariants

To prove the refinement of the hash set implementation (see Sec. 5), an invariant constraining the reachable states of the automaton is necessary. This invariant 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.

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

The latter property is expressed by the predicate cons,[6] which is defined using the auxiliary predicates allslotsfull and between.

$$\begin{aligned}\mathtt{cons}(ar) \leftrightarrow \forall\, n. \quad & n < \#ar \wedge ar[n] \neq \bot \\ & \to \mathtt{allslotsfull}(ar, \mathtt{gethash}(ar[n], \#ar), n, ar[n], \mathtt{false})\end{aligned}$$

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

[3] 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., $lsf(t).\mathtt{tid} = t$.

[4] The term $(x.\mathtt{sel} := y)$ yields x where the field sel is updated to y.

[5] The term $ar[n := e]$ yields the array ar where slot n is updated to e.

[6] In the actual KIV models, the predicate cons is called htok, like *"hash table okay"*.

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

The predicates encode that ar was filled by linear probing: any non-$\perp$ element $ar[n]$ in the array requires that all slots m between the element's hash value (calculated by `gethash`) and the slot n it is stored in are "full", i.e., are occupied by other non-$\perp$ elements. Since the search for a free slot continues at the first slot when the end of the array is reached (cf. Algorithm 1), the definition of `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 `false`, which is the case for the global invariant `cons`. The predicates are used with $b \leftrightarrow$ `true` only in local invariants to express that the array is filled completely (when all slots are considered, i.e., $n_0 = n$). Analogous to the algorithms, the predicates use natural numbers as arguments, so the definitions are formulated without any non-negativity conditions.

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 /* φ */ at the respective label (cf. lines I07 and I08 of Algorithm 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 Algorithm 1 encode the progress of linear probing: in every iteration of the loop, all slots between the hash value $\mathtt{gethash}(e, \#ar)$ of the element and the current index n are occupied (I04 → I16 is a shorthand for the range I04, I05, …, I15, I16). The critical step here is from I16 to I17, where the index n is incremented. At this point, the boolean flag of `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.

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

Note that for this particular case, there are no thread-specific rely conditions required (the parameter t is irrelevant in the formula), but often there is the necessity to give such conditions. For example, when locks are used or some kind of ownership properties are relevant (which can be added to the algorithms using auxiliary/ghost state variables), the `rely` predicate typically includes conditions like "*a lock acquired by thread t remains locked by t*", or "*the part of the state that is owned by t is unchanged*".

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 notation: 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

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

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 $pcval$

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

```
idle : InvInsert(e)
precondition : $e \neq \bot$
atomic with invInsert(t, e) {
    le := e;
    return invIns
}

invIns : DoInsert(do)
atomic with doInsert(t, do) {
    lb := do;
    if do then set := set ∪ {le};
    return retIns
}

retIns : RetInsert(; ; b)
atomic with retInsert(t, b) {
    b := lb;
    return idle
}
```

```
idle : InvMember(e)
precondition : $e \neq \bot$
atomic with invMember(t, e) {
    le := e;
    return invMem
}

invMem : DoMember()
atomic with doMember(t) {
    lb := le ∈ set;
    return retMem
}

retMem : RetMember(; ; b)
atomic with retMember(t, b) {
    b := lb;
    return idle
}
```

Fig. 1. Canonical automaton for set operations.

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**).

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.[7] 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 $\mathtt{LInv}_{\mathrm{pcval}}$ and $\mathtt{LInv}_{\mathrm{pcval}'}$ 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 [2].

5. Local proof obligations for refinement

While invariants ensure general consistency properties of a system, they do not ensure that each operation has a desired effect. For the hash set case study, the invariants guarantee that the array ar is always in a consistent state, but they do not imply that, for example, insert adds at most the element given as input and deletes nothing.

5.1. Correctness of concurrent libraries by refinement

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.

For concurrent libraries like the one we have presented in Sec. 3, the standard correctness notion is *linearizability* [1], 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. A correct refinement from an abstract automaton A to a concrete 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, cf. Sec. 2.

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*. Other criteria require different abstract automata, another one will be shown for the other case study in Section 7.

The canonical automaton has a state consisting of a data structure, for the hash set case study, this is 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. These 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. Again, these can generally be arbitrary programs, although we only need simple assignments here.

[7] Reflexivity is a standard property of relies. We also generate a respective PO, which can typically be proven automatically.

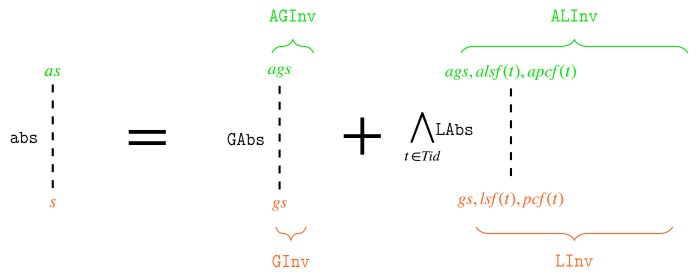

Fig. 2. Splitting the abstraction relation of an abstract automaton A with state as and a concrete automaton C with state s.

The first of the three steps for each operation is an invoking step, that changes the program counter of the thread from `idle` to an invoked state (`invIns` or `invMem`, 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 state of the thread to a returning state (`retIns` or `retMem` respectively), from which the *Return* step returns a result (by making it visible in its action) resetting the program counter 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 do that is also present in the action executed.

Like for the algorithms of Sec. 3, 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.

5.2. *Proving a forward simulation with local proof obligations*

Finding a forward simulation between A and C essentially 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 [25] and queue [26] 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.

As shown in Fig. 2, the abstraction relation is again split into a global part `GAbs` and a thread-local part `LAbs` to allow the definition of thread-local and step-local proof obligations.

- The *global abstraction relation* $\texttt{GAbs}(gs, ags)$ specifies how global states correspond.
- A *local abstraction relation* $\texttt{LAbs}(gs, ls, pc, ags, als, apc)$ 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. In the proof obligations below, we refer to the formula that holds at a specific pc value *pcval* as $\texttt{LAbs}_{\text{pcval}}(gs, ls, ags, als, apc)$. The full `LAbs`-formula is defined as the conjunction of implications $pc = pcval \rightarrow \texttt{LAbs}_{\text{pcval}}(gs, ls, ags, als, apc)$ 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.

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

Based on the invariants `LInv`, `GInv` and `ALInv`, `AGInv` for the concrete resp. abstract specification (which are established by the proof obligations given in Sec. 4), we can now define thread-local, step-local proof obligations (POs) for a refinement. All POs share a number of common preconditions *Prec*.

$$Prec \equiv pcval = pcf(t),\ \texttt{pre}(gs, lsf(t), pcval, a),$$

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

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 the hash set 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

$$\begin{aligned}
&Prec,\ \varphi,\ ags' = \texttt{agstepf}(ags, alsf(t), apcf(t), a),\\
&als' = \texttt{alstepf}(ags, alsf(t), apcf(t), a),\\
&apc' = \texttt{apcstepf}(ags, alsf(t), apcf(t), a)\\
&\vdash \quad \texttt{apre}(ags, alsf(t), apcf(t)) \wedge \texttt{GAbs}(gs', ags')\\
&\quad \wedge\ \texttt{LAbs}_{\texttt{pcval}'}(gs', ls', ags', als', apc')
\end{aligned}$$

PO-pcval-pcval′-other

$$\begin{aligned}
&Prec,\ \varphi,\ t \neq t',\ \texttt{LInv}(gs, lsf(t'), pcf(t')),\ \texttt{ALInv}(ags, alsf(t'), apcf(t')),\\
&ags' = \texttt{agstepf}(ags, alsf(t), apcf(t), a),\\
&\texttt{LAbs}(gs, lsf(t'), pcf(t'), ags, alsf(t'), apcf(t'))\\
&\vdash \texttt{LAbs}(gs', lsf(t'), pcf(t'), ags', alsf(t'), apcf(t'))
\end{aligned}$$

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

PO-pcval-pcval′-same

$$Prec,\ \varphi \vdash \texttt{GAbs}(gs', ags) \wedge \texttt{LAbs}_{\texttt{pcval}'}(gs', ls', ags, alsf(t), apcf(t))$$

PO-pcval-pcval′-other

$$\begin{aligned}
&Prec,\ \varphi,\ t \neq t',\ \texttt{LInv}(gs, lsf(t'), pcf(t')),\ \texttt{ALInv}(ags, alsf(t'), apcf(t'))\\
&\texttt{LAbs}(gs, lsf(t'), pcf(t'), ags, alsf(t'), apcf(t'))\\
&\vdash \texttt{LAbs}(gs', lsf(t'), pcf(t'), ags, alsf(t'), apcf(t'))
\end{aligned}$$

Fig. 3 depicts the commuting diagrams for the proof obligations of Definition 3. In each commuting diagram the initial abstract (left hand side) and the concrete step (bottom transition) are assumed (essentially these assumptions are the formula $Prec$). The state in the upper right corner is determined by the initial abstract state and the executed action, so it remains to prove the abstraction relation at the right hand side of the diagram.

Note that the **with** clauses in the algorithms fix the condition φ of the POs under which a step executes a specific abstract action (for the case study, if it is a linearization point). Given that a thread t executes a step with action a in the concrete system C, case **1)** gives a corresponding step of the abstract system A if a is also an abstract action. When a is not an abstract action, case **2)** ensures that the abstraction is stable over a step of C while A stutters. The two POs of each case further distinguish between preserving the abstraction for the thread t that executes the concrete step itself (**same**-POs, left-hand side of Fig. 3) and all other threads $t' \in Tid \setminus \{t\}$ (**other**-POs, right-hand side of Fig. 3; the obligations generalize to all threads by considering an arbitrary thread $t' \neq t$).

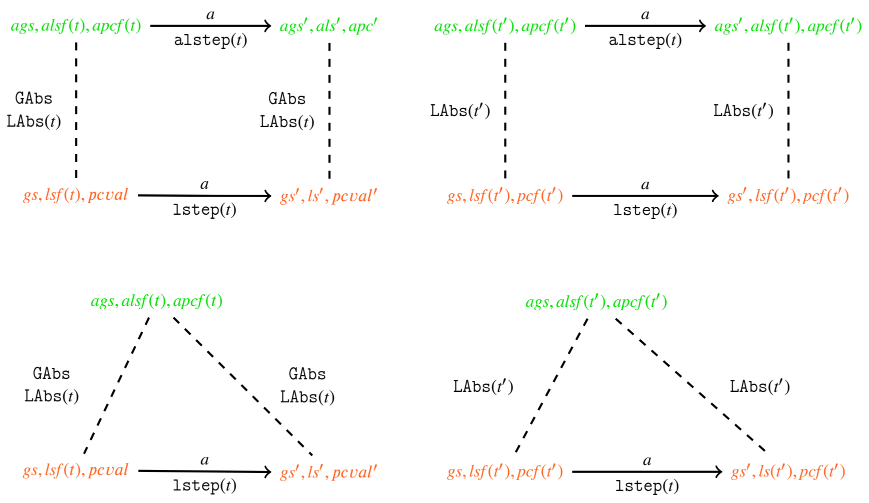

Fig. 3. Thread-local, step-local proof obligations for a refinement of an abstract automaton A to a concrete automaton C. The upper half depicts the **PO-pcval-pcval′-same** and **PO-pcval-pcval′-other** obligations for **Case 1)**, the respective obligations for **Case 2)** are shown in the lower half.

This distinction allows the proof obligation generator to drop **other**-POs when steps do not change the global state since the local state (as well as the program counter) can only change for the executing thread t (thus, the obligations become trivial). When the global state changes, 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 show that these proof obligations actually imply a forward simulation as defined by Definition 2, i.e., that Theorem 1 holds.

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.*

Proof. Except for the specific choice of pre, φ, and a, which fixes one of the possible steps the concrete system has available, all preconditions of the thread-local POs are implied by the assumption that abs holds for the initial states in the forward simulation conditions (2) and (3). abs in the postcondition of (2) and (3) follows by looking at each individual predicate it consists of: it was already verified that the global and local invariants hold again for each of the two automata C and A individually by the POs of Sec. 4. Predicate GAbs is established by the **same**-PO, and LAbs is established by the **same**-PO for thread t itself and by the **other**-PO for all other threads $t' \in Tid \setminus \{t\}$. Combining the **same**- and **other**-POs for **Case 1)** yields the commuting diagram for (2) as shown in Fig. 4 at the top, and the combined POs for **Case 2)** yield the commuting diagram for (3) (with $n = 0$), as shown in Fig. 4 at the bottom. □

5.3. *Proving refinement for the hash set case study*

To prove that the hash set library is a refinement of the canonical set automaton given in Fig. 1, linearization points of the **Insert** and **Member** algorithms (see Algorithm 1 and Algorithm 2) must be found first.

For **Insert**, 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 because the array is full.

For **Member**, 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, **Member** will return false if $\perp$ is loaded. 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 another thread might have inserted e between executing the load and this step, 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.

As global abstraction relation, absset(gs.ar, ags.set) is used, determining that the non-$\perp$ elements stored in ags.set must be identical to those in ags.set.

$$\texttt{absset}(ar, set) \leftrightarrow \big(\forall\ e.\ e \in set \leftrightarrow \exists\ n.\ n < \#ar \land e = ar[n] \land e \neq \perp\big)$$

For the local abstraction relation, various assertions are given. Some examples are:

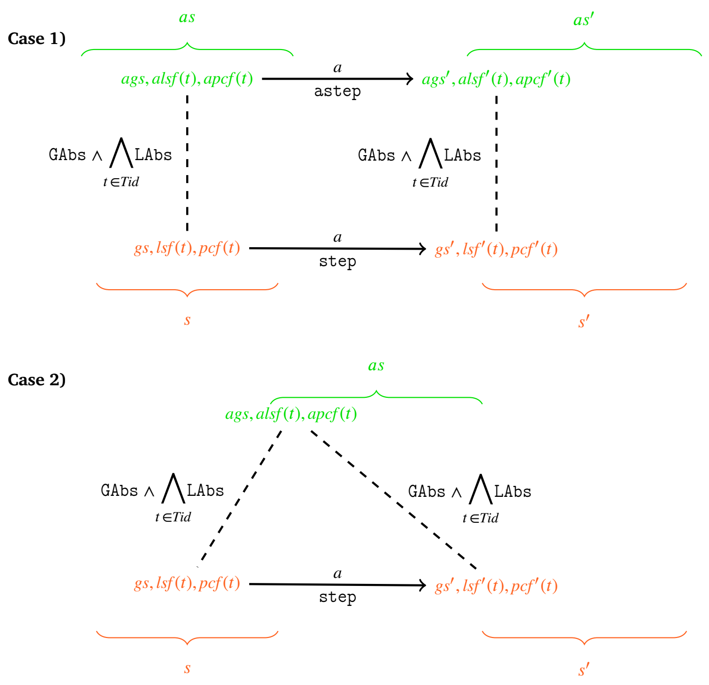

Fig. 4. Constructing commuting diagrams for a forward simulation abs between an abstract automaton A and a concrete automaton C from thread-local, step-local proof obligations.

$$\mathtt{I5} : apc = (b \supset \mathtt{retIns}\,;\mathtt{invIns}) \wedge (b \rightarrow \neg\, lb)$$

$$\mathtt{I7} : apc = (e = e_0 \supset \mathtt{retIns}\,;\mathtt{invIns}) \wedge (e = e_0 \rightarrow lb);$$

At I5, the abstract program counter apc is before/after the *Do*-step, depending on the value of b, and the local variable lb of the abstract specification is true when variable b used in the algorithm is true. Similarly at I7, apc is after the *Do*-step if the value e_0 just loaded (at I6) is equal to the value e to be inserted, and thus, the lb flag is set to true. Otherwise, apc is still before the *Do*-step at invIns.

Using local proof obligations leads to a significant reduction in proof effort. The main reduction is that the proof obligation generator already handles

- all the case splits over available steps,
- the relevant quantifier reasoning for threads,
- the reduction of LInv and LAbs to the assertions $\mathtt{LInv}_{\mathrm{pcval}}$ and $\mathtt{LAbs}_{\mathrm{pcval}}$ that hold at a specific $pcval$,
- and dropping all trivial proof obligations.

For the case study, this 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 cons 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 ⊥. In this case, we need the lemma

$$\mathtt{cons}(ar),\ ar[n] = \bot,\ e \neq \bot,$$

$$\mathtt{allslotsfull}(ar, \mathtt{gethash}(e, \#ar), n, e, \mathtt{false})$$

$$\vdash (\forall\ m.\ m < \#ar \rightarrow ar[m] \neq e)$$

The second case is that the last slot is loaded ($(n+1) \bmod sz = \mathtt{gethash}(e, \#ar)$ 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

$$n < \#ar \wedge ar[n] = \bot \wedge \mathtt{absset}(ar, set)$$

$\rightarrow (\texttt{absset}(ar[n := e], set \cup \{e\}) \leftrightarrow e \neq \bot)$

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.

6. Software transactional memory and TML

The development of thread-local proof obligations for refinement was motivated and first tested with an earlier case study [6] on the correctness of Software Transactional Memory (STM) implementations. In this section, we will explain the basic concepts of STMs and give the simple implementation of TML (Transactional Mutex Lock). The general correctness criterion of *opacity* for STMs and how it is mapped to refinement correctness will be discussed in the two next sections.

The use of STMs is motivated by the difficulties of getting concurrent multi-threaded programs right. The standard technique used most of the time is to use locks (e.g., mutexes or reader-writer locks). Getting their use right is difficult, in particular when several shared data structures must be updated that are protected with different locks. Bugs are often hard to find since unwanted behavior (e.g., a deadlock) can usually only be observed non-deterministically and is hard to reproduce.

Software Transactional Memory (STM) offers a uniform alternative by offering the simple concept of *transactions* for programs, see [27] for a comprehensive overview. Using an STM is often somewhat less efficient than an optimized solution, but relieves the programmer from reasoning in detail about possible data races or deadlocks.

All the programmer has do is to put the relevant *code* into an atomic block:

boolean $success$:= **tryatomic {** ⟨*code*⟩ **}**

The implementation of STM then guarantees that all threads that execute such atomic blocks can be thought of as executing the blocks in some sequential order. A trivial implementation would use a single global lock that would indeed enforce all code to execute sequentially.

Like for database transactions, an efficient STM implementation will however execute code concurrently and only check that an illegal interleaving of two executions, that would violate the view of the programmer of the two executing sequentially (a "conflict"), does not happen. In particular, if two atomic code blocks update different parts of memory (e.g., different data structures), the STM implementation will notice that the code can be executed concurrently without any conflict.

An efficient STM implementation will therefore execute code concurrently and just check for conflicts. If none is found, the transaction is successful, and variable $success$ is set to true. Otherwise the transaction *aborts*, and the STM implementation guarantees that the aborted transaction behaves as if nothing was executed at all: all data structures in memory are unchanged.

The standard reaction of programs to an aborted transaction is to retry it, so implementations of STMs also offer to just write

atomic { ⟨*code*⟩ **}**

to retry the transactional code until it succeeds.

To ensure that checks for conflicts are possible, STM implementations use a routine **TMBegin** to start and a routine **TMEnd** to finish an atomic code block. Reads and writes to shared memory locations within the code are replaced with calls to routines **TMRead** and **TMWrite**, respectively.

In many implementations of STMs, the replacement does not need to be programmed. Instead the compiler transforms ("instruments") the code within atomic blocks automatically. As an example, the compiler will instrument the statement

success := **tryatomic {** i := i + 1 **}**

as

TMBegin;
locali := **TMRead**(& i);
TMWrite(& i, locali + 1);
success := **TMEnd**

where '& i' is the address (*location*) at which shared variable i is stored in memory. The STM implementation will make sure that it cannot happen that two threads first load variable i and then both write i + 1. Instead, two executions of the atomic block will (when both succeed) be guaranteed to have done two increments.

Implementations of STMs (just like implementations of database transactions) differ in when they check for conflicts. It is possible to already check for conflicts early (in **TMRead** and **TMWrite**; then these may already abort the transaction) or to delay checks until **TMEnd** (*pessimistic* vs. *optimistic* strategy). There are also two strategies for writing: the *eager* strategy uses **TMWrite** to directly write to main memory. It then has to maintain an "undo log", which is used when the transaction must abort to undo any changes. The

Algorithm 3 The Transactional Mutex Lock (TML).

```
Init: $glb = 0$

TMBegin:
B1:  $loc := glb$;
B2:  while ($loc$ & 1) do
B3:      $loc := glb$;
B4:  return ok;

TMRead($addr$):
R1:  $val := *addr$;
R2:  if ($glb = loc$)
R3:      return $val$;
R4:  else return abort;

TMEnd:
E1:  if ($loc$ & 1)
E2:      $glb := loc + 1$;
E3:  return commit;

TMWrite($addr, val$):
W1:  if ($loc$ & 0)
W2:      if (!CAS(&$glb$, $loc$, $loc + 1$))
W3:          return abort;
W4:      else $loc$++;
W5:  $*addr := val$;
W6:  return ok;
```

alternative is the *lazy* strategy. In this strategy, writes are not done in **TMWrite**. Instead, **TMWrite** just stores them in a thread-local *write set*. Main memory is then updated in **TMEnd** only, where a succeeding transaction applies all updates together, while an aborting transaction just discards the write set.

The case study where thread-local proof obligations for refinement were developed verifies a simple STM implementation called TML ("Transactional Mutex Lock") [5]. The implementation is efficient when there are few writes but many reads. There are other, more sophisticated implementations like TL2 [28] and NOrec [29], which perform better when there are lots of writes (and therefore more conflicts). The implementation of TML is shown in Algorithm 3.

The implementation is eager (it updates main memory directly during writes). It allows only a single concurrent transaction that has done any writes (the *writer*), while a lot of transactions (*readers*) that have not (yet) done any writes can run concurrently. When there is no writer, a transaction that is so far a reader can become the writer. Having a single writer that never aborts its transaction allows to avoid the use of an *undo log*, which is usually necessary for the eager strategy.

The code uses a single shared variable[8] called glb, that is initialized to 0. An odd value in glb signals that there currently is a writer, while an even value indicates that there is no writer. All transactions initially load glb into a local variable loc. To ensure that they can read consistent values, they wait until there is no writer, i.e., until the loaded value is even (check at B2).

A transaction that wants to become a writer (by doing its first write) atomically tries to increment glb from an even to an odd value: The CAS at line W2 succeeds if glb still equal to the even loc as loaded at the start of the transaction in **TMBegin**. If the value has changed, then some other transaction has already become a writer (and probably changed memory), so the transaction is aborted (line W3). If the CAS succeeds, it sets glb to the odd value of $loc + 1$. Then loc is incremented too (line W4), and the odd value signals that the transaction now is the *writer*. All further writes of the transaction then see that they already have an odd loc value (line W1), so they can directly do the actual write to main memory (line W5).

The algorithm uses the pessimistic strategy for resolving conflicts: when reading recognizes that a new writer has started (test at R2), the transaction is aborted. Note that for the writer, the check at R2 will always succeed since both glb and loc are odd in this case, and no other transaction will then change an odd glb, so the writer will indeed never abort. When a transaction finishes, it checks whether it is the writer (line E1). If so, if increments the odd glb back to an even value.

Note that using a boolean for signaling whether a writer exists would not be sufficient to ensure correctness of the algorithm: then, the CAS at W2 could succeed for transaction t_1, even if another transaction t_2 became the writer, modified memory, and finished, since the boolean is false again. In this case, t_1 becoming the writer would be incorrect since its earlier reads could have read values before the updates of t_1, while further reads and writes would read/write updated values, violating atomicity.

7. Opacity and TMS2

In the previous section we have defined correctness of an STM implementation informally: the calling program with its concurrent should observe behaviors that look as if successful transactions execute in some sequential order, which is the criterion of *serializability*. A natural restriction is that if a transaction t_1 finishes before a transaction t_2 starts, then t_1 should occur before t_2 in the sequential order, which leads to *strict serializability* [10].

Serializability just requires that aborting transactions should not have any effect. A subtle question there is whether aborting transactions should be added to the sequential order. (Strict) serializability makes no such requirement, while the stronger criterion of *opacity* [7] requires that even aborting transactions read values from one consistent memory snapshot between two successful transactions. The difference is more important for STMs than databases, since database queries typically do not execute code that may contain infinite loops or throw exceptions. However, such code is possible in STMs, and corresponding behavior may result. As a simple example, consider a shared memory storing two integers x and y where transactions must preserve the invariant $x = y + 3$. A transaction that preserves this invariant is

t_1 = **tryatomic** { y := y + 3; x := x + 3 }

[8] A counter that is used as described is usually called a sequence lock.

The following second transaction with local variables lx and ly would then potentially get stuck in an infinite loop:

```
$t_2$ = **tryatomic** {
        $\text{lx} := \text{x}; \text{ly} := \text{y};$
        **while** $\text{ly} + 1 \neq \text{lx}$ **do** $\text{ly} := \text{ly} + 1$
    }
```

To understand this, consider the following concurrent run from an initial state where $y = 0$ and $x = 3$. First, t_2 loads $lx = 3$. Then, transaction t_1 starts and increments y to 3, after which t_2 loads this value into ly. Having read values from two different memory snapshots implies that t_2 must abort. If, however, the implementation of the STM follows the optimistic strategy of checking for consistency at the end of the transaction, t_2 will execute an infinite loop and never return. For the same reason, a transaction t_3 that replaces the loop body of t_2 with a division by $lx - ly$ may attempt to divide by zero.

Both behaviors are impossible when a simple global lock is used to protect transactions. In other words, behavioral refinement is violated. Opacity repairs this defect: it requires that transactional code can always rely to execute on consistent memory (where the invariant $x = y + 3$ holds), even when the transaction is aborted at the end. In the example, the STM implementation must ensure, that the attempt to load y by t_1 will either abort or will not yet see the updated value 3 and still load 0.

Opacity can be formalized by defining histories. These are sequences of events, where events are either read or write events for memory locations or transactional events for starting, (successfully) committing, or aborting a transaction. Informally, a history is opaque if it can be reordered into a history where transactions execute sequentially with the same results. A formal definition can be found in [6].

Since we are interested in thread-local proof obligations, we immediately give a sufficient criterion formalized as: the implementation must refine the TMS2 automaton given in Fig. 5.[9]

TMS2 formalizes the idea that any successful transaction (that has done any writes) creates a new memory snapshot when committing, and that other transactions can only read from a single memory snapshot. Therefore, TMS2 stores all memory snapshots created so far in a list $memories : List(L \to V)$, where each memory is assumed to map locations from L to values from V. A new memory is attached to the end of the list when a transaction commits successfully.

TMS2 assumes that the implementations of operations **TMBegin**, **TMRead**, **TMWrite**, and **TMEnd** are not atomic. So like in the canonical automaton (cf. Fig. 1 for set operations), TMS2 splits each of them into three steps: an *invoking* step, where the operation starts, a *do*-step which is similar to a linearization point, where the effect of the operation is observed, and a *return* step, when the operation finishes. The program counter pc_t ensures that a transaction t (starting with $pc_t = \texttt{notStarted}$) must first execute **TMBegin**, then (when $pc_t = \texttt{ready}$) can do any number of reads and writes, and finally must execute a **TMEnd**, finishing in $pc_t = \texttt{committed}$.

In contrast to linearizability, however, all four operation may also *abort* the transaction at any point: **RetAbort** can be executed before or after any do-step, while one of the four operations is running. An aborted transaction finishes with $pc_t = \texttt{aborted}$.

Reading and writing are done on transaction-local variables $rdSet_t : L \nrightarrow V$ and $wrSet_t : L \nrightarrow V$. Both store finite maps (partial functions) from locations L to V that have been observed/written so far (thus, both maps are initially empty for all transactions t). Note that they also use transaction-local auxiliary variables $lloc : L$ and $lval : V$ for storing inputs and outputs (analogously to the canonical automaton of Fig. 1).

Writing is done in **DoWrite** by adding to $wrSet_t$, and a nonempty write set leads to a new memory snapshot at the end of a successful transaction in **DoCommitWriter**. Both steps use an operation $\oplus$ that overwrites (when the location already has a value) or adds (when the map is finite and has no entry for the location).

Reading is more complex since it must be assured that a transaction only reads from a single valid memory snapshot. To ensure *strict* serializability, this snapshot cannot be older than the last one that was created before the transaction started. It can however be a newer one, that is created while the transaction is running (e.g., another transaction may commit before the transaction does any reading). Therefore, when **TMBegin** is executed, the transaction remembers the index of the last memory snapshot (the length of $memories$ minus one) that is currently available in $beginIdx_t$. Since this is all **TMBegin** has to do, **InvTMBegin** and **DoTMBegin** are combined into **InvTMBegin**.

As long as a transaction reads values only (i.e., when $\text{dom}(wrSet_t) = \emptyset$) it can be serialized right after its start, so even if many new memory snapshots are committed while the transaction is running, the read values may be from $memories(beginIdx_t)$. They can also be from any of the new memory snapshots that have been created so far. Therefore, **DoRead** can choose any index n that is at least $beginIdx_t$ and less than the number of snapshots created at the time of reading (determined by the precondition $\texttt{validIdx}(t, n)$). Reading is then done from $memories(n)$ checking that all values read earlier are those of stored in $memories(n)$ as well (check that $rdSet_t \subseteq memories(n)$).

The situation is different for transactions that do reads and writes. First, a transaction that writes a value val to a location loc and then reads this location, must return val and not the value of the snapshot it otherwise reads from. This results in the extra case for $lloc \in \text{dom}(wrSet_t)$ in **DoRead** which requires no consistency check for the read set.

Second, when such a transaction commits, it adds a new snapshot and must be serialized after all transactions that have committed so far.[10] Therefore, its reads must be consistent with the *last* memory snapshot available, for which the do-step of **TMEnd** is split

[9] There is an automaton TMS1 [8] equivalent to opacity, but no practically relevant STM implementation that needs this generalized version has been found so far.

[10] Note that transactions may still finish in a different order since the order of **RetTMEnd** may be different.

```
notStarted : InvTMBegin()
atomic {
    $beginIdx_t$ := $\#memories - 1$;
    return beginPending
}

ready : InvTMRead($loc$)
atomic {
    $lloc$ := $loc$;
    return invRead
}

ready : InvTMWrite($loc, val$)
atomic {
    $lloc$ := $loc$, $lval$ := $val$;
    return invWrite
}

ready : InvTMEnd()
atomic {
    return invCommit
}

ready : InvCancel()
atomic {
    return cancelPending
}

invCommit : DoCommitReadOnly(n)
precondition
    $\mathrm{dom}(wrSet_t) = \emptyset \wedge$ validIdx$(t, n)$
atomic {
    return retCommit
}

invRead : DoRead(n)
precondition
    $lloc \in \mathrm{dom}(wrSet_t) \vee$ validIdx$(t, n)$
atomic with doRead($t$) {
    if $lloc \in \mathrm{dom}(wrSet_t)$ then {
        $lval$ := $wrSet_t(lloc)$
    } else {
        $lval$ := $memories(n)(lloc)$;
        $rdSet_t$ := $rdSet_t \oplus \{lloc \to lval\}$;
    };
    return retRead
}
```

```
beginPending : RetTMBegin()
atomic {
    return ready
}

retRead : RetTMRead(; ; $val$)
atomic {
    $val$ := $lval$;
    return ready
}

retWrite : RetTMWrite()
atomic {
    return ready
}

retCommit : RetTMEnd()
atomic {
    return committed
}

$\mathbb{R}$ : RetAbort()
atomic with retAbort(t) {
    return aborted
}

invCommit : DoCommitWriter()
precondition
    $rdSet_t \subseteq memories$.last
atomic {
    let $mem = memories$.last $\oplus\ wrSet_t$ in
      $memories$ := attach$(memories, mem)$;
    return retCommit
}

invWrite : DoWrite()
atomic with doWrite($t$) {
    $wrSet_t$ := $wrSet_t \oplus \{lloc \to lval\}$;
    return retWrite
}
```

$\mathbb{R}\ \equiv\ \{$beginPending, cancelPending, invRead, invWrite, invCommit, retRead, retWrite$\}$
validIdx$(t, n)\ \equiv\ beginIdx_t\ \leq n < \#memories \wedge rdSet_t \subseteq memories(n)$

Fig. 5. The TMS2 automaton.

into two cases: One for a transaction that has done some writes (**DoCommitWriter**) where the write set is nonempty, and one for a transaction that has done reads only (**DoCommitReadOnly**).

Choosing an index n and checking $rdSet_t \subseteq memories(n)$ is not strictly necessary in the latter case (the index can be chosen identical to the one of the last read), but makes clear where in the sequential order the transaction should be placed: in between the transaction that created $memories(n)$ and the one that created $memories(n+1)$.

Proving that TML is a refinement of TMS2, such that the invoke and return steps match, is sufficient to prove opacity. Like for linearizability, it is possible in this proof to use the thread-local proof obligations.

8. Proof obligations for the refinement from TMS2 to TML

The original proof that TML refines TMS2 was done in Isabelle and is described in [6] together with an alternative approach that is somewhat more complex, since it needs to formalize histories to prove opacity directly. The Isabelle refinement proofs were then ported to KIV. Both the proofs in Isabelle and in KIV instantiate a formalization of forward simulation for IO Automata.

Originally, the case study involved a lot of overhead to translate programs to predicate logic definitions of automaton transitions and to instantiate the global proof obligations for forward simulation with formulas that talk about all threads and the full state. This motivated the development of a translation from programs to automata in KIV and the development of thread-local rely-guarantee proof obligations for invariants as described in [6], allowing to verify durable linearizability of a complex queue implementation.

Algorithm 4 The Transactional Mutex Lock (TML) in KIV.

```
notStarted: TMBegin()
  B1:  $loc := glb$;
  B2:  while odd($loc$) do
  B3:     $loc := glb$;
  B4:  return idle;

idle: TMWrite($l$, $v$)
  W1:  if even($loc$) then {
  W2:     if* $loc = glb$ then {
             $glb := loc + 1$, $w :=$ Some($t$)
          } else {
             skip;
  W3 with retAbort($t$):
             return aborted;
          };
  W4:     $loc := loc + 1$
       };
  W5 with doWrite($t$):
       $mem := mem(l := v)$;
  W6:  return idle;
```

```
idle: TMEnd()
  E1 with (even($loc$) ⊃ doCommitRO($t$); $\tau$):
       if odd($loc$) then
  E2 with doCommitWriter($t$):
          $glb := loc + 1$, $w :=$ None
  E3:  return committed;

idle: TMRead($l$; ; $v$)
  R1:  $v := mem(l)$;
  R2 with ($loc = glb$ ⊃ doRead($t$); $\tau$):
       if $loc = glb$ then
  R3:     return idle;
       else
  R4 with retAbort($t$):
          return aborted;

assertions
  R2 :  $loc = glb \rightarrow mem(l) = v$;
  W4, W5 :  $w =$ Some($t$);
  W4 :  even($loc$) $\wedge\ glb = loc + 1$;
  ...
```

For this, some extensions were necessary, e.g., the addition of global system transitions that model crashes or flushing memory from volatile to persistent memory, which are also discussed in the paper.

Thread-local invariant proof obligations reduced the effort to specify the TML and TMS2 automata of the case study considered here as well. However, the refinement proof still used a global forward simulation, and lemmas had to be defined to lift local simulation conditions to a global simulation and fix which steps of TML refine a specific step of TMS2 (or the empty step).

The support for thread-local proof obligations now allows to formalize only the essential aspects required to do a formal proof:

- For the verification, it is crucial to know which thread is currently the writer, if there is one. Therefore, an auxiliary global variable w is added to the code (cf. the KIV version of TML in Algorithm 4), that is set to Some(t) when thread t becomes the writer by incrementing glb to an odd value with a successful CAS in line W2. The auxiliary variable is reset to its initial value None when the writer increments glb back to an even value at the end of the transaction in line E2. The assertions of a thread have access to w as well as to their thread id t, so they can check whether the thread is currently the writer with $w =$ Some(t).
- Global and thread-local invariants, rely conditions and assertions for the two automata must be given. For TML, the global invariant is that writer w is None iff glb is even. A crucial condition is the rely condition: Every thread that is currently the writer can rely on memory mem, glb, and w being unchanged by steps of other threads. If the thread is not the writer, it can at least rely on glb not getting smaller. There is no thread-local invariant for TML, but several simple assertions hold for most program points. Two examples are: loc is never bigger than glb, except when loc has not been set, which is the case when the program counter is either idle or at B1; if a thread is not the writer, then it always has an even loc, except when its program counter is at E3. Finally, there are a few simple assertions for specific program points, e.g., that loc is even at W2. For TMS2, the global invariant simply states that $memories$ is never empty. The only assertion needed is that $beginIdx_t < \#memories$ holds whenever a transaction is not in its initial state notStarted, where $beginIdx_t$ is not specified. TMS2 needs no rely condition since all its steps are atomic.
- For refinement, the essential information needed is to fix the steps of TML that refine specific do-steps of TMS2 (that the external invoke and return steps of TML must refine the corresponding steps of TMS2 is clear). Like for Algorithm 1, this information is given in Algorithm 4 by annotating the relevant steps of TML with the corresponding abstract action using a **with** clause:
  - a successful check that $glb = loc$ at R2 refines doRead of TMS2 (a negative check refines skip).
  - Writing a value to memory at W5 refines doWrite.
  - Aborting at R4 or at W3 refines retAbort of TMS2.
  - Finally, the check that loc is odd at E1 refines doCommitRO when it is negative, and incrementing glb back to an even value at E2 refines doCommitWriter.
- For refinement, we finally need to relate global/local states of TML to global and local states of TMS2 with the local and global abstraction relation. This consists of a simple part that e.g. asserts that program counters of TML map to the corresponding ones of TMS2: all values of TML before the step that refines the do-Step of TMS2 (e.g., W1 to W5 for writing) are mapped to the single program counter value before the do-step of TMS2 (here: invWrite), all after this step (here: W6) are mapped to the value after the step (here: retWrite). The interesting part for correctness are the following three properties for thread t:
  - If there is currently no writer in TML, then the current memory mem of TML has the same content as the last snapshot $memories$.last of TMS2.
  - If t is the writer, then the memory of TML is identical to the result of applying the write set $wrSet_t$ of t to the last memory snapshot $memories$.last of TMS2. If t is not the writer in TML, its write set in TMS2 is empty.

- If *loc* is even, then $wrSet_t$ is empty.
- While a transaction is running (pc_t is not one of `idle`, `B1`, `committed`, `aborted`, or `E3`) the following holds: if $glb = loc$ or $pc_t = $ `W4`, the values read so far are from the last snapshot: $rdSet_t \subseteq memories$.`last`; otherwise the read set is from some snapshot n with $beginIdx_t \le n < \#memories$: $rdSet_t \subseteq memories(n)$.

With these properties given, KIV generates 49 proof obligations for the correctness of the refinement. 41 of them are proven automatically, the remaining 8 required 42 interactions. This should be contrasted with the proofs for the earlier version which proved a global forward simulation and needed 245 interactions, as well as manual definitions of lemmas, that are very similar to the thread-local proof obligations we generate now.

The most complex proof obligations result from the step where a thread becomes a writer with a successful CAS that changes *glb* to an odd value at `W2`, and from the step at `E2` where a writer commits and TMS2 executes the corresponding **DoCommitWriter**. These needed 9 and 14 interactions for the "other" proof obligations that shows that another thread is not affected by the now odd *glb* resp. the new memory snapshot created by **DoCommitWriter**.

To check that the generated thread-local proof obligations are correctly computed, the case study available at [30] also includes the (rather tedious) proof that the proof obligations of the case study imply a global forward simulation. Such a proof is now unnecessary in further case studies.

9. Related work

Our approach is based on standard interleaving semantics used by many other formalisms. The more general semantics of concurrent ASMs [31] allows several threads (called agents) to make steps at the same time at the cost of considering clashes. In our hash table implementation successful CAS instructions could then have clashes since two of them could be enabled with different new values at the same time, so an additional synchronization mechanism would be required. Since the algorithms we investigate cannot have clashes, we prefer a relational model for transitions. 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 [32] and the translation of plusCAL [33] to TLA+. The thread-local proof obligations for invariants are influenced by rely-guarantee calculus [34,35]. 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 a specific action a there is at most one state s' with `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. [4], 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 [36]). 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 [37]. 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.

A survey on verification methods for linearizability can be found in [38]. For an overview of work on verifying STM implementations, see for example [39,40]. More recent work often uses model checking to verify properties like deadlock- or starvation-freedom, e.g., [41] based on CSP or [42] based on Timed Automata. The technique presented in this paper is closely related to the approach of Lesani et al. [43,44], who also used I/O Automata and simulation proofs to verify the NOrec and TL2 algorithms with PVS [45]. In [46], Lesani presents *labeled synchronization logic (LSL)*, a first-order logic based on execution and linearization orders for reasoning about transaction algorithms. There, LSL is used for an alternative PVS proof for TL2, avoiding the translation of the algorithm to a transition system.

10. Conclusion

We have defined an approach to verifying 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 to verify concurrent implementations. We illustrated the approach using two case studies, showing that it is not specific to linearizability but also generalizes to other correctness criteria, such as opacity. All KIV specifications and proofs for these case studies can be found online: [36] for hash sets and [30] for TML.

In recent work [47], we applied the methodology to verify FliT [48], a persistency library for non-volatile memory (NVM). Using multiple refinements, we proved FliT to work correctly on a realistic weak memory model (PTSO [49], the persistent version of the TSO memory model of Intel's x86 processor [50]). In contrast to this work, the proof also includes a refinement using non-atomic programs as abstract specification (as an intermediate step, we consider FliT on a simpler, sequential consistent memory model).

A comparison to the program calculus we alternatively use (cf. for example [51]) is beyond the scope of this paper. In future work, we plan to extend the approach further, e.g., with progress conditions. Finally, it would also be interesting to see how incremental development of concurrent algorithms using several refinements could benefit.

CRediT authorship contribution statement

Gerhard Schellhorn: Writing – review & editing, Writing – original draft, Software, Methodology, Investigation, Conceptualization. **Stefan Bodenmüller:** Writing – review & editing, Writing – original draft, Software, Methodology, Formal analysis, Conceptualization. **Wolfgang Reif:** Supervision, Resources, Funding acquisition.

Declaration of competing interest

The authors declare the following financial interests/personal relationships which may be considered as potential competing interests: Stefan Bodenmueller reports financial support was provided by German Research Foundation. If there are other authors, they declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

References

[1] M. Herlihy, J.M. Wing, Linearizability: a correctness condition for concurrent objects, ACM Trans. Program. Lang. Syst. 12 (3) (1990) 463–492.
[2] J. Derrick, S. Doherty, B. Dongol, G. Schellhorn, H. Wehrheim, Verifying correctness of persistent concurrent data structures: a sound and complete method, Form. Asp. Comput. 33 (4–5) (2021) 547–573.
[3] VerifyThis Program Verification Competition Series, https://www.pm.inf.ethz.ch/research/verifythis.html.
[4] J.-R. Abrial, Modeling in Event-B: System and Software Engineering, Cambridge University Press, 2010.
[5] L. Dalessandro, D. Dice, M. Scott, N. Shavit, M. Spear, Transactional mutex locks, in: Euro-Par 2010 - Parallel Processing, Springer, 2010, pp. 2–13.
[6] J. Derrick, S. Doherty, B. Dongol, G. Schellhorn, O. Travkin, H. Wehrheim, Mechanized proofs of opacity: a comparison of two techniques, Form. Asp. Comput. 30 (5) (2018) 597–625.
[7] R. Guerraoui, M. Kapalka, On the correctness of transactional memory, in: Proc. of Symposium in Principles and Practice of Parallel Programming (PPoPP), 2008, pp. 175–184.
[8] S. Doherty, L. Groves, V. Luchangco, M. Moir, Towards formally specifying and verifying transactional memory, Form. Asp. Comput. 25 (5) (2013) 769–799.
[9] G. Schellhorn, S. Bodenmüller, W. Reif, Thread-Local, step-local proof obligations for refinement of state-based concurrent systems, in: Proc. of Rigorous State-Based Methods (ABZ), in: LNCS, vol. 14010, 2023, pp. 70–87.
[10] C.H. Papadimitriou, The serializability of concurrent database updates, J. ACM 26 (4) (1979) 631–653.
[11] M. Herlihy, N. Shavit, On the nature of progress, in: OPODIS, in: LNCS, vol. 7109, Springer, 2011, pp. 313–328.
[12] G. Schellhorn, S. Bodenmüller, J. Pfähler, W. Reif, Adding concurrency to a sequential refinement tower, in: Proc. of International Conference on Rigorous State-Based Methods (ABZ), in: LNCS, vol. 12071, Springer, 2020, pp. 6–23.
[13] S. Bodenmüller, G. Schellhorn, M. Bitterlich, W. Reif, Flashix: modular verification of a concurrent and crash-safe flash file system, in: Logic, Computation and Rigorous Methods: Essays Dedicated to Egon Börger on the Occasion of His 75th Birthday, in: LNCS, vol. 12750, Springer, 2021, pp. 239–265.
[14] Q. Xu, W.-P. de Roever, J. He, The rely-guarantee method for verifying shared variable concurrent programs, Form. Asp. Comput. 9 (2) (1997) 149–174, https://doi.org/10.1007/BF01211617.
[15] A. Gotsman, B. Cook, M. Parkinson, V. Vafeiadis, Proving that nonblocking algorithms don't block, in: POPL, ACM, 2009, pp. 16–28.
[16] B. Tofan, G. Schellhorn, W. Reif, Formal verification of a lock-free stack with hazard pointers, in: Proc. of ICTAC, in: LNCS, vol. 6916, Springer, 2011, pp. 239–255.
[17] G. Schellhorn, O. Travkin, H. Wehrheim, Towards a thread-local proof technique for starvation freedom, in: Integrated Formal Methods IFM 2016, in: LNCS, vol. 9681, Springer, 2016, pp. 193–209.
[18] C. Hoare, An Axiomatic Basis for Computer Programming (1969), Commun. ACM 576–580.
[19] C.B. Jones, Specification and design of (parallel) programs, in: Proceedings of IFIP'83, North-Holland, 1983, pp. 321–332.
[20] N.A. Lynch, M.R. Tuttle, Hierarchical correctness proofs for distributed algorithms, in: Proc. of ACM Symposium on Principles of Distributed Programming (PODC), ACM, 1987, pp. 137–151.
[21] E. Börger, R.F. Stärk, Abstract State Machines — A Method for High-Level System Design and Analysis, Springer, 2003.
[22] J. Derrick, 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.
[23] N. Lynch, F. Vaandrager, Forward and backward simulations – part I: untimed systems, Inf. Comput. 121 (2) (1995) 214–233.
[24] 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.
[25] D. Hendler, N. Shavit, L. Yerushalmi, A scalable lock-free stack algorithm, in: Proc. of Parallelism in Algorithms and Architectures (SPAA), ACM, 2004, pp. 206–215.
[26] M. Moir, D. Nussbaum, O. Shalev, N. Shavit, Using elimination to implement scalable and lock-free FIFO queues, in: Proc. of Parallelism in Algorithms and Architectures (SPAA), ACM, 2005, pp. 253–262.
[27] T. Harris, J.R. Larus, R. Rajwar, Transactional Memory, Synthesis Lectures on Computer Architecture, Morgan &, 2nd edition, Claypool Publishers, 2010.
[28] D. Dice, O. Shalev, N. Shavit, Transactional locking II, in: Proc. of International Symposium on Distributed Computing (DISC), 2006, pp. 194–208.
[29] L. Dalessandro, M.F. Spear, M.L. Scott, NOrec: streamlining STM by abolishing ownership records, in: Proc. of Symposium on Principles and Practice of Parallel Programming (PPoPP), 2010, pp. 67–78.
[30] Verification of Opacity of a Transactional Mutex Lock with KIV and Isabelle, http://www.informatik.uni-augsburg.de/swt/projects/Opacity-TML.html, 2016.
[31] E. Börger, K.-D. Schewe, Concurrent abstract state machines, Acta Inform. 53 (2016) 469–492.
[32] Z. Manna, A. Pnueli, Temporal Verification of Reactive Systems – Safety, Springer, 1995.
[33] L. Lamport, The PlusCal algorithm language, in: Proc. of Theoretical Aspects of Computing (ICTAC), Springer, 2009, pp. 36–60.
[34] C.B. Jones, Tentative steps toward a development method for interfering programs, Trans. Program. Lang. Syst. 5 (4) (1983) 596–619.
[35] W.-P. de Roever, F. de Boer, U. Hannemann, J. Hooman, Y. Lakhnech, M. Poel, J. Zwiers, Concurrency Verification: Introduction to Compositional and Noncompositional Methods, Cambridge Tracts in Theoretical Computer Science, vol. 54, Cambridge University Press, 2001.
[36] Verification of Linearizability of Hash Sets with Local Proof Obligations with KIV, http://www.informatik.uni-augsburg.de/swt/projects/HashSets.html, 2023.

[37] V. Vafeiadis, Automatically proving linearisability, in: Proc. of Computer Aided Verification (CAV), in: LNCS, vol. 6174, Springer, 2010, pp. 450–464.
[38] B. Dongol, J. Derrick, Verifying linearisability: a comparative survey, ACM Comput. Surv. 48 (2) (2015).
[39] M. Lesani, On the Correctness of Transactional Memory Algorithms, Ph.D. thesis, University of California, Los Angeles (UCLA), 2014.
[40] A. Cristal, B.K. Ozkan, E. Cohen, G. Kestor, I. Kuru, O.S. Unsal, S. Tasiran, S.O. Mutluergil, T. Elmas, Verification tools for transactional programs, in: Transactional Memory. Foundations, Algorithms, Tools, and Applications, in: LNCS, vol. 8913, 2015, pp. 283–306.
[41] C. Xu, X. Wu, H. Zhu, M. Popovic, Modeling and verifying transaction scheduling for software transactional memory using CSP, in: Proc. of International Symposium on Theoretical Aspects of Software Engineering (TASE), 2019, pp. 240–247.
[42] B. Kordic, M. Popovic, S. Ghilezan, Formal verification of python software transactional memory based on timed automata, Acta Polytech. Hung. 16 (7) (2019) 197–216.
[43] M. Lesani, V. Luchangco, M. Moir, A framework for formally verifying software transactional memory algorithms, in: Proc. of International Conference on Concurrency Theory (CONCUR), Springer, 2012, pp. 516–530.
[44] M. Lesani, J. Palsberg, Decomposing opacity, in: Proc. of International Symposium on Distributed Computing (DISC), in: LNCS, vol. 8784, Springer, 2014, pp. 391–405.
[45] S. Owre, J.M. Rushby, N. Shankar, PVS: a prototype verification system, in: D. Kapur (Ed.), Proc. of International Conference on Automated Deduction (CADE), in: LNCS, vol. 607, Springer, 1992, pp. 748–752.
[46] M. Lesani, Transaction protocol verification with labeled synchronization logic, in: Proc. of NASA Formal Methods (NFM), in: LNCS, vol. 11460, Springer, 2019, pp. 280–297.
[47] S. Bodenmüller, J. Derrick, B. Dongol, G. Schellhorn, H. Wehrheim, A fully verified persistency library, in: Proc. of International Conference on Verification, Model Checking, and Abstract Interpretation (VMCAI), in: LNCS, 2024, pp. 26–47.
[48] Y. Wei, N. Ben-David, M. Friedman, G.E. Blelloch, E. Petrank, FliT: a library for simple and efficient persistent algorithms, in: Proc. of Symposium in Principles and Practice of Parallel Programming (PPoPP), ACM, 2022, pp. 309–321.
[49] A. Khyzha, O. Lahav, Taming x86-TSO persistency, Proc. ACM Program. Lang. 5 (POPL) (2021) 47:1–47:29.
[50] A. Raad, J. Wickerson, G. Neiger, V. Vafeiadis, Persistency semantics of the intel-x86 architecture, Proc. ACM Program. Lang. 4 (POPL) (2020) 11:1–11:31.
[51] G. Schellhorn, S. Bodenmüller, M. Bitterlich, W. Reif, Software & system verification with KIV, in: The Logic of Software. A Tasting Menu of Formal Methods: Essays Dedicated to Reiner Hähnle on the Occasion of His 60th Birthday, in: LNCS, vol. 13360, Springer, 2022, pp. 408–436.