Symbolic Abstract Data Type Inference

Abstract

Formal specification is a vital ingredient to scalable verification of software systems. In the case of efficient implementations of concurrent objects like atomic registers, queues, and locks, symbolic formal representations of their abstract data types (ADTs) enable efficient modular reasoning, decoupling clients from implementations. Writing adequate formal specifications, however, is a complex task requiring rare expertise. In practice, programmers write reference implementations as informal specifications.

In this work we demonstrate that effective symbolic ADT representations can be automatically generated from the executions of reference implementations. Our approach exploits two key features of naturally-occurring ADTs: violations can be decomposed into a small set of representative patterns, and these patterns manifest in executions with few operations. By identifying certain algebraic properties of naturally-occurring ADTs, and exhaustively sampling executions up to a small number of operations, we generate concise symbolic ADT representations which are complete in practice, enabling the application of efficient symbolic verification algorithms without the burden of manual specification. Furthermore, the concise ADT violation patterns we generate are human-readable, and can serve as useful, formal documentation.

Categories and Subject Descriptors F.3.1 [*Specifying and Verifying and Reasoning about Programs*]: Mechanical verification

General Terms Reliability, Verification

Keywords Concurrency; Refinement; Linearizability

1. Introduction

Effective scalable reasoning about nontrivial software implementations generally requires considering each software module separately, in isolation, using abstract specifications for other modules. When modules are objects whose methods may be called concurrently, their behavior is typically understood in terms of invocation sequences of abstract data types (ADTs): an execution with overlapping method invocations is considered valid when those invocations can be *linearized* into a sequence admitted by the ADT (Herlihy and Wing 1990). For example, consider the execution history depicted in Figure 1 in which the add operations numbered 2 and 3 overlap with each other, and, respectively, with operations 1 and 4. This execution is valid with respect to the atomic queue ADT because among the five possible ways to linearize its six operations, the sequence 1, 3, 2, 4, 5, 6 is admitted. ADT specifications thus decouple reasoning about object implementations from their clients' invocations:

- Is there a valid linearization for each implementation execution?
- Does every valid linearization preserve client invariants?

The former question depends only on a given object's implementation, and the latter only on a given object's clients.

Example 1. *Consider the parallel program in Figure 1 invoking the add and remove methods of an atomic queue implementation, adding increasing integer values w and x tagged with the integers $\{1, 2\}$ indicating on which parallel branch each add occurs. Intuitively this program is correct since values with the same tag are added in increasing order, and, crucially, the values of the queue ADT are removed in the same order in which they are added. Among the six possible ways to linearize these operations, the comparison* `i == j` *of tags only holds for those two beginning with*

```
1: add(a)       #           assume w < x
2: add(b)       ###         (  (q.add(<1,w>); q.add(<1,x>))
3: add(c)         ###       || (q.add(<2,w>); q.add(<2,x>)) )
4: add(d)           #       i, y := q.remove()
5: remove => a        #     j, z := q.remove()
6: remove => c          #   assert i == j ==> y < z
```

Figure 1. An execution history with six numbered operations (left), and a parallel program invoking six operations (right). The "#" symbols depict the time intervals spanned by operations horizontally.

`q.add(<1,w>); q.add(<1,x>)` *and* `q.add(<2,w>); q.add(<2,x>)`

Since the queue ADT dictates that elements are removed in the order added, we conclude that w and x are removed in order when `i == j`*, and thus* `y < z` *holds when* `i == j` *holds.*

Although formal ADT specifications are indispensable for scalable program reasoning, formal-specification writing is a burden for which few programmers posses the required combination of expertise and willingness to overcome. Typically programmers write simple ADT *reference implementations*, e.g., whose methods are synchronized via a global lock, and refine them with more efficient fine-grained implementations, e.g., reducing synchronization bottlenecks using specialized hardware instructions such as atomic compare and swap (CAS).

Our goal in this work is thus the automated generation of formal ADT specifications, derived from reference implementations, which are suitable for automated reasoning. In particular, we aim to generate *symbolic representations* of valid invocation sequences for ADTs which are given implicitly by reference implementations. We target declarative symbolic representations rather than imperative state-based representations in order to harness efficient symbolic reasoning algorithms: rather than enumerating linearizations explicitly, and checking their validity one by one, a symbolic reasoning engine may simultaneously rule out many possible linearizations. Previous work demonstrates that such symbolic reasoning can increase efficiency by orders of magnitude (Emmi et al. 2015).

In this work we demonstrate that effective symbolic ADT representations can be generated automatically from the executions of reference implementations, enabling the application of efficient symbolic reasoning algorithms without the burden of writing formal specifications manually. Our approach exploits two key features of concurrent-object ADTs: that violations of each ADT can be decomposed into a small set of representative patterns, and that these patterns manifest in executions with few operations. The first feature allows us to represent symbolic ADTs finitely, as exclusions of violation patterns. The second allows us to extract violation patterns from finite enumerations of executions.

The fundamental challenge is in identifying the algebraic properties of ADTs which allow us to characterize an infinite set of violating executions with a finite set of patterns. This characterization is non-trivial since an execution with more operations than a given violating execution is not necessarily a violation itself. For instance, an execution which contains only a single pop operation returning the value 1 is a violation to the atomic stack ADT, whereas an execution containing an additional push(1) operation, overlapping in time with the pop, is not. Further complication arises from the infinite set of possible data values, i.e., method argument and return values. Our patterns must be sensitive to the *relation* among data values without being sensitive to the data values themselves. For instance, a sequential execution in which 1 and 2 are pushed and subsequently popped in the same order violates the atomic stack ADT. Yet, while replacing both values 1 and 2 with the value 1 results in a *valid* stack execution, replacing them with 3 and 4 results in a violation.

Our algebraic insight is based on grouping the operations of an execution into *matchings*. Intuitively, operations which refer to the same instances of values belong to the same matching. For example, a pop operation returning the value 1 matches a preceding push(1) operation. By comparing executions by the characteristics of their matchings, rather than the actual data values they use, we capture the relation among data values independently of the data values themselves. Furthermore, the notion of matchings provides a key algebraic property of ADTs: the executions of naturally-occurring concurrent object ADTs are closed under the removal of matchings. For example, any execution of the atomic stack ADT using values 1, 2, and 3 would remain a valid execution were all operations using the value 2 deleted. Conversely, any execution which extends a violating execution with additional matchings is itself a violation. This property, along with analogous algebraic properties concerning operation order and completion, allow us to compare executions via a violation-preserving embedding relation. This relation is a well-founded partial order on executions, and thus allows us to characterize the infinite set of violations to an ADT with a finite basis set, ultimately leading to a finite symbolic representation.

Computing the basis sets of ADT violation patterns is a challenging problem, requiring the computation of global properties of an infinite number of executions — analogously to the inference of inductive invariants. Exploiting a hypothesis that violation patterns manifest in executions with few operations, we propose an under-approximating algorithm which extracts the patterns observed in all violating executions up to a given number of operations. In theory, for an arbitrary ADT, this is clearly incomplete: any violation which only surfaces with a greater number of operations would not be captured, thus resulting in a symbolic ADT representation which can fail to identify violations — though still guarantees never to classify a valid execution as a violation, and is thus sound for program reasoning. Empirically, however, we demonstrate that our hypothesis holds: the patterns emerging from executions with few operations are complete in characterizing all violations of naturally-occurring concurrent object ADTs, thus allowing us to compute complete symbolic ADT representations in practice.

Although our approach does require annotating the operations of concurrent object executions with a matching relation, and we demonstrate that these relations are easy to provide for naturally-occurring ADTs, we also outline an automatic means for computing such *matching schemes*. Again by sampling executions, we leverage automated symbolic reasoning engines to synthesize matching schemes for which given implementations are closed under the removal of matches. This further lowers the burden of automated verification. Rather than providing formal ADT specifications, or even matching schemes, users need only provide the predicates relevant in the logic of matching schemes, and we could automatically compute effective matching schemes, and ultimately, effective symbolic ADT representations.

Algorithm 1: Abstract algorithm for symbolic ADT inference.

```
input : A reference implementation 𝓘
Result: A formula representing the ADT of 𝓘
patterns ← ∅ ;
for each sequential history h do
    if ∗ then
        break
    else if h is executable with 𝓘 then
        continue
    else if h is redundant with patterns then
        continue
    else
        add h to patterns
    end
end
return exclusion of patterns
```

In summary, the contributions and outline of this work are:

- An abstract notion of execution histories based on groups of matching operations (§3).
- The statement of the symbolic ADT inference problem (§4).
- Identification of the algebraic properties allowing a finite characterization of infinite ADT violation sets (§5).
- The symbolic representation of ADT violations (§6).
- The computability of symbolic ADT representations (§7).
- An algorithm to infer the matching schemes required for our algebraic characterization of ADTs (§8).
- An empirical study validating that naturally-occurring ADTs satisfy the properties required for completeness of our inference algorithm, and that our algorithm computes precise symbolic representations thereof (§9).

We conclude with a discussion about the limitations of our approach (§10) and related work (§11).

To the best of our knowledge, this work is the first to suggest the automatic generation of ADT specifications. By removing the burden of writing formal ADT specifications manually, this work broadens the scope of modular program reasoning using efficient symbolic algorithms to newly-designed ADTs, apart from those few traditionally studied in the literature.

2. Overview of an ADT Inference Algorithm

Our basic approach to inferring ADT specifications, as outlined by the abstract algorithm in Algorithm 1, is to identify a finite set of sequential execution histories which capture all of the reasons for which a sequence could be considered invalid, according to the ADT of a given reference implementation. These sequences thus serve as patterns indicating violations in the sequences which contain them. Thus the linearizations which exclude all violation patterns are considered valid.

As an example of this algorithm at work, consider the sequential histories listed in Figure 2. These sequences arise from an enumeration of all 202 possible method invocation sequences[1] of length at most 4 of an object with add and remove methods for which

[1] Here we consider equivalence among sequences which are isomorphic up to renaming of data values, e.g., to avoid considering `add(1); add(2); remove => 1` and `add(3); add(2); remove => 3` as distinct. The notion of *matching* introduced in Section 3 provides a clean formal treatment.

```
[1:X] remove => 1  #
---                                  [1:1] add(1)       #
[1:1] add(1)           #             [2:X] remove => 2    #
[2:2] remove => empty    #           ---
---                                  [1:X] remove => 1  #
[1:2] remove => 1  #                 [2:X] remove => 1    #
[2:2] add(1)         #               ---
---                                  [1:X] remove => 1      #
[1:1] add(1)       #                 [2:2] remove => empty    #
[2:2] add(2)         #               ---
[3:2] remove => 2      #             [1:1] remove => empty  #
---                                  [2:X] remove => 1        #
[1:1] add(1)       #                 ---
[2:1] remove => 1    #               [1:1] add(1)       #
[3:1] remove => 1      #             [2:2] add(2)         #
---                                  [3:X] remove => 3      #
[1:1] add(1)           #             ---
[2:2] remove => empty    #           [1:1] add(1)           #
[3:1] remove => 1          #         [2:2] add(2)             #
---                                  [3:3] remove => empty      #
[1:1] add(1)       #                 ---
[2:2] add(2)         #               [1:1] add(1)       #
[3:2] remove => 2      #             [2:1] remove => 1    #
[4:1] remove => 1        #           [3:X] remove => 2      #
```

Figure 2. Invalid sequences according to the atomic queue ADT. The histories on the right are each redundant with some history on the left, and those on the left constitute a complete set. The "#" symbols depict the time intervals spanned by operations horizontally, and the "[i:j]" notation refers to the current line's operation identifier i, and the identifier j of its matching operation. Adds and empty removes match themselves, and the "X" symbol denotes an absent match.

remove can return empty. The 31 sequences which are executable by a correct reference implementation of an atomic queue — i.e., with consistent return values for each invocation — are discarded. The remaining 171 sequences are invalid, 164 of which are redundant with the seven PATTERNS listed on the left-hand side of Figure 2. Seven of these redundant sequences are shown on the right-hand side of Figure 2. For example, the first five sequences on the right, as well as the last, are redundant with the first on the left since they each describe a violation in which a removed element was never added. The sixth sequence on the right is redundant with the second on the left since they both describe a violation in which remove returns empty before previously-added elements are removed. Other represented violations include removing an element before it is added, and removing elements in the opposite order in which they were added. Finally, the exclusion of the computed pattern set can be expressed as a conjunction of formulas in first order logic describing the exclusion of each individual pattern. For example, the first pattern could be described by the formula

$$\exists o.\ \mathsf{method}(o) = \texttt{remove} \wedge \mathsf{unmatched}(o)$$

describing an unmatched remove operation. The formula which excludes all violation patterns is satisfied by an invocation sequence if and only if that sequence is admitted by the given reference implementation's implicit ADT.

The key technical obstacle in realizing the abstract algorithm of Algorithm 1, overcome in Section 5, is the classification of violations into a finite set of patterns, rendering the remaining invalid sequences redundant. The symbolic representation which excludes a given set of patterns is relatively straightforward to construct, and is given in Section 6. Note however that the termination of this abstract algorithm is nondeterministic, and thus the inferred ADT specification is generally not the strongest possible in the sense that it may not exclude certain invalid linearizations. Nevertheless, specifications inferred by this algorithm are sound, in the sense that when modular program reasoning with inferred specifications succeeds, correctness follows. In any case, in Section 7 we give conditions under which our refinement of this abstract algorithm is both sound and complete, and show that these conditions hold for naturally-occurring ADTs, thus resulting in sound and complete symbolic ADT specifications. The remainder of this article develops the technical machinery required to realize this abstract inference algorithm.

3. Implementations & Histories

Abstract data types (ADT) implementations provide methods which can be invoked concurrently by threads of client programs. We capture the possible histories of call and return actions in the executions of client programs as partially-ordered method invocations. While many ADTs are "atomic" (e.g., queues, locks) in the sense that their ADTs are specified as sets of sequential histories, many (e.g., rendezvous synchronizers, barriers) are non-atomic (Hemed and Rinetzky 2014). While much of the following development could be simplified for atomic ADTs by considering invocation sequences rather than concurrent invocation histories, we maintain generality so that our results apply to non-atomic ADTs as well.

To this end, we fix an arbitrary infinite set $\mathbb{O}$ of operation identifiers, and given sets $\mathbb{M}$ and $\mathbb{V}$ of method names and values. A *call action* binds an operation $o \in \mathbb{O}$ with a method name $p \in \mathbb{M}$ and argument value $v \in \mathbb{V}$, while a *return action* binds an operation $o \in \mathbb{O}$ with a return value $v \in \mathbb{V}$. An *execution* e is a sequence of call and return actions where

- each operation identifier is used in at most one call action, and in at most one return action, and
- each return action is preceded by a call action with the same operation identifier.

An *implementation* $\mathcal{I}$ is a prefix-closed set of executions which is additionally closed under

- appending call actions (of fresh operations),
- permuting call actions backward, and
- permuting return actions forward.

These conditions capture the environment in which implementations execute: calls are always enabled in their client programs, and thread schedulers may induce arbitrary delay between implementation code and the associated call and return actions (Bouajjani et al. 2015a).

Example 2. *Consider the following three executions of a single-value register object with read and write methods:*

e_1	e_2	e_3
1: call write(a)	1: call write(a)	1: call write(a)
1: return	2: call write(b)	1: return
2: call write(b)	1: return	2: call write(b)
2: return	2: return	3: call read
3: call read	3: call read	2: return
3: return => a	3: return => a	3: return => a

Operation identifiers precede actions. Executions e_2 and e_3 are obtained from e_1, respectively, by permuting the return actions of Operations 1 and 2 and the call actions of operations 2 and 3. Thus while the operations of Execution e_1 are sequential, each following the previous in time, those of e_2 and e_3 overlap. While e_1 should not be admitted by an atomic register, since the read of Operation 3 does not return the most-recently-written value, both e_2 and e_3 should be admitted, since Operation 3 returns the most-recently-written value in some linearization of the overlapping operations.

Histories abstract executions, retaining method names yet losing exact argument and return values, and retaining the relative order of

operations, yet losing the exact sequence of call and return actions. Formally, a *history* is a tuple $h = \langle O, <, c, f, m, r\rangle$ where

- $O \subseteq \mathbb{O}$ is a set of *operations*,
- $<$ is a *happens-before* interval order[2] on O,
- $c : O \to \mathbb{B}$ labels operations as *completed*, or not,
- $f : O \to \mathbb{M}$ labels operations with method names,
- $m : O \rightharpoonup O$ is a partial *matching* function, and
- $r : O \to \mathbb{B}$ labels operations as *read-only*, or not.

The relation $<$ is an interval order because executions' call and return actions are totally ordered (Bouajjani et al. 2015a). Non-completed operations are *pending*, and are maximal in the happens-before order.

Two operations $o_1, o_2 \in O$ are *identical* when

- they have the same labels: $c(o_1) = c(o_2)$, $f(o_1) = f(o_2)$, and $r(o_1) = r(o_2)$, and
- they have the same matching: either $m(o_1) = m(o_2)$, or both $m(o_1)$ and $m(o_2)$ are undefined.

The *frequency* of an operation o is the number of operations identical to o, denoted $\mathrm{freq}(o)$. We say that o has *duplicates* when $\mathrm{freq}(o) > 1$. An operation $o \in \mathrm{img}(m)$ in the image of m is a *match target*, and the inverse set $m^{-1}(o)$ of a target is a *match*. We assume $o \in m^{-1}(o)$ for every o. A operation o is *unmatched* when o is completed and $m(o)$ is undefined.

The *width* of a history is the maximum number of its operations which are mutually unordered, and the *width* of a history set is the maximum width of its elements. Width-1 histories are *sequential*. Since the operation identifiers of a history have no intrinsic meaning, we consider equality between histories up to renaming of operation identifiers.

Example 3. *We draw histories by writing one operation per line, starting with its operation identifier and match target, followed by its method label, possibly a read-only marker, and its happens-before interval. For instance, in the history*

```
[1:1] write(a)        #
[2:1] read => a (RO)    #
```

Operation 1 *is a write operation which matches itself, and precedes a read-only read operation which also targets Operation* 1*. Note that we label method argument and return values for illustrative purposes only; histories do not keep them. In the following history, Operations* 1 *and* 2 *execute concurrently*

```
[1:1] write(a)        #
[2:1] read => a (RO)  #
[3:X] read => b (RO)    #
[4:_] read*     (RO)      #
```

Operation 3 *is unmatched, indicated by the* `X`*, and Operation* 4 *is pending, indicated by the* `*` *on its label.*

Our abstraction of executions relies on correlating the operations associated with the same values in an execution via the partial matching function of a history. We construct partial matching functions systematically. A *matching scheme* $\langle M, R\rangle$ associates to each execution e with operations O a partial matching function $M(e) : O \rightharpoonup O$ and a read-only operation labeling $R(e) : O \to \mathbb{B}$.

Example 4. *Consider the following matching scheme for the read and write operations of a single-value register object:*

- *write*(v) *operations match themselves, and*
- *read* $\Rightarrow v$ *operations are read-only, and match themselves when they return* $v = -$*, or a write*(v) *operation, if one exists, and otherwise have no match,*

which is well defined when each value $v \in \mathbb{V}$ *appears as the argument of at most one write operation in any execution. This matching scheme corresponds to the matching functions in the histories of Example 3.*

The history $H(e, M, R)$ of an execution e under matching scheme $\langle M, R\rangle$ is the tuple $\langle O, <, c, f, M(e), R(e)\rangle$ where

- O are the operations of e,
- $o_1 < o_2$ iff operation o_1 returns before o_2 is called in e,
- $c(o)$ iff operation o returns in e, and
- $f(o)$ is the name of the method executed by o in e.

We denote the set $\{H(e, M, R) : e \in E\}$ of histories of an execution set E by $H(E, M, R)$. When the matching scheme $\langle M, E\rangle$ is clear from the context, we abbreviate $H(e, M, R)$ and $H(E, M, R)$ by $H(e)$ and $H(E)$.

Example 5. *The histories of the executions of Example 2 according to the matching scheme of Example 4 are* $H(e_1)$*:*

```
[1:1] write(a)        #
[2:2] write(b)          #
[3:1] read => a (RO)      #
```

in which all three operations are sequential, $H(e_2)$*:*

```
[1:1] write(a)        #
[2:2] write(b)        #
[3:1] read => a (RO)    #
```

in which the first two operations overlap, and $H(e_3)$*:*

```
[1:1] write(a)        #
[2:2] write(b)          #
[3:1] read => a (RO)    #
```

in which the last two operations overlap.

A matching scheme $\langle M, R\rangle$ is *faithful* to a set E of executions when $e \in E$ iff $e' \in E$ for any two executions e and e' such that $H(e, M, R) = H(e', M, R)$. A set of executions E is *data independent*[3] when there exists a faithful matching scheme. By definition, our abstraction of executions into histories incurs no loss of precision for data-independent execution sets.

Lemma 1. *$H(e, M, R) \in H(E, M, R)$ if and only if $e \in E$, for any faithful matching scheme $\langle M, R\rangle$.*

In Section 9 we demonstrate faithful matching schemes for the executions of naturally-occurring ADTs, and in Section 8 we demonstrate how to infer faithful matching schemes. Otherwise, for the remainder of this work, we assume each set of executions comes equipped with a faithful matching scheme $\langle M, R\rangle$.

Two histories h_1 and h_2 are related by $\to_x$, for $x = \mathrm{o}, \mathrm{c}, \mathrm{p}$, when h_2 is obtained from h_1 by:

- unordering a pair of ordered operations (o),
- making a completed operation pending (c), or
- adding a pending operation (p).

A set of histories H is *closed* under a relation $\to$ when $h_2 \in H$ whenever $h_1 \to h_2$ and $h_1 \in H$. A fundamental property of implementations is that their histories are closed under weakening via less ordering, fewer operations completed, and additional pending operations (Bouajjani et al. 2015a).

[2] An interval order (Fishburn 1985) is a partial order whose elements can be mapped to integral intervals preserving the order relation, or equivalently, a partial order for which $w < x$ and $y < z$ implies $w < z$ or $y < x$.

[3] Our definition of *data independence* formalizes an existing informal notion, which stipulates that the implementation generating a set of executions does not predicate its actions on the data values passed as method arguments.

Example 6. *By un-ordering the first two operations of the history*

```
[1:1] write(a)        #
[2:2] write(b)          #
[3:2] read => b (RO)      #
```

we derive the $\rightarrow_{\mathrm{o}}$-related history

```
[1:1] write(a)        #
[2:2] write(b)        #
[3:2] read => b (RO)    #
```

from which we can derive the $\rightarrow_{\mathrm{c}}$-related history

```
[1:1] write(a)        #
[2:2] write(b)        #
[3:_] read* (RO)        #
```

by making Operation 3 pending, and from which we can derive the $\rightarrow_{\mathrm{p}}$-related history

```
[1:1] write(a)        #
[2:2] write(b)        #
[3:_] read* (RO)        #
[4:4] write(c)*         #
```

by adding an additional pending operation.

As these weakening operations align with the environment-capturing closure properties on executions, the set of histories of an implementation is also closed.

Lemma 2. *$H(\mathcal{I})$ is closed under $\rightarrow_{\mathrm{o}}$, $\rightarrow_{\mathrm{p}}$, $\rightarrow_{\mathrm{c}}$.*

4. The Symbolic ADT Inference Problem

In this section we formalize a notion of abstract data type and define the corresponding refinement and inference problems. These are the foundational problems addressed in this work.

A Set K *generates* H when the closure of K under the relation $\rightarrow \;=\; (\rightarrow_{\mathrm{o}} \cup \rightarrow_{\mathrm{p}} \cup \rightarrow_{\mathrm{c}})$ is equal to H, i.e., $H = \{h : \exists h' \in K.\, h' \rightarrow^* h\}$. A *kernel* of a set H is a minimal set generating H. While the kernels of arbitrary sets need not be unique, the kernels of sets which have sequential kernels are unique. Furthermore, Section 9 demonstrates that the histories of naturally-occurring implementations have unique kernels, which we assume for the remainder of this work. An *abstract data type (ADT)* A is the kernel of the set $H(\mathcal{I})$ of histories of some implementation $\mathcal{I}$; we say that *A is the ADT of $\mathcal{I}$*.

ADTs and reference implementations serve as specifications to more efficient implementations in the sense that they limit the set of histories that efficient implementation may admit. This notion of refinement ensures that client program (safety) properties which hold using the ADT or reference implementation also hold using refined implementations (Bouajjani et al. 2015a).

Definition 1. *An implementation $\mathcal{I}_1$* refines *another implementation $\mathcal{I}_2$ when $H(\mathcal{I}_1) \subseteq H(\mathcal{I}_2)$. An implementation $\mathcal{I}$ refines an ADT A when $H(\mathcal{I}) \subseteq A^*$.*

Recent works demonstrate efficient[4] refinement-checking algorithms (Bouajjani et al. 2015a; Emmi et al. 2015) yet rely on hand-written symbolic ADT representations. In order to frame the problem of computing these automatically, we fix a language for symbolic representation. A *history formula* is a first-order logic formula with

- variables ranging over operation identifiers,
- constants from $\mathbb{M}$ for method names,
- function symbols f and m for labels and matching, and
- predicate symbols c, um, r, and $<$ for completion, non-matching (operations which are not in the domain of the matching function), read-only, and order.

[4] In time polynomial in the number of operations, per execution.

A history formula F is interpreted over a history h in the natural way, by binding variables to the operations of h, and binding function and predicate symbols to their interpretations in h. We write $h \models F$ when h is a model of F, and $h \not\models F$ otherwise.

Example 7. *The following history formula is satisfied by histories in which no write operation happens between a pair of matching write and read operations:*

$$\begin{aligned}\forall x_1, x_2, x_3.\ \mathsf{c}(x_1) \wedge \mathsf{f}(x_1) = \mathrm{write} \wedge \mathsf{m}(x_1) = x_1 \\ \wedge\, \mathsf{c}(x_2) \wedge \mathsf{f}(x_2) = \mathrm{write} \wedge \mathsf{m}(x_2) = x_2 \\ \wedge\, \mathsf{c}(x_3) \wedge \mathsf{f}(x_3) = \mathrm{read} \wedge \mathsf{m}(x_3) = x_1 \\ \wedge\, x_1 < x_2 \Rightarrow x_3 < x_2\end{aligned}$$

This is one of several requirements of atomic single-value register ADTs. It is satisfied by certain linearizations of the histories $H(e_2)$ and $H(e_3)$ from Example 5, yet not $H(e_1)$.

The *bounded complement* of a history set H of width $k \in \mathbb{N} \cup \{\omega\}$ is the set of histories of width at most k which are excluded from H. Let A be an ADT and B its bounded complement. We say that a history formula F *represents* A when

- $h \models F$ for all $h \in A$, and
- $h \not\models F$ for all $h \in B$.

ADT inference is to compute a formula representing an ADT.

Definition 2. *The* symbolic ADT inference problem *is to compute a history formula representing the ADT of a given implementation.*

Computing a history formula representing the ADT of a reference implementation $\mathcal{I}$ thus enables efficient modular program reasoning without the burden of writing precise formal specifications for $\mathcal{I}$.

5. Finite ADT Representations

In this section we demonstrate that naturally-occurring ADTs can be precisely represented by finite sets of histories, despite the fact that these ADTs admit infinite sets of histories. This results relies on the identification of certain algebraic properties of the sets of histories admitted by ADTs. In particular, we find that sets of histories admitted by ADTs are closed under the removal of certain operations, and that these sets adhere to a well-founded ordering under the relation induced by such removals.

In addition to the relations $\rightarrow_{\mathrm{o}}$, $\rightarrow_{\mathrm{p}}$, and $\rightarrow_{\mathrm{c}}$ of Section 3 under which all implementation history sets are closed, the histories of ADT implementations we consider in this work are also closed under additional relations which remove read-only operations, unmatched operations, entire matches, and duplicate operations. The relations $\rightarrow_{\mathrm{r}}$, $\rightarrow_{\mathrm{u}}$, $\rightarrow_{\mathrm{m}}$ and $\rightarrow_{\mathrm{d}}$ relate two histories h_1 and h_2 when h_2 is obtained from h_1 by

- removing a read-only operation (r),
- removing an unmatched operation (u),
- removing a match (m), or
- removing a duplicate operation (d).

We say an ADT whose histories are closed under $\rightarrow_{\mathrm{r}}$, $\rightarrow_{\mathrm{u}}$, $\rightarrow_{\mathrm{m}}$ and $\rightarrow_{\mathrm{d}}$ is *normal*. In Section 9 we demonstrate that naturally-occurring ADTs are normal. Defining the relation $\succeq$ as the reflexive and transitive closure of the above relations,

$$\succeq \;=\; (\rightarrow_{\mathrm{o}} \cup \rightarrow_{\mathrm{p}} \cup \rightarrow_{\mathrm{c}} \cup \rightarrow_{\mathrm{r}} \cup \rightarrow_{\mathrm{u}} \cup \rightarrow_{\mathrm{m}} \cup \rightarrow_{\mathrm{d}})^*$$

closure under $\succeq$ follows immediately.

Lemma 3. *Normal ADTs are closed under $\succeq$.*

Besides this closure property, the $\preceq$ relation enjoys a certain notion of well-foundedness when restricted to bounded-width histories:

the set of $\preceq$-minimal elements of any (potentially infinite) history set is finite. This property is what enables us to represent infinite sets of invalid ADT histories with a finite set of minimal examples. This property is captured formally with wqos: a *well-quasi-ordering (wqo)* R on a set X is a reflexive, transitive binary relation on X for which in every infinite sequence $x_0 x_1 \ldots$ of elements from X, there exists indices $i < j$ such that $R(x_i, x_j)$.

Example 8. *Consider the infinite history sequence $h_1 h_2 \ldots$ where each h_i contains $2i$ operations $o_1, o_1', \ldots, o_i, o_i'$ where each o_j is a completed* `push` *operation matching itself, and each o_j' is a pending* `pop` *operation with undefined matching. Because each successive h_i has both more matches and more pending operations, no two histories of the sequence are related by $\preceq$. Thus $\preceq$ is not a wqo.*

This example demonstrates that $\preceq$ is not a wqo by allowing each history h_i of the infinite sequence to contain more and more pending operations in order to ensure that $h_j \not\preceq h_i$ for every $j < i$. Curbing this ability by limiting the maximum amount of pending operations per history makes $\preceq$ a wqo. Although limiting to k pending operations essentially limits us to width-k histories, of executions with at most k operations parallel at any moment, e.g., of programs with at most k threads, this restriction comes at no loss of completeness when considering only the histories of bounded-width ADTs, which is the subject of the remainder of this section.

Lemma 4. *$\preceq$ is a wqo on bounded-width histories.*[5]

For the remainder of this section, we fix an ADT A, and let B be its bounded complement. When A has bounded width, so does B, and thus $\preceq$ is a wqo on B. Furthermore, when A normal, it is closed under $\succeq$, and thus B is closed[6] under $\preceq$. Closure under a relation satisfying Lemma 4 implies representation by a finite set. Formally, we say a set X is *finitely representable* if there exists a finite set Y and a relation $R \subseteq Y \times X$ such that $X = \{x : \exists y \in Y.\ R(y, x)\}$. In our case, we obtain a finite set from which exactly the elements of B are related by $\preceq$.

Lemma 5. *B is finitely representable if A is normal.*

Example 9. *The following four histories generate the complement of the atomic single-value register ADT, witnessing either an unmatched read operation:*

```
[1:X] read => 1 (RO)  #
```

a read of an uninitialized register occurring after some write:

```
[1:1] write(1)        #
[2:2] read => - (RO)    #
```

a read which happens before its matching write operation:

```
[1:2] read => 1 (RO)  #
[2:2] write(1)          #
```

and a read matching a write which is not the most recent:

```
[1:1] write(1)        #
[2:2] write(2)          #
[3:1] read => 1 (RO)      #
```

Every sequential history not admitted by the atomic register ADT embeds at least one of these four histories.

6. Symbolic ADT Representations

While Section 5 demonstrates that (the complements) of naturally-occurring ADTs can be represented finitely, in this section we demonstrate that such finite representations have logical interpretations, allowing us to derive the formulas representing ADTs. In what follows, we describe how to obtain a formula which is satisfied by the histories which embed a given history, via $\preceq$. Then, using the finite set of histories which represent the complement of a given ADT, we represent the ADT itself as the conjunction of negations of these embedding formulas. The resulting formula is satisfied only on the histories which do not embed the generators of a given ADT's complement.

Let $h = \langle O, <, c, f, m, r\rangle$ be a history with operations $O = \{o_1, \ldots, o_n\}$. Without loss of generality, we assume the match targets $\{o_1, \ldots, o_k\}$ of h are indexed consecutively from 1 to k, for some $k \leq n$. For each $o \in O$, we define the macro $\textsc{Embed}_o(x, Y, z)$:

$$\begin{aligned}&\big(c(o) \Rightarrow \mathsf{c}(x)\big) \wedge \mathsf{f}(x) = f(o) \wedge \mathsf{r}(x) \Leftrightarrow r(o)\\ &\wedge \big(o \in \mathrm{dom}(m) \Rightarrow \neg\mathsf{um}(x) \wedge \mathsf{m}(x) = z\big)\\ &\wedge \big(o \notin \mathrm{dom}(m) \Rightarrow \mathsf{um}(x)\big) \wedge \bigwedge_{y \in Y} x < y\end{aligned}$$

capturing the correspondence between the operation o and the logical variable x representing o. The variables Y and z represent the operations ordered after x, and the operation which x matches. The macro $\textsc{Identical}(x, y)$:

$$\begin{aligned}&\big(\mathsf{c}(x) \Leftrightarrow \mathsf{c}(y)\big) \wedge \mathsf{f}(x) = \mathsf{f}(y) \wedge \big(\mathsf{r}(x) \Leftrightarrow \mathsf{r}(y)\big)\\ &\wedge \big(\mathsf{um}(x) \Leftrightarrow \mathsf{um}(y)\big) \wedge \mathsf{m}(x) = \mathsf{m}(y)\end{aligned}$$

captures whether the operations bound to x and y are identical. To express the constraints among the matches of embedded operations, we define the macro $\textsc{Match}(Y, z)$:

$$\forall x.\mathsf{m}(x) = z \Rightarrow \mathsf{r}(x) \vee \bigvee_{y \in Y} \textsc{Identical}(x, y)$$

which requires any operation which matches z to be either read-only, or identical to some operation in Y, which represents the operations of h which match z. Finally, we express the embedding of h with the macro $\textsc{Embed}_h$:

$$\exists x_1, \ldots, x_n.\ \bigwedge_{i=1}^{n} \textsc{Embed}_{o_i}(x_i, Y_i, z_i) \wedge \bigwedge_{i=1}^{k} \textsc{Match}(W_i, x_i)$$

where $Y_i = \{x_j : o_i < o_j\}$ are the variables corresponding to operations ordered after o_i, and z_i is the variable corresponding to $m(o_i)$, if defined, and $W_i = \{x_j : m(o_j) = o_i\}$ are the variables corresponding to operations matching o_i.

Example 10. *Consider again the histories of Example 9 which generate the complement of the atomic register ADT. The* Embed *formula for the first history,*

```
[1:X] read => 1 (RO)  #
```

after simplifications, like replacing $\mathrm{true} \Rightarrow p$ *with* p*, is*

$$\exists x_1.\ \mathsf{c}(x_1) \wedge \mathsf{f}(x_1) = \mathrm{read} \wedge \mathsf{um}(x_1) \wedge \mathsf{r}(x_1).$$

The Embed *formula for second history,*

```
[1:1] write(1)          #
[2:2] read => - (RO)    #
```

is similarly given by

$$\begin{aligned}&\exists x_1, x_2.\ x_1 < x_2\\ &\quad \wedge \mathsf{c}(x_1) \wedge \mathsf{f}(x_1) = \mathrm{write} \wedge \neg\mathsf{um}(x_1) \wedge \mathsf{m}(x_1) = x_1\\ &\quad \wedge \mathsf{c}(x_2) \wedge \mathsf{f}(x_2) = \mathrm{read} \wedge \neg\mathsf{um}(x_2) \wedge \mathsf{m}(x_2) = x_2 \wedge \mathsf{r}(x_2)\\ &\quad \wedge \big(\forall x.\ \mathsf{m}(x) = x_1 \Rightarrow \mathsf{r}(x) \vee \textsc{Identical}(x, x_1)\big)\\ &\quad \wedge \big(\forall x.\ \mathsf{m}(x) = x_2 \Rightarrow \mathsf{r}(x) \vee \textsc{Identical}(x, x_2)\big).\end{aligned}$$

The formulas for the other histories are similarly obtained.

Lemma 6. *$h_1 \models \textsc{Embed}_{h_2}$ iff $h_1 \succeq h_2$.*

[5] The proof of Lemma 4 appears in Appendix A.

[6] Actually, B is closed under $\preceq$ when restricted to width-bounded histories, i.e., if $h_1 \preceq h_2$, $h_1 \in B$, and h_2 is width-bounded, then $h_2 \in B$.

We obtain a formula representing an ADT by taking the conjunction of the negative embedding formulas $\{\neg\text{EMBED}_h : h \in H\}$ from any set H that generates its bounded complement B, i.e., whose closure under $\preceq$, denoted by H^* henceforth, is equal to B. We define the *exclusion formula* of a set of histories H as $F(H) = \bigwedge_{h \in H} \neg\text{EMBED}_h$.

Lemma 7. *Let B be the bounded complement of a bounded-width ADT A, and H a set of histories. $F(H)$ represents A if $H^* = B$.*

Proof. For a set Σ, let $\neg\Sigma$ denote its complement. By Lemma 3, the complement of A is closed under $\preceq$. Since $H \subseteq B \subseteq \neg A \subseteq \neg A$, we have that $H^* \subseteq \neg A$. By Lemma 6, the formula $F(H)$ holds exactly for the set of histories $\neg H^*$. By the previous inclusions, we have that $A \subseteq \neg H^* \subseteq \neg B$. Therefore, $F(H)$ holds for all histories in A and doesn't hold for histories in B. □

Example 11. *The conjunction of negations of the* EMBED *formula for the histories of Example 9, which are partially written in Example 10, represents the atomic register ADT.*

7. A Symbolic ADT Inference Algorithm

Algorithm 2 solves the symbolic ADT inference problem by computing a finite representation of an ADT complement B using the $\preceq$ relation of Section 5. In general, this is achieved by enumerating the elements of B while maintaining the $\preceq$-minimal elements, and recognizing a condition under which all the elements of B are related to the current set of minimals (Abdulla et al. 1996; Finkel and Schnoebelen 2001). In our case, we stratify our enumeration of B by the relative sizes of its histories. Formally, we say the *weight* of a history is the maximum among operation frequencies and the number of matches. We then define

$$B_i = \{h \in B : \text{weight}(h) \leq i\}$$

$$B'_i = \{h \in B : \exists h' \in B_i.\ h' \preceq h \text{ and } \text{weight}(h) \leq i + 1\}$$

respectively as the histories of B with at most i matches and duplicates, and those derived from B_i with at most $i + 1$ matches and duplicates. We say that an ADT with complement B is *predictable* if $B_i^* = B$ whenever $B'_i = B_{i+1}$, i.e., if all histories of B are represented by B_i whenever all histories of B_{i+1} are represented by B_i. Algorithm 2 then performs a weight-increasing enumeration of B, collecting $\preceq$-minimals from smaller-weight violations before advancing to greater weights. When no violation is found at a given weight, the algorithm terminates. This algorithm is guaranteed to terminate since $\preceq$ is a wqo (Abdulla et al. 1996; Finkel and Schnoebelen 2001). Furthermore, this algorithm is sound for arbitrary ADTs, and complete for predictable ADTs. Many naturally-occurring ADTs are predictable — in fact all of the examples we know of are predictable, as demonstrated in Section 9.

Theorem 1. *Algorithm 2 terminates. If the input implementation's ADT is predictable, then the returned formula represents it.*

Proof. Termination is a direct consequence of Lemma 4, and since the number of histories of any weight $i \in \mathbb{N}$ is finite[7] each B_i is computable. When A is predictable, then $B = \text{patterns}^*$, and thus by Lemma 6, the returned formula $F(\text{patterns})$ represents A. □

For non-predictable ADTs, the value of patterns* upon termination of Algorithm 2 is an under-approximation of the bounded ADT complement B. The resulting symbolic ADT representation is still satisfied by all histories in A, therefore it would still be sound for modular program reasoning, identifying only actual violations, and implying the correctness of client programs which do not depend on the stronger criteria which excludes unidentified violations. However, the under-approximation may be incomplete in identifying all violations, and in proving client programs which depend on the stronger criteria which excludes them all.

[7] As noted in Section 3, we consider equality between histories up to renaming of operation identifiers.

Algorithm 2: Symbolic ADT inference.

Input : A reference implementation $\mathcal{I}$ of width k
Result: A formula representing the ADT of $\mathcal{I}$
patterns $\leftarrow \emptyset$;
$w \leftarrow 0$;
repeat
  none-found $\leftarrow$ true ;
  for *each k-width history h with weight w* **do**
    if *h is* executable *with $\mathcal{I}$* **then**
      continue
    else if $\exists h' \in$ *patterns*. $h' \preceq h$ **then**
      continue
    else
      add h to patterns ;
      none-found $\leftarrow$ false
    end
  end
  $w \leftarrow w + 1$;
until *none-found*;
return $F(\text{patterns})$

8. Matching Scheme Inference

Our solution to the ADT inference problem relies on identifying faithful matching schemes for ADT implementations. Though Section 9 shows such matching schemes exist for naturally-occurring ADTs, the next natural question with regard to automation is whether these matching schemes can be generated automatically.

In this section we demonstrate that matching schemes can in fact be generated systematically with minimal manual specification by reduction to logical satisfiability. Essentially, we formulate matching schemes as uninterpreted functions in satisfiability queries constrained by the requirement that they *normalize* an implementation's executions, i.e., generate histories which are closed under the operation removals of Section 5. This requirement is given with respect to an enumeration of execution pairs in which the first is admitted by the given implementation, and the second is not, yet it is a projection of the first's operations. Consequently, this prohibits any normalizing matching scheme from considering the projected operations a match. In what follows, we suppose the read-only component R of matching schemes $\langle M, R\rangle$ is given, and focus on the generation of the per-execution matching functions M.

The required manual specification includes identifying a class of executions to which an implementation should be subjected, and a set of predicates that are required in the logic of matching schemes. Formally, a *language* $L = \langle E, P, Q\rangle$ is a set E of executions, along with finite sequences $P = P_1P_2\ldots$ and $Q = Q_1Q_2\ldots$ of binary and ternary predicates $P_i(e, o_1)$ and $Q_i(e, o_1, o_2)$ ranging over the executions of E and their operations. When P is a k-length sequence of n-ary predicates, we write $P(x_1, \ldots, x_n)$ to denote the valuation sequence

$$P_1(x_1, \ldots, x_n)\ldots P_k(x_1, \ldots, x_n).$$

Given a language $L = \langle E, P, Q\rangle$, we say a matching scheme M is *simple* when there exists an n-ary Boolean function G, for $n = 2 \cdot |P| + |Q|$, such that for each execution $e \in E$

- $G(P(e,o_1),P(e,o_2),Q(e,o_1,o_2))$ is satisfied for at most one operation o_1, for any operation o_2,
- $M(e)(o_2)$ is undefined unless there exists an operation o_1 for which $G(P(e,o_1),P(e,o_2),Q(e,o_1,o_2))$, and
- $M(e)(o_2)=o_1$ iff $G(P(e,o_1),P(e,o_2),P(e,o_1,o_2))$,

where o_1 and o_2 range over the operations of e.

Example 12. *We say an execution of read and write methods* writes unique values *if the argument value to each write operation is unique. Consider the language whose executions write unique values, with the following predicates:*

$\mathsf{w}(e,o)$	*o is a* write *operation in e, and,*
$\mathsf{v}(e,o_1,o_2)$	*o_1 and o_2 read/write the same values.*

We define the function $G(x_\mathsf{w},y_\mathsf{w},z_\mathsf{v})$ over valuations to the predicates above to be satisfied if and only if:

$$x_\mathsf{w}\wedge z_\mathsf{v}$$

Intuitively, this defines a simple matching scheme for which write operations match themselves, and read operations match the write which wrote the value read. In the case such a write exists, it is unique in any execution which writes unique values. The match is otherwise undefined. This scheme is exactly that which is specified in Example 4 of Section 3.

An implementation $\mathcal{I}$ *adheres* to a language $L=\langle E,P,Q\rangle$ if $\mathcal{I}\subseteq E$. A match scheme M *normalizes* an implementation $\mathcal{I}$ when M is faithful to $\mathcal{I}$ and $H(\mathcal{I},M)$ is normal.

Definition 3. *The* matching scheme inference problem *is to compute a simple matching scheme M which normalizes a given implementation $\mathcal{I}$ adhering to a given language L.*

We automate the computation of matching schemes by constructing a logical formula that characterizes the boolean functions G underlying a simple matching scheme. These boolean functions are defined as the interpretation of a function symbol g. The formula expresses the fact that the interpretations of g uniquely determine the match of each operation, and that they normalize the executions of an implementation. To this end, we fix an implementation $\mathcal{I}$ and a language $L=\langle E,P,Q\rangle$ to which $\mathcal{I}$ adheres, then consider any enumeration F of execution pairs $\langle e,e'\rangle\in E^2$ such that

- $e\in\mathcal{I}$ and $e'\notin\mathcal{I}$, and
- e' is obtained by deleting operations of e.

Any such pair of executions can be used to rule out several possibilities, e.g., that the operations removed from e to obtain e':

- do not constitute a match in e,
- are not all duplicate operations,
- are not all read-only operations,
- do not constitute multiple matches in e,

and so on. Ruling out these possibilities is sound since, for example, a normalized scheme M could not consider those operations a match: otherwise the history abstraction $H(\mathcal{I},M)$ which includes $H(e,M)$ must also include $H(e',M)$, being normal, and in particular closed under match removal. Such an M would thus not be faithful. For simplicity, in what follows we consider ruling out only the first possibility: that the operations removed from e are not a match. In principle, the approach extends to rule out all possibilities.

In what follows, we denote the operations of an execution e by O_e. To consider whether a given pair o_1,o_2 of operations of an execution e is a match according to the simple matching scheme based on g, for the given language L, we define the macro

$\textsc{IsMatch}_{e,o_1,o_2}$:

$$\mathsf{g}\big(P(e,o_1),P(e,o_2),Q(e,o_1,o_2)\big).$$

To enforce that each operation of an execution e will have a unique match, we define the macro $\textsc{UniqueMatch}_e$:

$$\bigwedge_{o_1,o_2\in O_e}\Big(\textsc{IsMatch}_{e,o_2,o_2}\Rightarrow\bigwedge_{o_1'\neq o_1}\neg\textsc{IsMatch}_{e,o_1',o_2}\Big).$$

Then a given set $O\subseteq O_e$ constitutes a match according to g when all operations $o_2\in O$ target some operation $o_1\in O$, and no other operation $o_2\in O\setminus O_e$ does. We express this with the macro $\textsc{EntireMatch}_{e,O}$:

$$\bigvee_{o_1\in O}\Big(\bigwedge_{o_2\in O}\textsc{IsMatch}_{e,o_1,o_2}\wedge\bigwedge_{o_2\in O_e\setminus O}\neg\textsc{IsMatch}_{e,o_1,o_2}\Big)$$

Finally, given a pair $\langle e,e'\rangle\in F$, we prohibit the operations $O_e\setminus O_{e'}$ from constituting a match according to g, since if g normalized $\mathcal{I}$, and $e\in\mathcal{I}$, then e' should also be in $\mathcal{I}$. We express this exclusion for all pairs of F with the macro $\textsc{Normalizes}_F$:

$$\bigwedge_{\langle e,e'\rangle\in F}\Big(\textsc{UniqueMatch}_e\wedge\neg\textsc{EntireMatch}_{e,(O_e\setminus O_{e'})}\Big).$$

We thus check satisfiability for the conjunction of non-matches $O_e\setminus O_{e'}$ over all pairs $\langle e,e'\rangle\in F$.

Example 13. *Consider again the language L_{reg} of read and write methods of Example 12 with predicates $\mathsf{w}(e,o)$ and $\mathsf{v}(e,o_1,o_2)$ whose executions write unique values. The following table lists all possible predicate valuations $\langle x_\mathsf{w},y_\mathsf{w},z_\mathsf{v}\rangle$, and for each valuation a valid execution which excludes the positive valuation of $\mathsf{g}(x_\mathsf{w},y_\mathsf{w},z_\mathsf{v})$ in the satisfaction of $\textsc{Normalize}_F$, in the case such an execution exists.*

x_w	y_w	z_v	*counterexample*	*reason*
0	0	0	$\mathrm{w}(1)\ \mathrm{r}(1)\ \mathrm{r}(1)\ \mathrm{w}(2)\ \mathrm{r}(2)$	*not unique*
0	0	1	$\mathrm{w}(1)\ \mathrm{r}(1)\ \mathrm{r}(1)$	*not unique*
0	1	0	$\mathrm{w}(1)\ \overline{\mathrm{r}(1)\ \mathrm{w}(2)}\ \mathrm{r}(2)$	$\rightarrow_{\mathrm{m}}$
0	1	1	$\overline{\mathrm{w}(1)\ \mathrm{r}(1)\ \mathrm{r}(1)}$	$\rightarrow_{\mathrm{m}}$
1	0	0	$\overline{\mathrm{w}(1)\ \mathrm{w}(2)}\ \mathrm{w}(3)\ \mathrm{r}(3)$	*not unique*
1	0	1	—	—
1	1	0	$\mathrm{w}(1)\ \mathrm{w}(2)\ \mathrm{w}(3)$	*not unique*
1	1	1	—	—

For example, the first valuation 000 *must be excluded since it allows the read* $\mathrm{r}(2)$ *to match both reads* $\mathrm{r}(1)$*, violating* UniqueMatch*. The second valuation* 001 *must also be excluded, since it allows either read* $\mathrm{r}(1)$ *to match both itself and the other read* $\mathrm{r}(1)$*. The third valuation* 010 *must be excluded since it allows the write* $\mathrm{w}(2)$ *to match the read* $\mathrm{r}(1)$*, in which case removing the match* $\{\mathrm{r}(1),\mathrm{w}(2)\}$ *results in the invalid execution* $\mathrm{w}(1)\ \mathrm{r}(2)$*. Reasoning follows similarly for the other rows. In this way, any enumeration F which includes the above executions will exclude all valuations except for* 101 *and* 111*, ultimately resulting in a satisfiable* $\textsc{Normalize}_F$ *in which* $\mathsf{g}(x_\mathsf{w},y_\mathsf{w},z_\mathsf{v})$ *is the same Boolean function $x_\mathsf{w}\wedge z_\mathsf{v}$ given in Example 12.*

On the one hand, checking satisfiability of $\textsc{Normalizes}_F$ can be used to conclude the impossibility of a good matching scheme — at least for the given language.

Lemma 8. *If* $\textsc{Normalizes}_F$ *is unsatisfiable, then there exists no simple matching scheme that normalizes $\mathcal{I}$ for the language L.*

The reason for unsatisfiability can be used as a counterexample to refine the given language, e.g., by adding additional predicates.

Example 14. *Consider again the language L_{reg} of read and write methods of Example 12, yet this time without the predicate*

$\mathsf{v}(e, o_1, o_2)$. *The following table lists all possible predicate valuations* $\langle x_\mathsf{w}, y_\mathsf{w} \rangle$*, and for each valuation a valid execution which excludes the positive valuation of* $\mathsf{g}(x_\mathsf{w}, y_\mathsf{w})$ *in the satisfaction of* NORMALIZE$_F$.

x_w	y_w	*counterexample*	*reason*
0	0	$\mathrm{w}(1)\ \mathrm{r}(1)\ \mathrm{r}(1)$	*not unique*
0	1	$\overline{\mathrm{w}(1)}\ \mathrm{r}(1)\ \mathrm{r}(1)$	$\rightarrow_\mathrm{m}$
1	0	$\overline{\mathrm{w}(1)}\ \overline{\mathrm{w}(2)}\ \mathrm{r}(2)$	*not unique*
1	1	$\mathrm{w}(1)\ \mathrm{w}(2)$	*not unique*

Thus any enumeration F *including the executions above excludes every possible valuation of* $\langle x_\mathsf{w}, y_\mathsf{w} \rangle$*, resulting in an unsatisfiable* NORMALIZE$_F$.

Satisfiability of NORMALIZES$_F$ does not necessarily lead to a unique matching scheme, since NORMALIZE$_F$ can have multiple satisfying assignments. Furthermore, NORMALIZES$_F$ does not necessarily normalize $\mathcal{I}$. For one reason, F may not be a complete set of examples of executions and invalid projections. Second, our simple characterization of NORMALIZE$_F$ does not rule out the other reasons for a given example $\langle e, e' \rangle \in F$ to be an invalid projection, e.g., that the removed operations do not constitute *multiple* matches. However, we believe that checking satisfiability of NORMALIZE$_F$ is useful nonetheless: at the very least, satisfying assignments can be used as *assistance* in constructing normalizing matching schemes.

9. Naturally-Occurring ADTs

In this section we demonstrate that the premises used in the development of our symbolic ADT inference algorithm — i.e., uniqueness and bounded-width, faithful and normalizing matching schemes, and predictability — hold for the ADTs which are typically provided by concurrent object libraries. Furthermore, we demonstrate that the algorithm developed in this work computes precise symbolic representations for these ADTs which can be used in symbolic checkers for observational refinement (Emmi et al. 2015). Our publicly available implementation[8] enumerates the sequential histories[9] not admitted by reference implementations of the undermentioned ADTs, keeping only the histories which are not generated by any other. In each case, our algorithm terminates in a matter of seconds.

9.1 The Atomic Register

The atomic register implements an atomic single-value store, providing two methods:

- write(v) stores the value v, and
- read $\Rightarrow v$ returns the last-stored value v, or the nil value $-$ if no value has yet been stored.

As its name implies, the atomic register ADT is sequential.

We say an execution of read and write methods *writes unique values* if the argument value to each write operation is unique. A faithful matching scheme is given over the set of executions which write unique values as follows:

- write(v) operations match themselves, and
- read $\Rightarrow v$ operations are read-only, and match themselves if $v = -$, or the unique write(v) operation, if one exists, and otherwise have no match.

This matching scheme is faithful since two executions with the same history are identical up to homomorphic renaming of data values, and the register ADT only relates data values via equality.

[8] The URL has been suppressed for anonymity purposes.

[9] Our current implementation is limited to atomic ADTs. For non-atomic ADTs, the enumeration must cover all k-width histories, for some $k \in \mathbb{N}$.

It is easy to see that the atomic register is normal. It is closed under removal of read-only and duplicate operations, since each match can contain an arbitrary number of read operations. Since the histories of atomic registers do not contain unmatched operations, they are also closed under their removal. Finally, since the sequences contain an arbitrary number of matches, atomic register histories are also closed under match removal.

Our inference algorithm computes the following four histories to generate the complement of the atomic register ADT:

```
[1:X] read => 1 (RO)  #
---
[1:1] write(1)        #
[2:2] write(2)          #
[3:1] read => 1 (RO)      #
```

```
[1:1] write(1)        #
[2:2] read => - (RO)    #
---
[1:2] read => 1 (RO)  #
[2:2] write(1)          #
```

As new generators are discovered for $n = 1$ and $n = 2$ matches only, the atomic register ADT is predictable.

9.2 The Atomic Queue & The Atomic Stack

Atomic queues and stacks implement atomic collections of data values with first-in-first-out (FIFO) and last-in-first-out (LIFO) removal order, respectively, providing two methods:[10]

- add(v) adds the value v to the collection, and
- remove $\Rightarrow v$ returns the nil value $v = -$ if the collection is empty, and otherwise removes and returns the least- or most-recently added value v, respectively.

As their names imply, these ADTs are sequential.

An execution *adds unique values* if the argument value to each add operation is unique. We give a faithful matching scheme over executions which add unique values as follows:

- add(v) operations match themselves, and
- remove $\Rightarrow v$ operations are read-only, and match themselves if $v = -$ is the nil value. If $v \neq -$, remove $\Rightarrow v$ operations are not read-only, and match the unique add(v) operation, if one exists, and otherwise have no match.

This matching scheme is faithful since two executions with the same history are identical up to homomorphic renaming of data values, and the queue and stack ADTs only relate data values via equality.

Atomic queues and stacks are normal. They are closed under removal of read-only operations: only empty removes, i.e., remove $\Rightarrow -$, are read-only. They are closed under the removal of duplicate operations: only non-empty removes can be duplicates, and such duplicates are not admitted in the first place, since each value is added only once. Similarly, unmatched operations, i.e., removes that return values that have not been added, are not admitted in the first place. Finally, since the removal of entire matches preserves the FIFO/LIFO behavior of the entire collection, and the correctness of empty returns, histories are also closed under match removal.

We compute the following seven histories to generate the complement of the atomic queue ADT:

```
[1:X] remove => 1  #
---
[1:1] add(1)          #
[2:2] remove => - (RO)  #
---
[1:2] remove => 1  #
[2:2] add(1)        #
---
[1:1] add(1)      #
[2:2] add(2)        #
[3:2] remove => 2     #
```

```
[1:1] add(1)      #
[2:1] remove => 1   #
[3:1] remove => 1     #
---
[1:1] add(1)          #
[2:2] remove => - (RO)  #
[3:1] remove => 1         #
---
[1:1] add(1)      #
[2:2] add(2)        #
[3:2] remove => 2     #
[4:1] remove => 1       #
```

[10] These methods are also known as enqueue and dequeue, or push and pop.

The histories computed for the atomic stack ADT are nearly identical, substituting only the bottom two histories for the following:

```
[1:1] add(1)       #          [1:1] add(1)       #
[2:2] add(2)         #        [2:2] add(2)         #
[3:1] remove => 1      #      [3:1] remove => 1      #
                              [4:2] remove => 2        #
```

As new generators are discovered for $n = 1$ and $n = 2$ matches only, these ADTs are predictable.

9.3 The Atomic Set

Atomic sets implement collections which store one copy of each inserted data value no matter how many times the same value is inserted, until removed, providing three methods:

- insert$(u) \Rightarrow v$ inserts the value u to the collection,
- remove$(u) \Rightarrow v$ removes u from the collection, and
- contains$(u) \Rightarrow v$ checks whether the set contains u.

Each operation returns the nil value $v = -$ when u is not yet present, and otherwise returns $v = u$. The atomic set ADT is sequential.

An execution *inserts unique values* if the argument value to each insert operation which returns $-$ is unique. We give a faithful matching scheme over executions which insert unique values as follows:

- any operation returning $v = -$ matches itself, and
- any operation returning $v \neq -$ matches the unique insert$(v) \Rightarrow -$ operation, if one exists, and otherwise has no match.

This matching scheme is faithful since two executions with the same history are identical up to homomorphic renaming of data values, and sets only relates data values via equality.

We compute the following nine histories to generate the complement of the atomic set ADT:

```
[1:X] insert(1) => 1 (RO)  #       [1:1] insert(1) => -  #
---                                [2:1] remove(1) => 1   #
[1:X] remove(1) => 1  #            [3:1] remove(1) => 1    #
---                                ---
[1:X] contains(1) => 1 (RO) #      [1:1] insert(1) => -        #
---                                [2:1] remove(1) => 1         #
[1:1] insert(1) => -       #       [3:1] contains(1) => 1 (RO)   #
[2:2] remove(1) => - (RO)   #      ---
---                                [1:1] insert(1) => -      #
[1:1] insert(1) => -         #     [2:2] remove(1) => - (RO)  #
[2:2] contains(1) => - (RO)   #    [3:1] remove(1) => 1        #
```

As new generators are discovered for $n = 1$ and $n = 2$ matches only, the atomic set ADT is predictable.

9.4 The Atomic Lock

Atomic locks implement resource-based mutual exclusion by providing two methods

- lock$(u) \Rightarrow v$ acquires the lock resource,
- unlock $\Rightarrow v$ releases the lock resource.

An operation returns the nil value $v = -$ when the lock is not currently held, indicating success for lock, and failure for unlock. Otherwise, the operations return the value $u = v$ passed as an argument to the last successful lock(u) operation.

An execution *uses unique keys* if the argument value to each lock operation is unique. We give a faithful matching scheme over executions using unique keys as follows:

- any operation returning $v = -$ matches itself, and
- any operation returning $v \neq -$ matches the unique lock$(v) \Rightarrow -$ operation, if one exists, and otherwise has no match.

This matching scheme is faithful, and normalizing.

We compute the following eleven histories to generate the complement of the atomic lock ADT:

```
[1:X] unlock => 1  #               [1:X] lock(1) => 1 (RO)  #
---                                ---
[1:1] lock(1) => -  #              [1:1] lock(1) => -  #
[2:2] lock(2) => -    #            [2:1] unlock => 1     #
---                                [3:1] unlock => 1       #
[1:1] lock(1) => -      #          ---
[2:2] unlock => - (RO)    #        [1:1] lock(1) => -      #
---                                [2:2] unlock => - (RO)    #
[1:1] lock(1) => -  #              [3:1] unlock => 1           #
[2:2] lock(2) => -    #            ---
[3:1] unlock => 1       #          [1:1] lock(1) => -  #
---                                [2:2] lock(2) => -    #
[1:1] lock(1) => -  #              [3:1] unlock => 1       #
[2:2] lock(2) => -    #            [4:2] unlock => 2         #
[3:2] unlock => 2       #          ---
---                                [1:1] lock(1) => -  #
[1:1] lock(1) => -       #         [2:2] lock(2) => -    #
[2:1] unlock => 1          #       [3:2] unlock => 2       #
[3:1] lock(2) => 1 (RO)      #     [4:1] unlock => 1         #
```

As new generators are discovered for $n = 1$ and $n = 2$ matches only, the atomic lock ADT is predictable.

9.5 Work-Stealing Queue

The work-stealing queue (Hendler et al. 2006) implements a collection of data values with first-in-first-out[11] (FIFO) removal order, proving three methods:

- give(v) adds the value v to the queue,
- take $\Rightarrow v$ removes the value v, and
- steal $\Rightarrow v$ removes the value v.

Unlike atomic queues, the work-stealing queue permits values to be removed twice: once normally, via the take operation, and once exceptionally, via the steal operation. A faithful matching scheme over executions which add unique values is analogous to the scheme for stacks and queues: give operations match themselves, while take and steal operations match the give operation which added their returned value, or themselves, in case the nil value is returned. Note that the work-stealing queue ADT has width 2, since a pairs of concurrent take and steal operations returning the same value are permitted, while sequentially they are not. As our implementation is currently limited to width-1 histories (see the discussion in Section 10), below we generate only the width-1 complement.

We compute the following twenty-four histories to generate the complement of the work-stealing queue ADT:

```
[1:_] take => 1  #                 [1:_] steal => 1  #
---                                ---
[1:1] give(1)          #           [1:1] give(1)           #
[2:2] take => - (RO)    #          [2:2] steal => - (RO)    #
---                                ---
[1:2] take => 1  #                 [1:2] steal => 1  #
[2:2] give(1)      #               [2:2] give(1)       #
---                                ---
[1:1] give(1)    #                 [1:1] give(1)     #
[2:2] give(2)      #               [2:2] give(2)       #
[3:2] take => 2      #             [3:2] steal => 2      #
---                                ---
[1:1] give(1)    #                 [1:1] give(1)     #
[2:1] take => 1    #               [2:1] take => 1     #
[3:1] take => 1      #             [3:1] steal => 1      #
---                                ---
[1:1] give(1)          #           [1:1] give(1)          #
[2:2] take => - (RO)    #          [2:2] take => - (RO)    #
[3:1] take => 1           #        [3:1] steal => 1          #
---                                ---
```

[11] There are actually four variations to the work-stealing queue, depending on the ends from which the take and steal operations remove values. While our approach works indifferently, below we focus on the variation where both remove the least recent.

```
[1:1] give(1)     #          [1:1] give(1)     #
[2:1] steal => 1    #        [2:1] steal => 1    #
[3:1] take => 1       #      [3:1] steal => 1      #
---                          ---
[1:1] give(1)          #     [1:1] give(1)          #
[2:2] steal => - (RO)    #   [2:2] steal => - (RO)    #
[3:1] take => 1            # [3:1] steal => 1          #
---                          ---
[1:2] take => 1   #          [1:3] take => 1   #
[2:2] give(1)       #        [2:3] steal => 1    #
[3:2] steal => 1      #      [3:3] give(1)         #
---                          ---
[1:2] steal => 1  #          [1:3] steal => 1  #
[2:2] give(1)       #        [2:3] take => 1     #
[3:2] take => 1       #      [3:3] give(1)         #
---                          ---
[1:1] give(1)    #           [1:1] give(1)     #
[2:2] give(2)      #         [2:2] give(2)       #
[3:2] take => 2      #       [3:2] take => 2       #
[4:1] take => 1        #     [4:1] steal => 1        #
---                          ---
[1:1] give(1)     #          [1:1] give(1)     #
[2:2] give(2)       #        [2:2] give(2)       #
[3:2] steal => 2      #      [3:2] steal => 2      #
[4:1] take => 1         #    [4:1] steal => 1        #
```

As new generators are discovered for $n = 1$ and $n = 2$ matches only, the work-stealing queue ADT is predictable.

10. Discussion

While the theoretical foundation proposed in this work covers the majority of naturally-occurring ADTs, it is worth mentioning a few conceptual limitations and possible solutions to overcome them.

First, while our theoretical foundation does cover "concurrency-aware" ADT specifications (Hemed and Rinetzky 2014), i.e., non-atomic ADTs like the *rendezvous synchronizer*, *barrier*, and *exchanger*, which have bounded width greater than 1, our current implementation only handles atomic ADTs. In principle, this limitation is not fundamental. The key difference is in the enumeration ADT histories. For a given ADT, determining whether a given history is admitted or not reduces to checking whether there exists an execution of which the history is an abstraction. For atomic ADTs, only a single sequential execution need be examined, in which no two operations overlap. For non-atomic ADTs, every possible interleaving of the internal actions of operations need be examined. While we expect this calculation to remain feasible given that ADT complements are normally represented with histories with few operations, we do expect it to incur a noticeable cost.

Second, ADTs whose specifications rely on relations (besides equality) on method argument and return values cannot be expressed with our notion of histories based on matching schemes. A notable example of such an ADT is the *priority queue*, whose *dequeue* operations return the smallest/largest *enqueue*d value which has not yet been dequeued. Overcoming this limitation would require a refinement to matching schemes which can track additional relations among data values.

Finally, in some cases it is unclear how to express matching schemes deterministically. For example, each *wait* operation of *semaphore* ADT should naturally match the *notify* operation which enabled it. However, it is not possible to determine this based on operation labels alone, and it is not clear whether all implementations effectively keep track of this correspondence. Furthermore, choosing matches arbitrarily would be unsound, and result in inferring ADTs with artificial constraints, e.g., that the notify and wait operations behave according to a FIFO discipline, each wait matching the oldest unmatched notify. Overcoming this limitation may require introducing a notion of nondeterminism into matching schemes.

11. Related Work

Inference of symbolic abstract data types is a recently-pertinent problem arising from the the emergence of efficient symbolic refinement-checking algorithms (Bouajjani et al. 2015a; Emmi et al. 2015) which require symbolic, logical representations of ADT specifications. Though approaches based on the explicit enumeration of execution linearizations (Wing and Gong 1993; Burckhardt et al. 2010; Burnim et al. 2011; Zhang et al. 2013) do not require symbolic ADT representations, and work directly with given reference implementations, these approaches are intractable as they elaborate an exponential[12] number of linearizations.

Other approaches to tractable refinement-checking are based on annotating method bodies with *linearization points* (Herlihy and Wing 1990; Amit et al. 2007; Liu et al. 2009; Vafeiadis 2010; O'Hearn et al. 2010; Zhang 2011; Shacham et al. 2011; Dragoi et al. 2013; Liang and Feng 2013) to reduce the otherwise exponential number of possible linearizations to one single linearization. This approach, however, does not apply to all implementations (Herlihy and Wing 1990), and applications of this approach often rely on manual annotation of the implementation source code. Furthermore, this approach does not admit conclusive evidence of a refinement violation in case of failure.

The idea of inferring minimal finite representations of otherwise-infinite sets of implementation behaviors has also been proposed as an optimization to reduce the impact of state-space explosion in compositional verification (Giannakopoulou et al. 2005). This approach is based on learning minimal automata for regular languages, and requires the implementation of both language membership and equivalence queries (Angluin 1987). In our setting, it is unclear whether ADT implementations may admit regular characterizations in general, and furthermore how to automatically discharge language equivalence queries between candidate automata and the source code of reference implementations.

The idea of decomposing ADT violations into a finite set of patterns has been proposed for atomic collections (Abdulla et al. 2013; Henzinger et al. 2013; Dodds et al. 2015; Bouajjani et al. 2015a), and generalized to ADTs which can be expressed with a particular form of recursive definition (Bouajjani et al. 2015b). These works suggest patterns which directly recognize violations in *concurrent* executions, effectively reducing observational refinement to the verification of classical temporal-logic properties. Some of these patterns are expressible only as *infinite*, though regular, languages. On the contrary, in this work we suggest patterns expressed as *finite* languages, which recognize violations in ADTs — i.e., the sequential executions of atomic objects, or the "concurrency-aware" executions of "concurrency-aware" specifications (Hemed and Rinetzky 2014). Though by themselves these patterns are incomplete in recognizing violations in concurrent executions, they are complete when considered as negative conditions on the linearizations of concurrent executions, effectively reducing verification of a single execution to satisfiability checking (Emmi et al. 2015).

A. Proof that $\preceq$ is a Well-Quasi-Order (Lemma 4)

We first prove that the embedding order, defined hereafter, is a wqo on width-bounded labeled interval orders. Let Σ be a finite alphabet. A labeled interval order is a triple $(A, \leq, \ell)$, where $(A, \leq)$ is an interval order, and $\ell : A \to \Sigma$ is a labeling function. The embedding order $\subseteq$ between labeled interval orders is defined as usual by:

- $(A_1, \leq_1, \ell_1) \subseteq (A_2, \leq_2, \ell_2)$ iff there exists an injective function $h : A_1 \to A_2$ that preserves the labeling, i.e., $\ell_1(x) = \ell_2(h(x))$, for every $x \in A_1$, and the order constraints, i.e., for every $x, y \in A_1$, if $x \leq_1 y$, then $h(x) \leq_2 h(y)$.

[12] The number of possible linarizations is exponential in execution length.

The width of an interval order $(A, \leq, \ell)$ is the maximum number of elements which are mutually unordered.

Lemma 9. *$\subseteq$ is a wqo on width-bounded interval orders.*

Proof. By definition, for every interval order $(A, \leq, \ell)$, there exists a mapping I from elements of A to intervals on $\mathbb{N}$ such that for every $x, y \in A$, if $x \leq y$, then the interval $I(x)$ ends before the interval $I(y)$ (i.e., the upper bound of $I(x)$ is strictly smaller than the lower bound of $I(y)$). Therefore, in the following, we assume that an interval order is a multiset Γ of triples of the form $[i, j, a]$ where $[i, j]$ is an interval on $\mathbb{N}$ and a is a symbol in Σ.

We prove that another order $\subseteqq$, stronger than the embedding order $\subseteq$, is a wqo. The order $\subseteqq$ is defined by: $\Gamma_1 \subseteqq \Gamma_2$ iff there exists an injective function $h : \Gamma_1 \to \Gamma_2$ such that

1. h preserves the labeling, i.e., for any triple $[i, j, a]$, $h([i, j, a]) = [i', j', a]$, for some i' and j',
2. h preserves the ordering constraints, i.e., for every two triples $[i, j, a]$ and $[k, l, b]$ such that $j < k$, $h([i, j, a]) = [i', j', a]$, $h([k, l, b]) = [k', l', b]$, and $j' < k'$.
3. for every two incomparable elements x and y of Γ_1, if the interval of x starts before the interval of y, then the interval of $h(x)$ also starts before the interval of $h(y)$. Formally, for every two triples $[i, j, a]$ and $[k, l, b]$ such that $i \leq k$ (i.e., the first interval starts before the second interval), $h([i, j, a]) = [i', j', a]$, $h([k, l, b]) = [k', l', b]$, and $i' \leq k'$.

We say that h *witnesses* $\Gamma_1 \subseteqq \Gamma_2$.

Assume $\Gamma_1, \Gamma_2, \ldots$ is a *bad* sequence, i.e., an infinite sequence of interval orders s.t. there exists no $i < j$ with $\Gamma_i \subseteqq \Gamma_j$. Also, assume that Γ_1 is the minimal size interval order that can start a bad sequence, Γ_2 is the minimal order that can continue a bad sequence starting with Γ_1, and so on.

For each Γ_k, let $Min(\Gamma_k)$ be a triple $[i, j, a]$ such that (1) i is the minimal lower bound of an interval in Γ_k, and (2) j is the minimal upper bound of an interval in Γ_k with lower bound i. Also, let $Init(\Gamma_k) = (a_k, P_k)$, where $Min(\Gamma_k) = [i, j, a_k]$, for some i and j, and P_k is the multiset of symbols labeling elements of Γ_k that are incomparable to $Min(\Gamma_k)$. Note that P_k is bounded since we assume width-bounded interval orders.

The infinite sequence $\Gamma_1, \Gamma_2, \ldots$ contains an infinite sequence $\Gamma_{k_1}, \Gamma_{k_2}, \ldots$ which have the same value for $Init$. For each Γ_k, let Λ_k be the interval order obtained from Γ_k by removing the triple $Min(\Gamma_k)$. By the minimality assumptions, the infinite sequence $\Gamma_1, \Gamma_2, \ldots, \Gamma_{k_1-1}, \Lambda_{k_1}, \Lambda_{k_2}, \ldots$ is not bad. Therefore, there exists $m < n$ such that $\Lambda_{k_m} \subseteqq \Lambda_{k_n}$.

It remains to prove that $\Gamma_{k_m} \subseteqq \Gamma_{k_n}$ when $Init(\Gamma_{k_m}) = Init(\Gamma_{k_n})$ and $\Lambda_{k_m} \subseteqq \Lambda_{k_n}$. Let h be the injective function witnessing $\Lambda_{k_m} \subseteqq \Lambda_{k_n}$. We prove that the extension h' of h between Γ_{k_m} and Γ_{k_n}, defined by $h'(Min(\Gamma_{k_m})) = Min(\Gamma_{k_n})$ and $h'(t) = h(t)$, for all $t \in \Lambda_{k_m}$, witnesses $\Gamma_{k_m} \subseteqq \Gamma_{k_n}$.

Clearly, h' preserves the labeling. To prove that h' preserves the ordering constraints, let $Min(\Gamma_{k_m}) = [i, j, a]$ and $[k, l, b] \in \Gamma_{k_m}$ such that $j < k$. Also, let $Min(\Gamma_{k_n}) = [i', j', a]$ and $h'([k, l, b]) = h([k, l, b]) = [k', l', b]$. Assume by contradiction that $k' \leq j'$. Since h is injective, there exists an element $[e, f, c] \in \Gamma_{k_m}$ incomparable to $Min(\Gamma_{k_m})$ which is mapped to an element $[e', f', c]$ greater than $Min(\Gamma_{k_n})$ (because, by definition, Γ_{k_m} and Γ_{k_n} have the same number of elements incomparable to $Min(\Gamma_{k_m})$ and respectively, $Min(\Gamma_{k_n})$). Since $[e, f, c]$ is incomparable to $[i, j, a]$, we obtain that $e \leq j < k$. Also, by the current assumptions, $k' \leq j' < e'$. Therefore, there exist two elements $[e, f, c]$ and $[k, l, b]$ such that the interval $[e, f]$ starts before $[k, l]$ which are mapped by h to the elements $[e', f', c]$ and $[k', l', b]$ such that the interval $[e', f']$ starts after $[k', l']$. This contradicts the fact that h witnesses $\Lambda_{k_m} \subseteqq \Lambda_{k_n}$.

The properties of $Min(\Gamma_{k_m})$ and $Min(\Gamma_{k_n})$ imply that h' satisfies also the third property in the definition of $\subseteqq$.

Finally, since $\subseteqq$ is stronger than $\subseteq$, we get that $\subseteq$ is a wqo on width-bounded labeled interval orders. □

Lemma 10. *$\preceq$ is a wqo on width-bounded histories.*

Proof. Let $h_1 h_2 \ldots$ be an infinite sequence of histories. We prove that there exists $i < j$ such that $h_i \preceq h_j$.

The *signature* of an operation o in a history h is the pair $\sigma(o) = (c(o), f(o))$. Then, the *signature* of a match $\mu = \{o' \in O : m(o') = o\}$ is the pair

$$\sigma(\mu) = (\sigma(o), \{\sigma(o') : m(o') = o, o' \neq o\})$$

containing the signature of the match target and the set of the signatures of the other operations. The *signature* of a history h is the tuple

$$\begin{aligned}\sigma(h) = \quad & (\{\sigma(\mu) : \mu \text{ a match}\}, \{\sigma(o) : o \text{ read-only}\}, \\ & \{\sigma(o) : o \text{ unmatched}\}, \\ & \{\{\sigma(o) : o \text{ pending and non-matched}\}\}).\end{aligned}$$

The history signature contains sets of signatures (for matches, read-only, completed, and unmatched operations) and the multiset of signatures of the pending and non-matched operations. Since the set of history signatures is bounded (because the set of methods and the set of pending operations are bounded), the sequence $h_1 h_2 \ldots$ contains an infinite sequence of histories $h'_1 h'_2 \ldots$ that have the same signature.

The *vector* of a history h is the tuple

$$\begin{aligned}\nu(h) = \quad & (\{\{\sigma(\mu) : \mu \text{ a match of } h\}\}, \{\{\sigma(o) : o \text{ read-only}\}\}, \\ & \{\{\sigma(o) : o \text{ unmatched}\}\}).\end{aligned}$$

Let $\leq$ be an order relation on multisets $\alpha : \Sigma \to \mathbb{N}$ of (match or operation) signatures from a finite set Σ defined by $\alpha \leq \alpha'$ iff $\alpha(\sigma) \leq \alpha'(\sigma)$, for every $\sigma \in \Sigma$. The relation $\leq$ is a wqo and so is the component-wise extension of $\leq$ to history vectors. Therefore, by known results, the sequence of histories $h'_1 h'_2 \ldots$ contains an infinite sequence $h''_1, h''_2, \ldots$ such that $\nu(h''_1) \leq \nu(h''_2) \leq \ldots$.

The *vector* of a match μ is the pair

$$\nu(\mu) = (\sigma(o), \{\{\sigma(o') : m(o') = o, o' \neq o\}\})$$

containing the signature of the match target and the multiset of the signatures of the other operations.

For every history h and match signature σ, the *σ-vector* of a history h is the multiset $\nu_\sigma(h) = \{\{\nu(\mu) : \sigma(\mu) = \sigma\}\}$ of match vectors of signature σ.

Let $\leq_v$ be an order relation on σ-vectors defined by $\nu_\sigma(h) \leq_v \nu_\sigma(h')$ iff for every match vector ν in $\nu_\sigma(h)$ there is a match vector ν' in $\nu_\sigma(h')$ s.t. $\nu \leq \nu'$ (here, $\leq$ is the order on multisets defined above). By known results, $\leq_v$ is a wqo.

Let $\sigma_1, \sigma_2, \ldots, \sigma_n$ be the match signatures defined over a set of methods $\mathbb{M}$. Since $\leq_v$ is a wqo, the sequence of histories $h''_1, h''_2, \ldots$ contains an infinite sequence $h^1_1, h^1_2, \ldots$ such that $\nu_{\sigma_1}(h^1_1) \leq_v \nu_{\sigma_1}(h^1_2) \leq_v \ldots$. Then, the sequence $h^1_1, h^1_2, \ldots$ contains an infinite sequence $h^2_1, h^2_2, \ldots$ such that $\nu_{\sigma_2}(h^2_1) \leq_v \nu_{\sigma_2}(h^2_2) \leq_v \ldots$. Applying a similar reasoning for the rest of the match signatures, we obtain that $h''_1, h''_2, \ldots$ contains an infinite sequence $h^n_1, h^n_2, \ldots$ such that $\nu_\sigma(h^n_1) \leq_v \nu_\sigma(h^n_2) \leq_v \ldots$, for every signature σ. Recall that we also have that $\nu(h^n_1) \leq \nu(h^n_2) \leq \ldots$ since $h^n_1, h^n_2, \ldots$ is a sub-sequence of $h''_1, h''_2, \ldots$.

Viewing histories as labeled interval orders, where the label of an element o is the triple $(c(o), f(o), r(o))$, we get that the embedding order $\subseteq$ is a wqo on histories. Therefore, the infinite sequence $h^n_1, h^n_2, \ldots$ contains two elements h^n_i and h^n_j with $i < j$ such that $h^n_i \subseteq h^n_j$. This implies that $h^n_i \preceq h^n_j$, which ends our proof. □

References

P. A. Abdulla, K. Cerans, B. Jonsson, and Y.-K. Tsay. General decidability theorems for infinite-state systems. In *Proc. 11th IEEE Symposium on Logic in Computer Science (LICS '96)*, pages 313–321. IEEE Computer Society, 1996.

P. A. Abdulla, F. Haziza, L. Holík, B. Jonsson, and A. Rezine. An integrated specification and verification technique for highly concurrent data structures. In *Proc. 19th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS '13)*, volume 7795 of *LNCS*, pages 324–338. Springer, 2013. URL `http://dx.doi.org/10.1007/978-3-642-36742-7_23`.

D. Amit, N. Rinetzky, T. W. Reps, M. Sagiv, and E. Yahav. Comparison under abstraction for verifying linearizability. In *Proc. 19th International Conference on Computer Aided Verification, (CAV '07)*, volume 4590 of *LNCS*, pages 477–490. Springer, 2007. URL `http://dx.doi.org/10.1007/978-3-540-73368-3_49`.

D. Angluin. Learning regular sets from queries and counterexamples. *Inf. Comput.*, 75(2):87–106, 1987. URL `http://dx.doi.org/10.1016/0890-5401(87)90052-6`.

A. Bouajjani, M. Emmi, C. Enea, and J. Hamza. Tractable refinement checking for concurrent objects. In *Proc. 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, (POPL '15)*, pages 651–662. ACM, 2015a. URL `http://doi.acm.org/10.1145/2676726.2677002`.

A. Bouajjani, M. Emmi, C. Enea, and J. Hamza. On reducing linearizability to state reachability. *arXiv:1502.06882*, 2015b. URL `http://arxiv.org/abs/1502.06882`.

S. Burckhardt, C. Dern, M. Musuvathi, and R. Tan. Line-up: a complete and automatic linearizability checker. In *Proc. 2010 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '10)*, pages 330–340. ACM, 2010. URL `http://doi.acm.org/10.1145/1806596.1806634`.

J. Burnim, G. C. Necula, and K. Sen. Specifying and checking semantic atomicity for multithreaded programs. In *Proc. 16th International Conference on Architectural Support for Programming Languages and Operating Systems, (ASPLOS '11)*, pages 79–90. ACM, 2011. URL `http://doi.acm.org/10.1145/1950365.1950377`.

M. Dodds, A. Haas, and C. M. Kirsch. A scalable, correct time-stamped stack. In *Proc. 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, (POPL '15)*, pages 233–246. ACM, 2015. URL `http://doi.acm.org/10.1145/2676726.2676963`.

C. Dragoi, A. Gupta, and T. A. Henzinger. Automatic linearizability proofs of concurrent objects with cooperating updates. In *Proc. 25th International Conference on Computer Aided Verification (CAV '13)*, volume 8044 of *LNCS*, pages 174–190. Springer, 2013. URL `http://dx.doi.org/10.1007/978-3-642-39799-8_11`.

M. Emmi, C. Enea, and J. Hamza. Monitoring refinement via symbolic reasoning. In *Proc. 36th Annual ACM SIGPLAN Conference on Programming Language Design and Implementation, (PLDI '15)*. ACM, 2015.

A. Finkel and P. Schnoebelen. Well-structured transition systems everywhere! *Theor. Comput. Sci.*, 256(1-2):63–92, 2001.

P. C. Fishburn. *Interval Orders and Interval Graphs: A Study of Partially Ordered Sets*. Wiley-Interscience Series in Discrete Mathematics. John Wiley & Sons Inc, 1985.

D. Giannakopoulou, C. S. Pasareanu, and H. Barringer. Component verification with automatically generated assumptions. *Autom. Softw. Eng.*, 12(3):297–320, 2005. URL `http://dx.doi.org/10.1007/s10515-005-2641-y`.

N. Hemed and N. Rinetzky. Brief announcement: concurrency-aware linearizability. In *ACM Symposium on Principles of Distributed Computing (PODC '14)*, pages 209–211. ACM, 2014. URL `http://doi.acm.org/10.1145/2611462.2611513`.

D. Hendler, Y. Lev, M. Moir, and N. Shavit. A dynamic-sized nonblocking work stealing deque. *Distributed Computing*, 18(3):189–207, 2006. URL `http://dx.doi.org/10.1007/s00446-005-0144-5`.

T. A. Henzinger, A. Sezgin, and V. Vafeiadis. Aspect-oriented linearizability proofs. In *Proc. 24th International Conference on Concurrency Theory (CONCUR '13)*, volume 8052 of *LNCS*, pages 242–256. Springer, 2013.

M. Herlihy and J. M. Wing. Linearizability: A correctness condition for concurrent objects. *ACM Trans. Program. Lang. Syst.*, 12(3):463–492, 1990.

H. Liang and X. Feng. Modular verification of linearizability with non-fixed linearization points. In *Proc. ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '13)*, pages 459–470. ACM, 2013. URL `http://doi.acm.org/10.1145/2462156.2462189`.

Y. Liu, W. Chen, Y. A. Liu, and J. Sun. Model checking linearizability via refinement. In *Proc. Second World Congress on Formal Methods (FM '09)*, volume 5850 of *LNCS*, pages 321–337. Springer, 2009. URL `http://dx.doi.org/10.1007/978-3-642-05089-3_21`.

P. W. O'Hearn, N. Rinetzky, M. T. Vechev, E. Yahav, and G. Yorsh. Verifying linearizability with hindsight. In *Proc. 29th Annual ACM Symposium on Principles of Distributed Computing (PODC '10)*, pages 85–94. ACM, 2010. URL `http://doi.acm.org/10.1145/1835698.1835722`.

O. Shacham, N. G. Bronson, A. Aiken, M. Sagiv, M. T. Vechev, and E. Yahav. Testing atomicity of composed concurrent operations. In *Proc. 26th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA '11)*, pages 51–64. ACM, 2011. URL `http://doi.acm.org/10.1145/2048066.2048073`.

V. Vafeiadis. Automatically proving linearizability. In *Proc. 22nd International Conference on Computer Aided Verification (CAV '10)*, volume 6174 of *LNCS*, pages 450–464. Springer, 2010. URL `http://dx.doi.org/10.1007/978-3-642-14295-6_40`.

J. M. Wing and C. Gong. Testing and verifying concurrent objects. *J. Parallel Distrib. Comput.*, 17(1-2):164–182, 1993. URL `http://dx.doi.org/10.1006/jpdc.1993.1015`.

L. Zhang, A. Chattopadhyay, and C. Wang. Round-up: Runtime checking quasi linearizability of concurrent data structures. In *2013 28th IEEE/ACM International Conference on Automated Software Engineering (ASE '13)*, pages 4–14. IEEE, 2013. URL `http://dx.doi.org/10.1109/ASE.2013.6693061`.

S. J. Zhang. Scalable automatic linearizability checking. In *Proc. 33rd International Conference on Software Engineering (ICSE '11)*, pages 1185–1187. ACM, 2011. URL `http://doi.acm.org/10.1145/1985793.1986037`.