Symbolic Abstract Data Type Inference

Michael Emmi
IMDEA Software Institute, Spain
michael.emmi@imdea.org

Constantin Enea
Université Paris Diderot, France
cenea@liafa.univ-paris-diderot.fr

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 z 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

\[\text{q.add(<1,w>); q.add(<1,x>) and q.add(<2,w>); q.add(<2,z>)\}

Since the queue ADT dictates that elements are removed in the order added, we conclude that w and z 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 possess 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.

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 z 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

\[\text{q.add(<1,w>); q.add(<1,x>) and q.add(<2,w>); q.add(<2,z>)\}

Since the queue ADT dictates that elements are removed in the order added, we conclude that w and z 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 possess 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 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.

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

- An abstract notion of execution histories based on groups of matching operations (§4).
- The statement of the symbolic ADT inference problem (§5).
- Identification of the algebraic properties allowing a finite characterization of infinite ADT violation sets (§5).
- The symbolic representation of ADT violations (§5).
- The computability of symbolic ADT representations (§7).
- An algorithm to infer the matching schemes required for our algebraic characterization of ADTs (§5).
- 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 by discussing limitations (§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 of length at most 4 of an object with add and remove methods for which 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

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.
While many ADTs are “atomic” (e.g., queues, locks) in the sense that their ADTs are specified as sets of sequential histories, many others capture the possible histories of call and return actions in the execution of client programs. We thus consider invocation histories rather than concurrent invocation histories, and the “X” symbol denotes an absent match.

invalid linearizations. Nevertheless, specifications inferred by this algorithm are sound, in the sense that when modular program reasoning is performed, the implementation’s implicit ADT.

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

c1: 1: call write(a) 1: call write(a) 1: call write(a)
2: return 2: return 2: return
3: call read 3: call read 3: call read
4: return 4: return 4: return

Operation identifiers precede actions. Executions e2 and e3 are obtained from e1, 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 1 are sequential, each following the previous in time, those of 2 and 3 overlap. While 1 should not be admitted by an atomic register, since the read of Operation 3 does not return the most-recently-written value, both 2 and 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 $h = (O, <, c, f, m, r)$ where

- $O \subseteq \mathbb{O}$ is a set of operations,
- $<$ is a happens-before interval order\footnote{An interval order\cite{fishburn85} 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 $y < x$.} on $O$,
- $c : O \rightarrow \mathbb{B}$ labels operations as completed, or not,
- $f : O \rightarrow \mathbb{M}$ labels operations with method names,
- $m : O \rightarrow O$ is a partial matching function, and
- $r : O \rightarrow \mathbb{B}$ labels operations as read-only, or not.

The relation $<$ is an interval order since call and return actions are totally ordered\cite{bouajjani15b}. Non-completed operations are pending, indicated by the $r$. Histories do not keep them. In the following history, we draw histories by writing one operation per line, Example 3.

We assume that operation identifiers have no intrinsic meaning, we consider equality of an operation $o$ is unmatched, indicated by the $r$. Histories abstract executions, retaining method names yet losing

$\langle 3:X \rangle \text{read} \Rightarrow b \ (\text{RO})$
$\langle 2:1 \rangle \text{read} \Rightarrow a \ (\text{RO})$
$\langle 1:1 \rangle \text{write}(a)$

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 $v = 0$ (empty), 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 $(M, R)$ is the tuple $(O, <, c, f, M(e), R(e))$ where

- $O$ are the operations of $e$,
- $o_1 < o_2$ if operation $o_2$ returns before $o_2$ is called in $e$,
- $c(o)$ if 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 by $H(E, M, R)$. When the matching scheme $(M, R)$ 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 $H(e_1)$:

$\langle 1:1 \rangle \text{write}(a)$ #
$\langle 2:1 \rangle \text{read} \Rightarrow a \ (\text{RO})$ #

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

$\langle 1:1 \rangle \text{write}(a)$ #
$\langle 2:2 \rangle \text{write}(b)$ #
$\langle 3:1 \rangle \text{read} \Rightarrow a \ (\text{RO})$ #

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

$\langle 1:1 \rangle \text{write}(a)$ #
$\langle 2:2 \rangle \text{write}(b)$ #
$\langle 3:1 \rangle \text{read} \Rightarrow a \ (\text{RO})$ #

in which the last two operations overlap.

A matching scheme $(M, R)$ 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\footnote{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.} 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 $(M, R)$.

In Section 2, 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 $(M, R)$.

Two histories $h_1$ and $h_2$ are related by $\rightarrow_{x}$, for $x = o, c, p$, 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 \( \rightsquigarrow \) when \( h_2 \in H \)
whenever \( h_1 \rightarrow h_2 \) 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. 2015b).

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

\[
\begin{align*}
[1:1] & \text{write(a)} & \# \\
[2:2] & \text{write(b)} & \# \\
[3:2] & \text{read} \leftrightarrow b \text{ (RO)} & \#
\end{align*}
\]

we derive the \( \rightarrow_{c}, \text{related history} \)

\[
\begin{align*}
[1:1] & \text{write(a)} & \# \\
[2:2] & \text{write(b)} & \# \\
[3:2] & \text{read} \leftrightarrow b \text{ (RO)} & \#
\end{align*}
\]

from which we can derive the \( \rightarrow_{p}, \text{related history} \)

\[
\begin{align*}
[1:1] & \text{write(a)} & \# \\
[2:2] & \text{write(b)} & \# \\
[3:3] & \text{read} \leftrightarrow \# \text{ (RO)} & \#
\end{align*}
\]

by making Operation 3 pending, and from which we can derive the
\( \rightarrow_{u}, \text{related history} \)

\[
\begin{align*}
[1:1] & \text{write(a)} & \# \\
[2:2] & \text{write(b)} & \# \\
[3:4] & \text{write(c}\#) & \#
\end{align*}
\]

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 \( \rightarrow_{o}, \rightarrow_{p}, \rightarrow_{c} \).

### 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 \( K \) \textit{generates} \( H \) when the closure \( K^{*} \) of \( K \) under the
relation \( \rightarrow \) 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
\textit{generating} \( H \). While the kernels of arbitrary sets need not be
unique, the kernels of sets which have sequential kernels are unique.
Furthermore, Section 5 demonstrates that the histories of naturally-occuring
implementations have unique kernels, which we assume for the remainder of this work. An \textit{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 \textit{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. 2015b).

**Definition 1.** An implementation \( \mathcal{I}_{1} \) \textit{refines} another implementation \( \mathcal{I}_{2} \) when \( H(\mathcal{I}_{1}) \subseteq H(\mathcal{I}_{2}) \). An implementation \( \mathcal{I} \) \textit{refines} an ADT \( A \) when \( H(\mathcal{I}) \subseteq H(\mathcal{I}_{A}) \).

Recent works demonstrate efficient refinement-checking algorithms (Bouajjani et al. 2015b, Emmi et al. 2015) yet rely on handwritten symbolic ADT representations. In order to frame the problem of computing these automatically, we fix a language for symbolic representation. A \textit{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 \( c, m, u, r, \) and \( < \) for completion, non-matching
(operations which are not in the domain of the matching function), read-only, and order.

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:

\[
\forall x_{1}, x_{2}, x_{3}. (c(x_{1}) \land f(x_{1}) = \text{write} \land m(x_{1}) = x_{1} \\
\land c(x_{2}) \land f(x_{2}) = \text{write} \land m(x_{2}) = x_{2} \\
\land c(x_{3}) \land f(x_{3}) = \text{read} \land m(x_{3}) = x_{1} \\
\land x_{1} < x_{2} \Rightarrow x_{1} < x_{3}
\]

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 \( H(e_{1}) \) from Example 5 yet not \( H(e_{1}) \).

The \textit{bounded complement} of a history set \( H \) of width \( k \in \mathbb{N} \cup \{0\} \) is the set of histories of width at most \( k \) which are excluded from \( H \). Let \( A \) be an ADT and \( \mathcal{B} \) its bounded complement. We say that a history formula \( F \) \textit{represents} \( A \) when

• \( h \models F \) for all \( h \in A \), and
• \( h \not\models F \) for all \( h \in \mathcal{B} \).

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 result 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 \( \rightarrow_{o}, \rightarrow_{p}, \) and \( \rightarrow_{c} \) 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
\( \rightarrow_{r}, \rightarrow_{m}, \rightarrow_{u}, \) and \( \rightarrow_{d} \) 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 \( \rightarrow \), \( \rightarrow_u \), \( \rightarrow_m \) and \( \rightarrow_d \) is normal. In Section 5 we demonstrate that naturally-occurring ADTs are normal. Defining the relation \( \succeq \) as the reflexive and transitive closure of the above relations,
\[
\succeq = (\rightarrow_0 \cup \rightarrow_p \cup \rightarrow_u \cup \rightarrow_d \cup \rightarrow_m \cup \rightarrow_d)^* \]
closure under \( \succeq \) follows immediately.

**Lemma 3.** Normal ADTs are closed under \( \succeq \).

Besides this closure property, the inverse \( \preceq \) 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.

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 \( A \) is normal, it is closed under \( \succeq \), and thus \( B \) is closed 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 \( \subseteq Y \times X \) 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:

\[
\begin{align*}
&\text{[1:1] read } \Rightarrow 1 \text{ (RO)} \# \\
&\text{[1:1] write}(1) \# \\
&\text{[2:2] write}(2) \# \\
&\text{[3:1] read } \Rightarrow 1 \text{ (RO)} \#
\end{align*}
\]

Every sequential history not admitted by the atomic register ADT embeds (via \( \preceq \)) 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 those representations have logical interpretations, allowing us to derive formulas representing ADTs. In what follows we describe how to obtain formulas 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 \( h = (O, c, f, m, r) \) 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 \( o \in O \), the macro \( \text{EMBED}_o(x, y, z) \):

\[
\begin{align*}
&\{ \text{c}(x) \equiv \text{c}(y) \} \land f(x) = f(y) \land (r(x) \equiv r(y)) \\
&\land (o \in \text{dom}(m) \Rightarrow \text{um}(x) \land m(x) = z) \\
&\land (o \not\in \text{dom}(m) \Rightarrow \text{um}(x) \land \bigwedge y \in Y x < y)
\end{align*}
\]

captures 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 \( \text{IDENTICAL}(x, y) \):

\[
\{ c(x) \equiv c(y) \} \land f(x) = f(y) \land (r(x) \equiv r(y)) \\
\land (\text{um}(x) \equiv \text{um}(y)) \land m(x) = m(y)
\]
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 \( \text{MATCH}(Y, z) \):

\[
\forall x. (\neg \text{um}(x) \land m(x) = z) \Rightarrow r(x) \lor \bigvee y \in Y \text{IDENTICAL}(x, y)
\]

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 \( \text{EMBED} \):

\[
\exists x_1, \ldots, x_n \bigwedge_{i=1}^n \text{EMBED}_{o_i}(x_i, Y_i, z_i) \land \bigwedge_{i=1}^k \text{MATCH}(W_i, x_i)
\]

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 2 which generate the complement of the atomic register ADT. The \( \text{EMBED} \) formula for the first history:

\[
\begin{align*}
&\text{[1:1] read } \Rightarrow 1 \text{ (RO)} \# \\
&\text{[2:2] read } \Rightarrow - \text{ (RO)} 
\end{align*}
\]

after simplifications, like replacing \( \text{true } \Rightarrow p \) with \( p \), is

\[
\exists x_1. c(x_1) \land f(x_1) = \text{read} \land \text{um}(x_1) \land r(x_1)
\]

The \( \text{EMBED} \) formula for the second history:

\[
\begin{align*}
&\text{[1:1] write}(1) \# \\
&\text{[2:2] read } \Rightarrow - \text{ (RO)} \#
\end{align*}
\]
is similarly given by
\[ \exists x_1, x_2 : x_1 < x_2 \]
\[ \land c(x_1) \land f(x_1) = \text{write} \land \neg \text{um}(x_1) \land m(x_1) = x_1 \]
\[ \land c(x_2) \land f(x_2) = \text{read} \land \neg \text{um}(x_2) \land m(x_2) = x_2 \land r(x_2) \]
\[ \land (\forall x. \ m(x) = x_1 \Rightarrow r(x) \lor \text{IDENTICAL}(x, x_1)) \]
\[ \land (\forall x. \ m(x) = x_2 \Rightarrow r(x) \lor \text{IDENTICAL}(x, x_2)). \]

The formulas for the other histories are similarly obtained.

Lemma 6. $h_1 \models \text{EMBED}_{h_2}$ iff $h_1 \geq h_2$.

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 $\leq$, denoted by $H^*$ henceforth, is equal to $B$. The exclusion formula of a set of histories $H$ is $F(H) := \bigwedge_{h \in H} \neg \text{EMBED}_h$.

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. Let $\neg \Sigma$ denote the complement of a set $\Sigma$. By Lemma 3 the complement of $A$ is closed under $\preceq$. Since $H^* \subseteq B \subseteq \neg A$, we know $H^* \subseteq \neg A$. By Lemma 6 the formula $F(H)$ holds exactly for the histories $\neg H^*$. By the previous inclusions, we have that $A \subseteq H^* \subseteq \neg B$. Therefore, $F(H)$ holds for $A$ and not for $B$. \qed

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

7. A Symbolic ADT Inference Algorithm

Algorithm 2 solves symbolic ADT inference by computing a finite representation of an ADT complement $B$ using the $\preceq$ relation of Section 5. This is generally achieved by enumerating $B$ while maintaining the $\preceq$-minimal elements, and recognizing a condition under which all the elements of $B$ are related to the current set of minimal representatives (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, the weight of a history is the maximum among operation frequencies and the number of matches. We then define

\[ B_i = \{ h \in B : \text{weight}(h) \leq i \} \]
\[ B'_i = \{ h \in B : \exists h' \in B_i, h' \preceq h \land \text{weight}(h) \leq i + 1 \} \]

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 6.

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, each $B_i$ is computable. When $A$ is predictable, then $B = \text{patterns}^*$, and thus by Lemma 5 the returned formula $F(\text{patterns})$ represents $A$. \qed

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.

8. Matching Scheme Inference

Our solution to the ADT inference problem relies on identifying faithful matching schemes for ADT implementations. Though Section 5 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 executions. 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 $(M, R)$ 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 exposed, and a set of predicates required in the logic of matching schemes. Formally, a language $L = (E, P, Q)$ is a set $E$ of executions, along with finite sequences $P = P_1 P_2$ and $Q = Q_1 Q_2$ and binary and ternary predicates $P(e, o_1)$ and $Q(e, o_1, o_2)$ ranging over executions 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
\[ P_1(x_1, \ldots, x_n) \ldots P_n(x_1, \ldots, x_n). \]

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 $G(P(e, o_1), P(e, o_2), Q(e, o_1, o_2))$,

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:

- $w(e, o)$ is a write operation in $e$, and
- $v(e, o_1, o_2)$ is obtained by deleting operations of $e$.

We define the function $G(x_w, y_w, z_w)$ over valuations to the predicates above to be satisfied if and only if $x_w \land z_w$. 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 is the scheme specified in Example 2 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 $(e, e') \in E^2$ 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 $\text{IsMatch}_{e, o_1, o_2}$:

\[ g(P(e, o_1), P(e, o_2), Q(e, o_1, o_2)). \]

To enforce that each operation of an execution $e$ has a unique match, we define the macro $\text{UniqueMatch}_{e, o}:

\[ \bigwedge_{o_1, o_2 \in O_e} \text{IsMatch}_{e, o_1, o_2} \Rightarrow \bigwedge_{o_1 \neq o_2} \neg \text{IsMatch}_{e, o_1, o_2}. \]

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

\[ \bigwedge_{o_1 \in O} \left( \bigwedge_{o_2 \in O \setminus O_e} \text{IsMatch}_{e, o_1, o_2} \land \bigwedge_{o_2 \in O \setminus O_e} \neg \text{IsMatch}_{e, o_1, o_2} \right). \]

Finally, given a pair $(e, e') \in F$, 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 $\text{Normalize}_{e, O}$:

\[ \bigwedge_{(e, e') \in F} \left( \text{UniqueMatch}_{e, O_e} \land \neg \text{EntireMatch}_{e, (O_e \setminus O_{e'})} \right). \]

We thus check satisfiability for the conjunction of non-matches $\text{Normalize}_{e, O_e}$ over all pairs $(e, e') \in F$.

**Example 13.** Consider again the language $L_{reg}$ of read and write methods of Example 7 with predicates $w(e, o)$ and $v(e, o_1, o_2)$ whose executions write unique values. The following table lists all possible predicate valuations $(x_w, y_w, z_w)$, and for each valuation a valid execution which excludes the positive valuation of $g(x_w, y_w, z_w)$ in the satisfaction of $\text{Normalize}_{e, O_e}$, in the case such an execution exists.

<table>
<thead>
<tr>
<th>$x_w$</th>
<th>$y_w$</th>
<th>$z_w$</th>
<th>counterexample</th>
<th>reason</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 0 0</td>
<td>w(1) r(1) r(1)</td>
<td>not unique</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0 0 1</td>
<td>w(1) r(1) r(1)</td>
<td>not unique</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0 1 0</td>
<td>w(1) r(1) w(2) r(2)</td>
<td>$\neg \mathcal{M}$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0 1 1</td>
<td>w(1) r(1) r(1)</td>
<td>$\neg \mathcal{M}$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1 0 0</td>
<td>w(1) w(2) w(3) r(3)</td>
<td>not unique</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1 0 1</td>
<td>w(1) w(2) w(3)</td>
<td>not unique</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1 1 0</td>
<td>w(1) w(2) w(3)</td>
<td>not unique</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1 1 1</td>
<td>w(1) w(2) w(3)</td>
<td>not unique</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

For example, the first valuation 000 must be excluded since it allows the read r(2) to match both reads r(1), violating UniqueMatch. The second valuation 010 must also be excluded, since it allows either read r(1) to match both itself and the other read r(1). The third valuation 011 must be excluded since it allows the write w(2) to match the read r(1), in which case removing the match (r(1), w(2)) results in the invalid execution w(1) 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 $\text{Normalize}_{e, O_e}$ in which $g(x_w, y_w, z_w)$ is the same Boolean function $x_w \land z_w$ given in Example 7.

On the other hand, checking satisfiability of $\text{Normalize}_{e, O_e}$ can be used to conclude the impossibility of a good matching scheme — at least for the given language.

**Lemma 8.** If $\text{Normalize}_{e, O_e}$ 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 $L_{\text{seq}}$ of read and write methods of Example 7 yet this time without the predicate $\nu(e, o_1, o_2)$. The following table lists valid executions excluding each of the possible predicate valuations $(x_w, y_w)$ in the satisfaction of $\text{NORMALIZE}_F$.

<table>
<thead>
<tr>
<th>$x_w$</th>
<th>$y_w$</th>
<th>counterexample</th>
<th>reason</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 0</td>
<td></td>
<td>$w(1) r(1) w(1)$</td>
<td>not unique</td>
</tr>
<tr>
<td>0 1</td>
<td></td>
<td>$w(1) r(1) w(2)$</td>
<td>not unique</td>
</tr>
<tr>
<td>1 0</td>
<td></td>
<td>$w(1) w(2) r(2)$</td>
<td>not unique</td>
</tr>
<tr>
<td>1 1</td>
<td></td>
<td>$w(1) w(2)$</td>
<td>not unique</td>
</tr>
</tbody>
</table>

Thus any enumeration $F$ including the executions above results in an unsatisfiable $\text{NORMALIZE}_F$.

Satisfiability of $\text{NORMALIZES}_F$ does not necessarily lead to a unique matching scheme, since $\text{NORMALIZE}_F$ can have multiple satisfying assignments. Furthermore, $\text{NORMALIZE}_F$ does not necessarily normalize $I$. For one reason, $F$ may not be a complete set of examples of executions and invalid projections. Second, our simple characterization of $\text{NORMALIZE}_F$ does not rule out the other reasons for a given example $(e, e') \in F$ to be an invalid projection, e.g., that the removed operations do not constitute multiple matches. However, we believe that checking satisfiability of $\text{NORMALIZE}_F$ 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 enumerates the sequential histories 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 with a list of human-readable ADT violation patterns.

9.1 **The Atomic Register**

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

- $\text{write}(v)$ stores the value $v$, and
- $\text{read} \Rightarrow v$ 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:

- $\text{write}(v)$ operations match themselves, and
- $\text{read} \Rightarrow v$ 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 $\text{add}(v)$ 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 register ADT only relates data values via equality.

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:

- $[1:3] \text{read} \Rightarrow 1$ (RO) #
- $[2:2] \text{read} \Rightarrow -$ (RO) #
- $[1:1] \text{write}(1)$ #
- $[2:2] \text{write}(2)$ #

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:

- $\text{add}(v)$ adds the value $v$ to the collection, and
- $\text{remove} \Rightarrow v$ 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:

- $\text{add}(v)$ operations match themselves, and
- $\text{remove} \Rightarrow v$ 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 $\text{add}(v)$ 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:

- $[1:3] \text{remove} \Rightarrow 1$ #
- $[1:1] \text{add}(1)$ #
- $[2:2] \text{remove} \Rightarrow -$ (RO) #
- $[1:1] \text{add}(1)$ #
- $[2:2] \text{add}(2)$ #
- $[3:2] \text{remove} \Rightarrow 2$ #
- $[4:1] \text{remove} \Rightarrow 1$ #

https://github.com/imdea-software/violin

Our current implementation is limited to atomic ADTs. For non-atomic ADTs, the enumeration must cover all $k$-width histories, for some $k \in \mathbb{N}$.
The histories computed for the atomic stack ADT are nearly identical, substituting only the bottom two histories for the following:

\[ [3:1] \text{remove} \Rightarrow 1 \; \# \quad [2:2] \text{add} \Rightarrow - \; (\text{RO}) \; \# \quad [2:1] \text{lock} \Rightarrow - \; (\text{RO}) \; \#
\]

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:

- \( \text{insert}(u) \Rightarrow v \) inserts the value \( u \) to the collection,
- \( \text{remove}(u) \Rightarrow v \) removes \( u \) from the collection, and
- \( \text{contains}(u) \Rightarrow v \) 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 returning \( v = - \) matches its unique value; otherwise, \( v \neq u \). We give a faithful matching scheme over executions which insert unique values as follows:

- any operation returning \( v = - \) matches itself, and
- any operation returning \( v \neq u \) matches the unique \( \text{insert}(v) \Rightarrow - \) 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:

\[
\begin{array}{c}
[1:1] \text{insert}(1) \Rightarrow 1 \; (\text{RO}) \; \# \\
[1:1] \text{remove}(1) \Rightarrow 1 \; (\text{RO}) \; \#
\end{array}
\]

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 proving two methods:

- \( \text{lock}(u) \Rightarrow v \) acquires the lock resource,
- \( \text{unlock} \Rightarrow v \) 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 \( \text{lock}(u) \) 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 \( v \neq u \) matches the unique \( \text{lock}(v) \Rightarrow - \) 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:

\[
\begin{array}{c}
[1:1] \text{lock}(1) \Rightarrow 1 \; (\text{RO}) \; \#
\end{array}
\]

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](#10)) implements a collection of data values with first-in-first-out (FIFO) removal order, proving three methods:

- \( \text{give}(v) \) adds the value \( v \) to the queue,
- \( \text{take} \Rightarrow v \) removes the value \( v \), and
- \( \text{steal} \Rightarrow v \) 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 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:

\[
\begin{array}{c}
[1:1] \text{give}(1) \Rightarrow 1 \; (\text{RO}) \; \#
\end{array}
\]

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

---

10 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.
While we expect this calculation to remain feasible given that ADT optimizations can be applied, theoretical foundations for such optimizations are not always straightforward. In this work, we propose a method for inferring symbolic ADT specifications, which rely on relations (besides equality) on method arguments and return values. We introduce patterns expressed as interval orders, which can track additional relations among data values.

Second, ADTs whose specifications rely on relations (besides equality) on method arguments and return values cannot be expressed with our matching-scheme based notion of histories. A notable example is the priority queue, whose dequeue operations return the smallest/largest enqueued 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 a 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. Choosing matches arbitrarily would be unsound, and result in inferring ADTs with 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, barriers, and exchangegers, 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 of 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 arguments and return values cannot be expressed with our matching-scheme based notion of histories. A notable example is the priority queue, whose dequeue operations return the smallest/largest enqueued 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 a 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. Choosing matches arbitrarily would be unsound, and result in inferring ADTs with 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, barriers, and exchangegers, 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 of 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 arguments and return values cannot be expressed with our matching-scheme based notion of histories. A notable example is the priority queue, whose dequeue operations return the smallest/largest enqueued 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 a 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. Choosing matches arbitrarily would be unsound, and result in inferring ADTs with 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, barriers, and exchangegers, 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 of 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.
\textbf{Proof.} By definition, for every interval order \((A, \preceq, \ell)\), there exists a mapping \(f\) from elements of \(A\) to intervals on \(\mathbb{N}\) such that for every \(x, y \in A\), if \(x \preceq y\), then the interval \(f(x)\) ends before the interval \(f(y)\) (i.e., the upper bound of \(f(y)\) is strictly smaller than the lower bound of \(f(x)\)). Therefore, in the following, we assume that an interval order is a multiset \(\Gamma\) of triples of the form \([i, j, a]\) where \([i, j]\) is an interval on \(\mathbb{N}\) and \(a\) is a symbol in \(\Sigma\).

We prove that another order \(\leq\) stronger than the embedding order \(\preceq\), is a wqo. The order \(\leq\) is defined by: \(\Gamma_1 \leq \Gamma_2\) if 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 any 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 \(\Gamma_1\), 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 \(i < k\) (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 \(i' < k'\).

We say that \(h\) witnesses \(\Gamma_1 \leq \Gamma_2\).

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 \(\Gamma_i \leq \Gamma_j\). Also, assume that \(\Gamma_1\) is the minimal size interval order that can start a bad sequence. \(\Gamma_2\) is the minimal order that can continue a bad sequence starting with \(\Gamma_1\), and so on.

For each \(\Gamma_k\), let \(\text{Min}(\Gamma_k)\) be a triple \([i, j, a]\) such that (1) \(i\) is the minimal lower bound of an interval in \(\Gamma_k\), and (2) \(j\) is the minimal upper bound of an interval in \(\Gamma_k\) with lower bound \(i\). Also, let \(\text{Init}(\Gamma_k) = (a, P_k)\), where \(\text{Min}(\Gamma_k) = [i, j, a]\), for some \(i\) and \(j\), and \(P_k\) is the multiset of symbols labeling elements of \(\Gamma_k\) that are incomparable to \(\text{Min}(\Gamma_k)\). 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 \(\text{Init}\). For each \(\Gamma_k\), let \(A_k\) be the interval order obtained from \(\Gamma_k\) by removing the triple \(\text{Min}(\Gamma_k)\). By the minimality assumption, the infinite sequence \(\Gamma_1, \Gamma_2, \ldots, \Gamma_{k_1}, \Gamma_{k_2}, \ldots\) is not bad. Therefore, there exists \(m < n\) such that \(A_m \preceq A_n\).

It remains to prove that \(\Gamma_{k_m} \preceq \Gamma_n\), when \(\text{Init}(\Gamma_{k_m}) = \text{Init}(\Gamma_n)\) and \(A_m \preceq A_n\). Let \(h\) be the injective function witnessing \(A_m \preceq A_n\). We prove that the extension \(h'\) of \(h\) between \(\Gamma_{k_m}\) and \(\Gamma_n\), defined by \(h'(\text{Min}(\Gamma_{k_m})) = \text{Min}(\Gamma_n)\) and \(h'(t) = h(t)\), for all \(t \in A_m\), witnesses \(\Gamma_{k_m} \preceq \Gamma_n\).

Clearly, \(h'\) preserves the labeling. To prove that \(h'\) preserves the ordering constraints, let \(\text{Min}(\Gamma_{k_m}) = [i, j, a]\) and \([k, l, b]\) in \(\Gamma_{k_m}\) such that \(j < k\). Also, let \(\text{Min}(\Gamma_n) = [i', j', a]\) and \([k', l', b]\) be \(h'(k, l, b) = [k', l', b]\). Assume by contradiction that \(k' \leq j'\). Since \(h\) is injective, there exists an element \([e, f, c]\) \(\in \Gamma_{k_m}\) comparable to \(\text{Min}(\Gamma_{k_m})\) which is mapped to an element \([e', f', c]\) greater than \(\text{Min}(\Gamma_{k_m})\) (because, by definition, \(\Gamma_{k_m}\) and \(\Gamma_n\) have the same number of elements comparable to \(\text{Min}(\Gamma_{k_m})\) and respectively, \(\text{Min}(\Gamma_n)\)). Since \([e, f, c]\) is comparable to \([i, j, a]\), we obtain that \(e \preceq j\). Also, by the current assumptions, \(k' \leq j' < c'\). Therefore, there exist two elements \([e, f, c]\) and \([k, l, b]\) such that the interval \([e', f', c]\) starts before \([k', l', b]\) which is 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 \(A_m \preceq A_n\).

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

Finally, since \(\leq\) is stronger than \(\preceq\), we get that \(\leq\) is a wqo on width-bounded labeled interval orders.

\textbf{Acknowledgments} This work is supported in part by the VECLIB project (ANR-14-CE28-0018), the N-GREENS Software project (Ref. S2013/ICE-2731), and AMAROUT-II (EU-FP7-COFUND-291803).