Automated Inference of Atomic Sets for Safe Concurrent Execution

Peter Dinges, Minas Charalambides, and Gul Agha

Department of Computer Science
University of Illinois at Urbana–Champaign, USA
pdinges@acm.org, charala1@illinois.edu, agha@illinois.edu

Abstract. Atomic sets are a synchronization mechanism in which the programmer specifies the groups of data that must be accessed as a unit. The compiler can check this specification for consistency, detect deadlocks, and automatically add the primitives to prevent interleaved access. Atomic sets relieve the programmer from the burden of recognizing and pruning execution paths which lead to interleaved access, thereby reducing the potential for data races. However, manually converting programs from lock-based synchronization to atomic sets requires reasoning about the program's concurrency structure, which can be a challenge even for small programs. Our analysis eliminates the challenge by automating the reasoning. Our implementation of the analysis allowed us to derive the atomic sets for large code bases such as the Java *collections* framework in a matter of minutes. The analysis is based on execution traces; assuming all traces reflect intended behavior, our analysis allows safe concurrency by preventing unobserved interleavings which may harbor latent *Heisenbugs*.

1 Introduction

A program that synchronizes every single field access for each concurrently used object can still exhibit *high-level* data races [1,2]: fields are often connected through invariants and must be updated together to maintain the object's consistency [19,12]. For example, the value of the `length` field of a list object must equal the number of elements in the array that stores the list entries. Interleaved access to such fields from concurrent threads can expose or produce an inconsistent state in the object containing those fields.

High-level data races may be prevented with control-based synchronization mechanisms such as *locks*. However, to protect a group of data fields, the programmer must recognize and use locks to prune all execution paths that result in problematic interleavings. This requires complicated non-local reasoning over all execution paths. An alternative is to use data-centric synchronization [19,6,3] which localizes the reasoning by asking the programmer for annotations specifying which fields of an object are connected by a semantic invariant. A compiler can use these annotations to add primitives which prevent interleaved access to fields in the same semantic unit. This reduces the potential for high-level

data races on execution paths that the programmer may not have conceived of. Furthermore, the annotations can be statically checked for consistency [6] and deadlock-freedom [15].

Experience with converting a set of concurrent Java programs to data-centric synchronization shows that the annotations are expressive, and that the approach may achieve good performance [6]. However, while the end-results are encouraging, the conversion itself is time-consuming and can take several hours even for a relatively small and simple program. The difficulty in doing such conversion is understanding the program's concurrency structure which can be complicated even for small code sizes. For large legacy programs, understanding the concurrency structure is a daunting challenge, likely requiring a much higher time-investment for conversion, and resulting in higher error rates. There are two kinds of problems that result: unrelated fields may be connected by annotations (commission errors), or connections may be missed (omissions). The first type of error reduces available concurrency and the second type can result in an incorrect execution. Both problems occur in the six manually converted programs we examined. In two cases, the annotations accidentally introduce a global lock, and in two other cases, annotations for the synchronization of shared objects are omitted.

The contribution of this paper is a method for automated analysis of a program's concurrency structure which enables converting the program from control-centric to data-centric synchronization. This allows developers to transition to control-centric synchronization without having to pay for the conversion. Specifically, we provide a novel algorithm for the automatic inference of *atomic sets*, *units of work*, and *aliases* for data-centric synchronization of shared memory multi-threaded programs (for purposes of exposition, we use Java). Using program execution traces as input, our analysis algorithm is path- and flow-sensitive; existing synchronization mechanisms operate as guiding constraints. The algorithm fully supports arrays, as well as cyclic data structures. It assumes traces are provided or can be collected, and that such traces reflect programmer intent (e.g., such traces may be part of a testing phase). The inferred concurrency structure enables safe execution because it automatically prevents schedule-dependent *Heisenbugs* which are (by definition) unlikely to manifest during tracing [20].

2 Background: Atomic Sets

The algorithm presented in this paper infers annotations for data-centric synchronization using *atomic sets* [6]. An atomic set is a group of data fields inside an object indicating that the fields are connected by a consistency invariant. Objects can contain multiple disjoint atomic sets. Recall the list example from the introduction: the value of the list's length field must equal the number of elements in the entries array used to store the list entries. Hence, the fields length and entries form an atomic set. Figure 1 shows the respective code and annotations in AJ, a Java dialect that supports atomic sets. The **atomicset** statement in

```
1 class List {
2   atomicset L;
3   atomic(L) int length;
4   atomic(L) Object[] entries;
5
6   public void get(int index) {
7     if (0 <= index && index < length) {
8       return entries[i];
9   } else {
10        return null;
11  }
12
13  public void addAll(unitfor(L) List other) {
14    length = length + other.length;
15    /*...*/
16  }
17 }
18
19 class DownloadManager {
20  atomicset U;
21  atomic(U) List urls|L=this.U|;
22
23  public void downloadNext() {
24    Object u = urls.get(0);
25    if (u != null) {
26      urls.remove(0);
27      download(u);
28    }
29  }
30 }
```

Fig. 1. Sample classes in the AJ dialect of Java which adds data-centric synchronization via the annotations **atomicset**, **atomic**, **unitfor**, and |A=**this**.B|.

line 2 declares an atomic set L; the **atomic**(L) annotations of the field declarations add the fields length and entries to L.

Instead of requiring an explicit expression of the consistency invariant like length == entries.length, an atomic set is complemented by one or more *units of work*. A unit of work is a method that preserves the consistency of its associated atomic sets when executed sequentially. Thus, atomic sets can ensure the application's consistency by inserting synchronization operations that guarantee the sequential execution of all units of work. By default, all non-private methods of a class are units of work for all atomic sets declared in the class or any of its subclasses. Like field declarations, atomic sets use classes as scopes, but are instance specific at runtime. For example, the method get(**int**) in Figure 1 is a unit of work for the atomic set L of its containing List object. The atomic sets of

two List objects are distinct. Other methods can be declared units of work with the **unitfor** annotation. In line 13 of Figure 1, the method addAll(List) is not only a unit of work for the atomic set L of its own List object, but also for the atomic set L of its argument. The effect of these unit of work declarations is that two threads, t_1 and t_2, that concurrently invoke get(**int**) and addAll(List) on a List l cannot interleave when accessing l's field: either t_1 executes get(**int**) first, or t_2 executes addAll(List) first. The interleaved case where t_2 has updated l.length but not l.entries, which causes t_1 to violate the array bounds cannot occur.

Aliases extend atomic sets beyond object boundaries. An alias merges the atomic set containing a field with an atomic set in the object that is the field's value. For example, consider the DownloadManager class in Figure 1. The alias annotation |L=**this**.U| of the urls field declaration combines the atomic set L with the atomic set U. Hence, the downloadNext() method is a unit of work for this combined atomic set; its access to the urls list cannot be interleaved. This guarantees that no other thread can empty the list between the invocations of get(**int**) and remove(**int**). Coarse-grained atomic structures like trees can be implemented with aliases. AJ furthermore supports aliasing of array-elements, and even aliasing of atomic sets in the elements of arrays.

Additionally, AJ recognizes a number of advanced annotations like partial **unitfor** declarations and fastread. These simplify the specification of units of work and guide the AJ compiler towards generating more efficient concurrency-control code. Since these annotations are secondary, we do not consider their inference in this paper.

3 Algorithm Synopsis

This section outlines our algorithm for inferring the atomic set, alias, and unit of work annotations for data-centric synchronization.

The assumption behind our approach is that the methods of a program perform semantically meaningful operations. Thus, the fields accessed by a method are likely connected by some semantic invariant. The set of fields that a method accesses atomically is consequently a candidate atomic set; the method itself is a candidate unit of work for this atomic set. For example, the get(**int**) method that retrieves an entry from a list checks whether the requested index is less than the list's length before accessing the entries field. If the method always executes atomically, then this suggests having an atomic set containing the fields length and entries in the class List. Method get(**int**) is a unit of work for this atomic set.

A slight complication arises because high-level semantic operations often employ low-level operations. For example, the get(**int**) method might not access the length field directly, but rather check the list's length by calling getLength(). To nevertheless discover the semantic relationship between length and entries, we consider the fields accessed by a method to transitively include the fields accessed by called methods as well. Thus, the access to length in method getLength() propagates backwards to its caller get(**int**). This restores the desired connection between the fields length and entries.

Our algorithm builds upon these insights. It records the field access events, as well as method entries and exits during the execution of a program. Replaying these events on an abstract machine, it observes which fields each method transitively accesses during execution, and furthermore whether the access was atomic for one thread, or interleaved between multiple threads. The algorithm aggregates these dynamic observations, translates them to the static class structure of the program, and merges them into suggested atomic sets. After forming the atomic sets, it assigns the methods as units of work for the atomic sets that match the observed field access patterns.

To infer aliases between atomic sets, the algorithm tracks the *access paths* of accessed fields. An access path is a sequence of field names that leads from one of the method's parameters to the accessed field. For example, consider a download manager object that stores a list of URLs in its urls field. A method of the download manager that fetches and downloads a URL from the list (by invoking get(**int**)) observes access to the paths **this**.urls.length and **this**.urls.entries. If this always happens atomically, then it suggests an alias from the field urls in the download manager to the atomic set that contains the fields length and entries in class List.

Inference Procedure

Making above description more precise, our algorithm infers the annotations for data-centric synchronization in the following phases:

(1) Recording the field accesses, as well as method entries and exits during the execution of a program. This yields a field access trace as described in section 4.
(2) Replaying the field access trace on an abstract machine, observing which fields each method transitively accesses during execution, and furthermore detecting whether the access was atomic for one thread, or interleaved between multiple threads. Aggregating these dynamic observations and translating them to class scopes and field names yields the suggested atomic sets. This phase is formalized in section 5.
(3) Inferring atomic sets and aliases by merging the suggestions. Afterwards, methods are assigned as units of work for the atomic sets that match the observed field access patterns. See section 6.

4 Field Access Traces

The set of events that are relevant for the inference algorithm are recorded in *field access traces*, whose purpose is to record enough information to allow reproducing the field read and write patterns of methods in the different threads of a multithreaded program.

While conceived in the context of Java, the events logged in the traces are generic to object-oriented languages: they only presume the existence of classes

that contain methods and fields. Thus, field access traces can be obtained from programs written in other languages, which makes our atomic set inference algorithm applicable to any of these languages.

Formally, field access traces are sequences of events generated by the following grammar, in which names in angle brackets denote syntactic categories:

$$
\begin{aligned}
\langle \text{Trace} \rangle &::= (\ t,\ \langle \text{Event} \rangle\ )^\star \\
\langle \text{Event} \rangle &::= \langle \text{Enter} \rangle \mid \langle \text{Get} \rangle \mid \langle \text{Put} \rangle \mid \langle \text{Exit} \rangle \\
\langle \text{Enter} \rangle &::= \mathsf{enter}(\ m(v_1, ..., v_\ell)\ ) \\
\langle \text{Get} \rangle &::= \mathsf{get}(\ o.f,\ v\ ) \\
\langle \text{Put} \rangle &::= \mathsf{put}(\ o.f,\ v\ ) \\
\langle \text{Exit} \rangle &::= \mathsf{exit}
\end{aligned}
$$

Each event is marked with its source thread t to preserve the concurrency of events despite their serialization in the trace. There are four types of events:

(1) The enter event signals the start of the execution of a method. It supplies the name m of the entered method, along with the values v_1 to v_ℓ of the method parameters.
(2) Read access to an object's field is recorded as a get event. The event contains the identity o of the object, and the name f of the accessed field. It furthermore stores the value v read from the field.
(3) Writing fields produces put events. These contain the same information as get events.
(4) An exit event denotes that the current method finished, and control will return to the calling context. This happens for both regular returns and uncaught exceptions.

The enter, get, and put events record values. These values are either object identities or the special symbol $\perp$ which denotes all primitive, non-object values. By tracking globally unique object *identities* (like memory addresses), the observation phase of the algorithm can recognize objects across different scopes. In each scope, the algorithm can then resolve the respective *identifiers* for the object (like the variable name x). By treating classes as regular objects with a unique identity, access to static fields yields regular get or put events in the trace.

5 Access Pattern Observation

This section explains how the algorithm extracts the sets of fields that a method accesses from a field access trace. The next phase of the algorithm (section 6) aggregates these sets to infer the atomic sets, units of work, and aliases.

5.1 Abstract Machine

The observation phase extracts the sets of accessed fields by replaying a field access trace using an abstract machine that monitors the field access in objects.

Hence, the abstract machine executes the events in the trace and applies their effects to its state. To be able to detect interleaved accesses, the abstract machine explicitly models the threads present in the trace, as well as the heap shared by these threads. The model furthermore contains a stack for every thread. The frames of the stack include a component for storing the accessed fields, and a component for tracking the static names of objects. Finally, an extra field in the abstract machine's state stores the observations made during playback. More formally, a *configuration* of the abstract machine

$$(\gamma, \mathit{stacks}, \mathit{last}, \mathit{obs}) \in \text{Config}$$

consists of the global timestamp γ, the state of the stacks, the last-written timestamps of all heap objects, and the collected observations. The initial configuration has timestamp 0, and empty stacks, heap, and observations. The exact domains of the four components of Config are explained below. The transition function

$$\delta : \text{Config} \times \big(t, \langle \text{Event} \rangle\big) \to \text{Config}$$

defined in Figure 2 computes the successor configuration that includes the effects of the given input event from a field access trace.

5.2 Scope Management

Observing whether a field access was atomic or interleaved must be recorded in the syntactic scope—that is, the method—in which the access occurred. The abstract machine thus maintains a stack for each thread in the *stacks* component of the configuration, which is a thread-id indexed dictionary: $\mathit{stacks} \in \text{Thread} \to \text{Stack}$ with

$$\mathit{stacks}(t) \in \text{Stack} = (\text{Method} \times \text{Timestamp} \times \text{NameEnv} \times \text{FieldAccsEnv})^\star.$$

The stack frames $(m, \varepsilon, ne, \mathit{fae})$ consist of the name of the method m that defines the scope, the timestamp ε when the scope was created, as well as a *name environment* ne, and a *field access environment* fae.

Similarly to the customary call stack that a thread owns during execution, entering a method pushes a new frame on the top of the stack; exiting a method pops the top frame off the stack. The auxiliary methods $\mathit{push} : \text{Stack} \to \text{Stack}$ and $\mathit{pop} : \text{Stack} \to \text{Stack}$ implement this functionality. To reduce the nesting level of parentheses, we furthermore define auxiliary methods *push* and *pop* that operate on the *stacks* dictionary and accept a second index parameter that specifies which stack should be modified. Thus,

$$\mathit{pop}(\mathit{stacks}, t) = \mathit{stacks}\big[t \mapsto \mathit{pop}\big(\mathit{stacks}(t)\big)\big]$$

and similarly for *push*. The bracket denotes a point-wise update: $g[x \mapsto y]$ is the function h with $h(z) = y$ if $z = x$ and $h(z) = g(z)$ otherwise. Further auxiliary stack management functions are *top*, which returns the topmost frame on a stack; and *replaceTopWith*, which puts a given frame in place of the topmost frame and returns the resulting stack.

$\delta\big((\gamma, \mathit{stacks}, \mathit{last}, \mathit{obs}), (t, e)\big) = (\gamma + 1, \mathit{stacks}', \mathit{last}', \mathit{obs}')$. Unless defined otherwise, $\mathit{stacks}' = \mathit{stacks}$, $\mathit{last}' = \mathit{last}$, and $\mathit{obs}' = \mathit{obs}$.

If $e = \mathsf{enter}\big(m(v_1, \ldots, v_\ell)\big)$ then
| $\mathit{stacks}' = \mathit{push}\big(\mathit{stacks}, t, \mathit{frame}\big(\gamma, m(v_1, \ldots, v_\ell)\big)\big)$.

If $e = \mathsf{get}(o.f, v)$ then
| $\mathit{stacks}' = \mathit{recordRead}(\mathit{stacks}'', t, o, f)$;
where
| $$\mathit{stacks}'' = \begin{cases} \mathit{recordName}(\mathit{stacks}, t, o, f, v) & \text{if } v \neq \bot, \\ \mathit{stacks} & \text{otherwise.} \end{cases}$$

If $e = \mathsf{put}(o.f, v)$ then
| $\mathit{stacks}' = \mathit{recordWrite}(\mathit{stacks}'', t, o, f, \gamma)$;
| $\mathit{last}' = \mathit{last}\big[(o, f) \mapsto \gamma\big]$;
where
| $$\mathit{stacks}'' = \begin{cases} \mathit{recordName}(\mathit{stacks}, t, o, f, v) & \text{if } v \neq \bot, \\ \mathit{stacks} & \text{otherwise.} \end{cases}$$

If $e = \mathsf{exit}$ then
| $\mathit{stacks}' = \mathit{popAndMerge}(\mathit{stacks}, t)$,
| $\mathit{obs}' = \mathit{update}\big(\mathit{obs}, \mathit{observedAccess}\big(\mathit{top}(\mathit{stacks}, t)\big)\big)$.

Fig. 2. Configuration transition function of the abstract machine used for observing interleaving patterns in field access traces. The *recordRead* and *recordWrite* functions update the configuration parts that allow interleaving detection when the method exits. The *recordName* function tracks the information used to resolve the access paths of fields.

5.3 Interleaving Detection

The last-written timestamps of the heap objects, *last* in a configuration, represents the abstract machine's model of the memory shared by all threads. Its purpose is to help determine whether the access to an object's field was atomic or interleaved. Interleaved access is detected by comparing timestamps.

For each field of every object in memory, the access history records the timestamp of the last write access to the field. When a thread reads a field during replay, it stores the timestamp in the field access environment *fae* of its local stack frame. At method exit, comparing the local timestamp (in *fae*) to the memory timestamp (from *last*) reveals whether an intermediate update of the field occurred. By keeping its local timestamp current when writing a field (see the put case in Figure 2), a thread can be certain that the intermediate write originated in another thread, and thus that an interleaved access to the field occurred. Note that read–read access is not considered to be interleaved access because it is not a race condition, but an optimization whose semantics express peaceful co-existence. Formally, timestamps are simply natural numbers,

Timestamp $= \mathbb{N}$; we use Greek letters to denote them. The access history is a dictionary of timestamps indexed by object-id–field pairs:

$$\mathit{last} : \text{Object} \times \text{Field} \to \text{Timestamp}.$$

The field access environment *fae* in a stack frame tracks which fields of which object the method accesses during its invocation. For each object–field pair, it stores a tuple with access information that is updated by read and write events. When the method exits, the algorithm uses this information to determine whether the field access was atomic in the method's scope, or interleaved.

Each such tuple consists of two status flags and two timestamps. The first status flag describes the kind of access that occurred: none, read, or written. The second status flag signifies whether the access was direct or indirect—occurred in the current scope, or the scope of a called method. The timestamps record the version of the field's value that was first retrieved from memory, and the one that was last written by the current thread respectively.

$$\begin{aligned} \text{AccsStat} &= \{\text{none}, \text{read}, \text{written}\} \times \{\text{direct}, \text{indirect}\} \\ \text{FieldAccsEnv} &= \text{Object} \times \text{Field} \to \text{AccsStat} \times \text{Timestamp} \times \text{Timestamp}. \end{aligned}$$

Interleaving detection for a method's scope happens when the method exits. To ensure atomic access to a field that has only been read by the method and its callees, it suffices to verify that the last write access, $\mathit{last}(o, f)$, happened before the method was entered. In symbols: $\mathit{last}(o, f) < \varepsilon$. Fields that have been written require more attention because $\mathit{last}(o, f)$ is updated by the write event. The field access environment must maintain enough information to distinguish this update from updates by other threads. Two timestamps are necessary to achieve this: the initial timestamp α that was overwritten during the first access, and the latest timestamp ω written by the current access. At the time ρ when the method exits, the access has been atomic if the overwritten value dated back before the method entry time ε, and the last write access happened at ω. In symbols: $\alpha < \varepsilon$ and $\mathit{last}(o, f) = \omega$.

Defining the result set of interleaving detection as

$$\text{IntlStat} = \{\text{none}, \text{atomic}, \text{interleaved}\} \times \{\text{direct}, \text{indirect}\}$$

the detection procedure itself is

$$\mathit{intl}(o, f) = \begin{cases} (\text{none}, d) & \text{if } s = \text{none}; \\ (\text{atomic}, d) & \text{if } s = \text{read and } \mathit{last}(o, f) < \varepsilon; \\ (\text{atomic}, d) & \text{if } s = \text{written}, \alpha < \varepsilon \text{ and } \mathit{last}(o, f) = \omega; \\ (\text{interleaved}, d) & \text{otherwise}, \end{cases}$$

where $o \in \text{Object}$, $f \in \text{Field}$ and $\mathit{fae}(o, f) = (s, d, \alpha, \omega)$ with $\mathit{fae} \in \text{FieldAccsEnv}$. The direct-access flag d is copied verbatim; it influences the weight of observations when forming atomic sets.

Accessing a field updates its status flags and timestamps in the field access environment. The goal is keeping the data consistent to enable correct interleaving detection. Reading a field for the first time creates a new entry in the environment that uses the access history timestamp $last(o, f)$ for both local timestamps. However, subsequent reads do not modify the timestamps. This guarantees that the timestamps from prior operations are preserved. Since interleaving detection for read fields ignores the local timestamps, this is unproblematic. The direct access flag set is set in both cases. The configuration transition function δ uses the function *recordRead* to implement this behavior in its handler for get events.

$$recordRead(stacks, t, o, f) = stacks[t \mapsto stack']$$

for $t \in$ Thread, $o \in$ Object, $f \in$ Field, $(m, \varepsilon, ne, fae) = top(stacks, t)$ and $(s, d, \alpha, \omega) = fae(o, f)$ with

$$stack' = replaceTopWith\big((m, \varepsilon, ne, fae'), stacks(t)\big),$$

$$fae' = \begin{cases} fae\big[(o, f) \mapsto \big(\text{read}, \text{direct}, last(o, f), last(o, f)\big)\big] & \text{if } s = \text{none;} \\ fae\big[(o, f) \mapsto (\text{read}, \text{direct}, \alpha, \omega)\big] & \text{if } s = \text{read;} \\ fae\big[(o, f) \mapsto (\text{written}, \text{direct}, \alpha, \omega)\big] & \text{if } s = \text{written.} \end{cases}$$

The function *recordWrite* handles put events. Like *recordRead*, it uses the access history timestamp $last(o, f)$ for the locally stored *initial* access timestamp α during the first access to a field, and leaves α unchanged during subsequent writes. In contrast to *recordRead*, *recordWrite* updates the local *last* access timestamp ω to the abstract machine's current timestamp γ. Since the interleaving detection function *intl* only checks for concurrent access before α and after ω, the update must guarantee that no interleaved access occurred between α and ω. Thus, *recordWrite* updates ω only if no other thread wrote the field in the meantime. Ceasing updates on detected interleaved access ensures that the same information is available to detect the interleaving when the method exits. Using the same symbols as above, it is

$$recordWrite(stacks, t, o, f, \gamma) = stacks[t \mapsto stack']$$

with

$$stack' = replaceTopWith\big((m, \varepsilon, ne, fae'), stacks(t)\big),$$

$$fae' = \begin{cases} fae\big[(o, f) \mapsto \big(\text{written}, \text{direct}, last(o, f), \gamma\big)\big] & \text{if } s = \text{none,} \\ fae\big[(o, f) \mapsto (\text{written}, \text{direct}, \alpha, \gamma)\big] & \text{if } s \neq \text{none} \\ & \text{and } last(o, f) = \omega, \\ fae & \text{otherwise.} \end{cases}$$

Above description covers updates to the current field access environment. To observe field accesses across scopes—that is, in called methods—the information collected in the called methods must be merged into the calling method's field access environment. As with *recordWrite*, loss of interleaving information must

be prevented; the merge operator $\triangleright$ only updates environment entries if the access was atomic. The merge operator $\triangleright$ for field access environments is the point-wise extension of the entry merge operator $\triangleright$. For two field access environments fae_1, fae_2 and $(s_k, d_k, \alpha_k, \omega_k) = fae_k(o, f)$, $k = 1, 2$, it is

$$(s_1, d_1, \alpha_1, \omega_1) \triangleright (s_2, d_2, \alpha_2, \omega_2) = \begin{cases} (s_1, \text{indirect}, \alpha_1, \omega_1) & \text{if } s_2 = \text{none;} \\ (\text{written}, d_2, \alpha_2, \omega_1) & \text{if } s_2 \neq \text{none, } \alpha_1 \leq \omega_2 \\ & \text{and } s_1 = \text{written;} \\ (s_2, d_2, \alpha_2, \omega_2) & \text{otherwise.} \end{cases}$$

Merging of environments occurs when a method exits. The process is formalized by the function *popAndMerge*. With $(m_1, \varepsilon_1, ne_1, fae_1) = top\big(stacks(t)\big)$ and $(m_2, \varepsilon_2, ne_2, fae_2) = top\big(pop\big(stacks(t)\big)\big)$, define

$$popAndMerge(stacks, t) = stacks[t \mapsto stack'],$$

where

$$stack' = replaceTopWith\big((m_2, \varepsilon_2, ne_2, fae_1 \triangleright fae_2), pop(stacks, t)\big).$$

5.4 Syntactic Name Mapping

Atomic sets are defined using the static names of fields in the source code and, through aliases, can extend across object boundaries. However, field access traces are recorded during runtime and only contain the immediate field name that is accessed within an object (see section 4). To be able to correctly infer aliases, as in the example of section 3, it is necessary to determine the full *access path* that was used to access the field. The abstract machine therefore tracks the object graph generated by the trace and creates a mapping from dynamic to static names.

An access path for a value v is a sequence of static names that leads to v.

$$\text{AccsPath} = \text{Field } (\text{"."} \text{ Field})^\star.$$

The abstract machine uses the name environment component of stack frames to generate the access paths of objects. Each name environment is an object graph whose edges are labeled with the static names in its scope. The nodes of the graph are the dynamic object identities. To resolve the static names of an object identity o, the abstract machine aggregates the labels along all paths from a virtual root object to o.

For each `get` or `put` event to a field f of reference type in the trace, the graph is extended with an edge from the object containing f to f's value (the object to which f refers). The label of this edge is f. Formally, we define name environments as the set of labeled graph edges (which implies the set of nodes)

$$\text{NameEnv} = \mathcal{P}\big((\text{Object} \cup \{\text{root}\}) \times \text{Field} \times \text{Object}\big),$$

where root $\notin$ Object. The function for adding an edge is *recordName*,

$$recordName(stacks, t, o, f, v) = stacks[t \mapsto stack'],$$

with $t \in$ Thread, $o \in$ Object, $f \in$ Field and v a value. Its uses the replacement stack

$$stack' = replaceTopWith\big((m, \varepsilon, ne \cup \{(o, f, v)\}, fae), stack\big),$$

for $top(stack) = (m, \varepsilon, ne, fae)$, which is computed by exchanging the top frame in *stack* using the auxiliary function *replaceTopWith*.

The root object is defined to contain fields whose names correspond to those of the method's parameters—including the implicit **this** for instance methods. For consistent handling, static methods are extended with a virtual **class** parameter that plays the role of **this**, but refers to the static class object. The access path to an object therefore always starts with a method parameter; other reachable objects, like global static objects, lack an associated name path. This suffices because units of work cannot be declared for atomic sets of variables other than method parameters. The initialization of name environments takes place in the *frame* function, which also creates an empty field access environment.

$$frame\big(\gamma, m(v_1, \ldots, v_\ell)\big) = (m, \gamma, ne, fae),$$

where

$$\begin{aligned} ne &= \big\{(\text{root}, param(m, i), v_i) \mid i = 1, \ldots, \ell\big\}, \\ fae &= \big[(o, f) \mapsto (\text{none}, \text{indirect}, -, -) \mid (o, f) \in \text{Object} \times \text{Field}\big]. \end{aligned}$$

Because of variable and field name aliasing, the graph may contain parallel edges. A single object may thus be reachable via several access paths. However, since access paths are generated from the field access traces, only paths that were actually used at runtime will appear. Circular data structures create cycles in the graph. The access path generation must detect these to prevent infinite loops. Unlike in the field access stack, updates to the name environment affect only the topmost environment and do not propagate downwards.

5.5 Access Observations

The algorithm collects the observations when the method exits. Using the function *intl* defined above, it determines the interleaving status of all (transitively) accessed fields, and assigns it to the respective access paths.

$$observedAccess(\varepsilon, ne, fae) = \big[p.f \mapsto intl(o, f) \mid p \in accsPathsTo(o, ne)\big]$$

where $accsPathsTo(o, ne)$ are the label sequences along all paths in the object graph *ne* from root to o.

Units of work are defined statically, independently of the execution context. All observations made for a method in different contexts are therefore aggregated

into a single set of observations, which is an entry in the *obs* component of the abstract machine's configuration. The function

$$update(obs, m, ctxobs) = obs\big[m \mapsto \big(obs(m) \sqcup ctxobs\big)\big]$$

uses the point-wise extension of the join operator $\sqcup$ to monotonically update the observations. For $(i_k, d_k) \in \text{IntlStat}$, $k = 1, 2$, it is

$$(i_1, d_1) \sqcup (i_2, d_2) = (i_1 \sqcup i_2, d_1 \sqcup d_2)$$

with

$$i_1 \sqcup i_2 = \begin{cases} \text{none} & \text{if } \{i_1, i_2\} = \{\text{none}\}; \\ \text{atomic} & \text{if interleaved} \notin \{i_1, i_2\} \text{ and atomic} \in \{i_1, i_2\}; \\ \text{interleaved} & \text{if interleaved} \in \{i_1, i_2\}; \end{cases}$$

$$d_1 \sqcup d_2 = \text{direct if direct} \in \{d_1, d_2\}, \text{ else indirect.}$$

6 Atomic Set Formation

This section describes the derivation of atomic sets, aliases, and units of work from the collected observations. As explained in section 3, the fundamental idea is that every set of fields in a class that a method accesses atomically is a suggested atomic set for that class. The method is a unit of work for this atomic set. The algorithm collects the suggestions for each class by aggregating the observations of all methods. It forms the atomic sets by removing fields with interleaved access and merging overlapping suggestions. Finally, it assigns each method as unit of work for the formed atomic sets compatible with the suggestions.

6.1 Observation Pruning

The observations made during replay can be inconsistent regarding witnessed interleaved accesses. A *witness* of an interleaved access to a field f is an access path p ending in f such that $\big(obs(m)\big)(p) = (\text{interleaved}, \_)$ for some method m, where the underscore $\_$ denotes any value. For example, the observations for method downloadNext() may contain the witness **this**.urls and at the same time claim atomic access for **this**.urls.length. However, without the method having atomic access to **this**.urls, the List object containing the field length may be replaced by another thread while the method executes. Thus, the method should not suggest length as a member of an atomic set in the class List.

Consistency of the observations can be restored by dropping the observations for access paths with an interleaved prefix. With $\sqsubseteq$ denoting the prefix relationship between access paths, we define the set of pruned observations as

$$pobs(m) = obs(m)\big[p \mapsto (\text{none}, d) \mid \exists w : w \sqsubseteq p \text{ and } \big(obs(m)\big)(w) = (\text{interleaved}, d)\big].$$

6.2 Atomic Set Formation

Aggregating the pruned field access observations by class yields the suggested atomic sets. Depending on whether an observation concerns a field of the instance object (**this**) of a method or not, the observation is given different weight. Observations concerning the local object indicate a stronger semantic relationship than other observations. To reflect this distinction, such observations, witnesses, and atomic set suggestions are called *internal* for the object's class c (or its subclasses). All other observations, witnesses, and atomic set suggestions are *external* for c.

The internal and external atomic set suggestions are formed using the following inference rules:

- Fields with internal witnesses are excluded from all atomic sets. For a class c, these *non-atomic* fields are

$$nonAtomic(c) = \bigcup_{m \in methods(c)} \big\{ f \mid \big(pobs(m)\big)(\textbf{this}.f) = (\text{interleaved}, direct) \big\}.$$

  Limiting the witnesses to *direct* observations—made in the method's scope, not one of its callees—prevents over-emphasizing witnesses originating in scopes potentially far away in the call chain.
- Fields internally observed to be atomic are assumed to comprise a semantic unit. Each internal suggestion must therefore be a subset of one of the formed atomic sets—minus non-atomic fields. Given a class c, the internal suggestions for atomic sets are

$$\begin{aligned} intAtomic(m) &= \big\{ f \mid \big(pobs(m)\big)(\textbf{this}.f) = (\text{atomic}, \text{direct}) \big\}, \\ intSugg(c) &= \bigcup_{m \in methods(c)} \big(intAtomic(m) \setminus nonAtomic(c) \big). \end{aligned}$$

- Fields externally observed to be atomic have less semantic weight. While still suggesting the membership of the fields in an atomic set, the requirement of membership in the *same* atomic set is dropped. For a class c, the external suggestions for atomic sets are

$$\begin{aligned} extAtomic(m, c) = \big\{ f \mid \exists p : p \neq \textbf{this}, \big(pobs(m)\big)(p.f) = (\text{atomic}, -), \\ \text{and } scope(f) = c \big\} \\ extSugg(c) = \bigcup_{m \in methods(c)} \big(extAtomic(m, c) \setminus nonAtomic(c) \big). \end{aligned}$$

Using only the final segment of name paths in the definitions is no limitation because for every name path observed as atomic or interleaved (that is, not none), an atomic or interleaved observation exists for all its non-empty prefixes. The

reason for this prefix closure is that a field g cannot be accessed without accessing its containing object. Thus, *some* observation exists. After pruning, the observations of all non-empty prefixes indicate atomic access.

To obtain the final atomic sets, above suggestions are merged in a hierarchy representing their semantic weight. Non-atomic fields have the highest priority; they are removed from both internal and external suggestions. Since membership for only one atomic set can be declared per field, overlapping suggestions (containing the same field) are merged by taking their union. The output of the auxiliary function *merge* is a set of pairwise disjoint sets. The final step combines the external suggestions with the internal ones by using them as extensions. For every external suggestion, the algorithm adds its elements to the internal suggestion with which it shares most elements. However, the auxiliary function *extend*, which implements this process, maintains the disjunction of the internal suggestions. Elements that would result in overlapping internal suggestions are not added. The atomic sets $\mathit{atomicSets}(c)$ for a class c inferred by the algorithm are

$$\mathit{extend}\big(\mathit{merge}\big(\mathit{intSugg}(c)\big), \mathit{merge}\big(\mathit{extSugg}(c)\big)\big).$$

6.3 Aliases

The rules for inferring atomic sets focus on the observations about the final segments of name paths. Aliases can be inferred similarly by shifting the focus onto observations about adjacent segments. Recall the example in section 3: observing atomic access for the name path **this**.urls.length not only suggests including length in an atomic set, but also adding an alias from urls to that atomic set. More generally, observing atomic access for the name path $p.f.g$ suggests an alias from f to the atomic set of g, unless a witness against this alias exists. Thus, for a field f, the set of fields whose atomic sets should be aliased by f is

$$\mathit{aliasFields}(f) = \mathit{atomicSucc}(f) \setminus \mathit{nonAtomicSucc}(f)$$

with

$$\mathit{atomicSucc}(f) = \bigcup_{m \in \mathrm{Method}} \big\{ g \mid \exists p : \big(\mathit{pobs}(m)\big)(p.f.g) = (\mathrm{atomic}, -) \big\},$$
$$\mathit{nonAtomicSucc}(f) = \bigcup_{m \in \mathrm{Method}} \big\{ g \mid \exists p : \big(\mathit{pobs}(m)\big)(p.f.g) = (\mathrm{interleaved}, -) \big\}.$$

Unlike in the inference rules for atomic sets, all observations have the same semantic weight. Thus, combining atomic sets is avoided whenever concurrency between them has been observed, even indirectly and externally. The inferred annotations therefore capture the finest granularity of concurrency available in the input trace.

If the set of alias fields contains fields from multiple atomic sets in f's type, then the algorithm cannot infer the right semantics because at most one alias may be defined per field. Thus, it forwards the decision to the programmer,

who has several options: ignore the alias, pick one of the matching atomic sets, or merge the matching atomic sets. Merging the atomic sets is safe and allows full automation, but reduces concurrency. Developing better ways to resolve this choice is future work.

6.4 Units of Work

Inferring the atomic sets for which a method should be a unit of work follows the same principles as inferring aliases. For each parameter of the method, the algorithm determines the set of fields in the parameter's type that the method accessed atomically. The method is then a unit of work for all atomic sets in the parameter's type that contain one of the fields. Exempt from this rule are atomic sets that contain fields that were witnessed as interleaved in the method's scope. To avoid generating a too coarse grained concurrency structure, the algorithm prioritizes the observed interleavings over the more common atomic observations. Under the assumption that all observed interleavings reflect the programmer's intentions, this priority scheme is unproblematic. No choice of a single atomic set is necessary because, unlike alias annotations of fields, multiple **unitfor** annotations can be added to parameters. Consequently, a parameter p of a method m receives a *unitfor* annotation for the following atomic sets:

$$\mathit{units}(m, p, \text{atomic}) \setminus \mathit{units}(m, p, \text{interleaved}),$$

with

$$\mathit{units}(m, q, s) = \big\{A \in \mathit{atomicSets}\big(\mathit{type}(q)\big) \mid A \cap \mathit{paramFields}(m, q, s) \neq \emptyset\big\},$$
$$\mathit{paramFields}(m, q, s) = \big\{f \mid \big(\mathit{pobs}(m)\big)(q.f) = (s, \_)\big\}.$$

Computing the units of work for the implicit parameter **this** of instance methods is unnecessary because all instance methods are units of work for all atomic sets in the class.

7 Implementation

We have implemented the presented algorithm in a tool chain for Java programs[1]. The tool chain consists of a Java byte code instrumenter and an inference tool. The instrumenter uses WALA's [5] Shrike library to insert calls to the field access tracing library into the input byte code. After instrumentation, the target program must be executed to generate field access traces. The traces are the input for the inference tool, which is a Python implementation of the algorithm described in Sections 5 and 6.

[1] Available at `http://osl.cs.illinois.edu`

Limitations The inference tool currently infers only the basic AJ annotations. In particular, the tool cannot infer fast-read, partial and generalized **unitfor**, or internal class annotations. Supporting the inference of these annotations remains as future work. Furthermore, the tool ignores the limitations of the current AJ implementation. Consequently, it does not suggest the required refactorings like making nested classes into top-level classes, adding getter and setter methods, and using only one atomic set per class.

A minor limitation is that annotations have to be applied manually to the source code. More advanced implementations could output patch files that allow developers to automate this step. Additionally, the tool chain currently does not export the access flags of fields. It hence cannot remove fields from the output that are marked as final. Since the access flags are available in the byte code instrumenter, the solution of this issue is a straight forward programming task.

Finally, the implementation choices result in limited performance of the tool. First, explicit logging of field access traces incurs a high performance penalty because a large volume of data must be written to disk. We decided for a split–phase approach because it simplifies the documentation of the (field access traces used for the) experiments, makes results reproducible, and simplifies debugging. Second, implementing the algorithm in Python limits the speed of the inference process. To achieve higher performance, the algorithm could instead be implemented as an online tool that directly summarizes the observations while the program runs.

7.1 Extensions

The preceding sections presented the fundamental algorithm for inferring the atomic sets and units of work in a concurrent program. To enable the algorithm to handle realistic programs, we have extended it with the ability to handle arrays, synchronized blocks, and wait–notify synchronization.

Arrays. The algorithm can be extended to handle arrays by treating indices like field names: accessing index `i` in an array `entries` via `entries[i]` is regarded as a field access `entries.i`. Depending on whether the array element is read or written, the array operation generates a `get` or `put` event. Before forming the atomic sets, all array indices appearing in name paths are renamed to `[]` and their access patterns are merged. This automatically yields the desired array alias specifications supported by AJ [6].

Observing array operations with index precision means that concurrent operations on different parts of the array are not regarded as interleaved access. Thus, one thread reading and writing `entries[0]` while another thread reads and writes `entries[1]` will still result in an annotation that ensures atomicity for all operations on the array and its indices (but not necessarily the values at these indices). As with read–read sharing, we view this as the annotation which reflects the programmer's intentions best; no index of the array was accessed in an interleaved manner. The alternative, no annotation, would allow concurrent updates of the same index.

Synchronized Blocks. In AJ, only methods can be units of work for atomic sets. Consequently, **synchronized** blocks inside a method's body have to be extracted into auxiliary methods to maintain the intended semantics. While the refactoring is straight-forward and can easily be automated, it modifies the source code. To avoid such source code pre-processing, the byte code instrumentation tool treats **synchronized** blocks similar to methods: entering a **synchronized** block generates an enter event; exiting generates an exit event. Avoiding a pre-processing step allows the tool to bundle all necessary source code changes—adding annotations and extracting auxiliary methods—in its output, making their application a single step for the programmer.

Wait–Notify Synchronization. Inside a **synchronized** block, a thread may call the wait() method of the object monitoring the block. The call suspends the current thread and releases the monitor; when the thread is woken up, usually via notify(), it re-acquires the monitor and resumes execution at the next statement. This protocol ensures mutual exclusion between the participating threads. However, because the execution of the waiting thread never leaves the syntactic scope of the **synchronized** block, the inference algorithm falsely identifies access to objects in the scope as interleaved.

To prevent the false detection of interleaved access, the instrumentation tool emits a fake exit event for the **synchronized** block just before the call to wait(), and a fake enter event just after the call. Since the wait() method is defined in class Object and cannot be overwritten, the instrumenter can reliably identify these calls. The solution is similar to the refactoring proposed by Dolby et al. [6], which splits the bodies **synchronized** blocks at wait() calls and moves the halves into separate methods.

7.2 Heuristics

In addition to above extensions, we have included the following heuristics to improve the quality of the inferred annotations.

Monitor Variables. Objects serving as monitors for **synchronized** blocks are accessed concurrently by design. Each thread using the object for synchronization probes whether it is available before entering the critical section. Inside the method scope, this probing generates a witness of interleaved access to the monitor object. However, marking monitor objects as non-atomic likely contradicts the programmer's intention of guaranteeing atomicity. For example, access to the queue field in *weblech*'s Spider class is guarded by a **synchronized** block that uses the queue as monitor. Without the heuristic, queue would be considered non-atomic, which is clearly not the programmer's intention.

To better approximate the intended semantics, we modify the inference algorithm such that it discards witnesses of interleaved access to fields *in the scope* in which the field is used as monitor. Witnesses obtained in other scopes count as before.

Shared Objects. The inference algorithm generates an annotation for every field appearing in the field access trace. This includes fields whose value only a single thread accesses, that is, fields whose value is *thread local*. However, such fields are irrelevant: because they are never shared, they are trivially atomic and do not require synchronization. Removing the thread local fields from the output ensures that all annotations are relevant for the concurrent behavior of the program and thus improves the quality of the annotations.

The identification of shared objects is implemented as part of the abstract machine's memory model. Besides recording the write timestamps, the memory model also identifies the shared (not thread local) objects. The *observedAccess* function that collects the observed name paths considers shared objects only.

The memory model identifies shared objects by tracking the threads that access an object. An object is considered shared if at least one thread accesses it twice, with at least one other thread accessing it in between. Defining shared objects as those accessed by multiple threads is insufficient because objects are often created by one thread, which then passes them to another thread for processing. For example, the main thread of a traveling salesperson solver may create a route description and then ask a worker thread to evaluate it. Likewise, thread objects are first accessed by the parent thread during creation, and then by themselves during execution. Requiring alternating access as described above avoids including these cases.

Constructors. During object creation, the constructor of an object typically accesses a large portion—often all—fields of the object. Usually, this happens atomically because few constructors pass the local instance reference (**this**) to a concurrent thread. Thus, a typical constructor atomically accesses most of the fields in an object. Treating constructors as instance methods, the inference algorithm derives from this access pattern that most of the object's fields belong to the same atomic set, which limits the potential for concurrency under the inferred synchronization annotations.

As a workaround, the inference tool treats observations from constructors as external. The atomic sets suggested by constructors are therefore external and do not trigger the merging of internal atomic sets for the class. Instead, the constructor-suggested atomic sets are used to extend the internal atomic sets as described in section 6.

8 Evaluation

This section discusses the performance of our algorithm measured by the quality of the inferred annotations. After detailing the experimental setup, we summarize the results and gained insights.

8.1 Program Corpus

Table 1 lists the programs used to evaluate the inference algorithm. The list contains all Java programs for which an AJ version is publicly available, except

cewolf. The *cewolf* library was excluded because the AJ annotations concern only a very minor fraction of the code. For every program except *collections*, the corpus also includes the compiled AJ version. Both versions are used in the evaluation. These Java programs were manually converted to AJ by Dolby et al. [6]; archives containing their source are available on the *Data-Centric Concurrency Control* project website[2]. Sole exception is the AJ variant of the Java collections framework, which was kindly provided by Frank Tip.

In the table, the *kLoC* column lists the number of thousand lines of source code in the Java version of the program, excluding comments and empty lines. The *Classes* column shows the number of classes in the program. The number of classes that contain AJ annotations in the AJ version are given in the *Annotated* column.

Table 1. *Corpus of programs used to evaluate the inference algorithm.* In addition to the Java version using control-based synchronization, each program except *collections* is available as a compiled AJ version using data-centric synchronization. Both versions are used in the evaluation. The *kLoC* column lists the number of thousand lines of source code in the Java version of the program, excluding comments and empty lines. The *Classes* column shows the number of classes in the program. The number of classes that contain AJ annotations in the AJ version are given in the *Annotated* column.

Program	*Description*	*kLoC*	*Classes*	*Annotated*
collections	OpenJDK 1.6 collections framework	11.1	171	43
elevator	Elevator simulation	0.3	6	2
jcurzez1	Console window library (low concurrency)	2.7	78	9
jcurzez2	Console window library (high concurrency)	2.8	79	6
tsp2	Solver for the traveling salesman problem	0.5	6	2
weblech	Web site mirror tool	1.3	12	2

8.2 Method

Each program in the corpus is first instrumented and then run three times using the same workload. For *elevator* and *tsp2*, the workload consists of example input files distributed with the programs; *weblech* is used to aggregate files from a local web server; and the *collections* and *jcurzez* libraries are used for random operations by a custom fuzzing program. The workloads were set large enough to trigger the use of multiple operating system threads by the JVM in order to obtain traces with fine-grained interleavings. Comparing the annotations inferred for three separate runs gives us insight into the effects of (random) thread scheduling and allows us to verify that the annotations likely reflect consistent program behavior. We remove spurious observations and consolidate the annotations from all runs into a single set of annotations.

[2] `http://sss.cs.purdue.edu/projects/aj/`

Next, we compare these inferred annotations against the ones Dolby et al. manually inserted when converting the program to AJ. For every difference, we investigate and record whether it results in disparate program behavior by analyzing the source code of both variants. Furthermore, we discuss the root cause that led to inferring a differing annotation. See section A for a detailed discussion of each program.

For each program, we count the number of classes with differing annotations. Grouped by atomic set, alias, and unit of work declaration, we track whether annotations are missing from a class, and whether the class contains additional annotations. Missing annotations (*Missed* columns in Tables 2, 3, and 4) are those that were manually added, but not inferred. Added annotations (*Added* columns in Tables 2, 3, and 4) are those that were inferred, but not manually added. A single class can contribute to both counts in each group. To obtain better insight into the algorithm's overall behavior, we categorize each counted class by the root cause. These causes are:

- Algorithm deficiencies—actual problems with our approach.
- Bugs in the manual annotations.
- Structure differences (refactorings) between the original and the AJ version.
- Tool deficiencies like ignorance of **final** attributes for fields, which can be fixed in a better implementations.
- Workload insufficiencies that omit relevant behavior, which prevents observation by the tool.

We follow this subjective *qualitative* approach for two reasons. First, the goal of our algorithm is to infer annotations that not only enforce, but also document the intended concurrency structure of the program. Evaluating how well the inferred annotations meet this goal requires human inspection of the program. Simple quantification of the differences between manual and inferred annotations alone—for example their number or size—does not convey meaningful information because most AJ versions have been refactored and structurally differ from the Java versions; furthermore, some of the manual annotations are incomplete or even wrong. Second, using other quantitative measures like execution speed is unfeasible because the prototype AJ compiler is currently defunct.

The refactorings in the AJ variants were executed to meet the requirements of AJ, to work around limitations of the used implementation, and to simplify the conversion. Refactorings to meet requirements include introducing getter and setter methods for fields. Workaround refactorings include removing the nesting of classes and splitting classes to achieve concurrent execution. Simplification refactorings include dropping specialized iterator classes in the *collections* framework.

We do not report the processing times because in a source code conversion workflow, it suffices to execute the tool once. Instrumentation of all programs finished within seconds; generating the traces and inferring the annotations took less than 25 minutes for each program on an Intel Core i7 processor with 2 GB of RAM.

Table 2. *Number of classes with differing atomic set annotations.* The "*-AJ*" rows show the number of differing classes for the annotations inferred for the AJ variant of the program that was ported by Dolby et al. The *Missed* columns show for how many classes the inferred annotations miss a manual annotation; the *Added* columns count the classes with additional annotations. The differences are categorized by their cause: Algorithm deficiency (*A*); Bug in the manual annotations (*B*); Structural difference from refactorings (*S*); Tool implementation limitation (*T*); and workload insufficiency (*W*). A dot "·" denotes a zero.

Atomic Sets	*Missed*					*Added*				
Program	*A*	*B*	*S*	*T*	*W*	*A*	*B*	*S*	*T*	*W*
collections	·	1	4	5	6	·	·	·	6	·
elevator	·	·	·	·	·	3	·	1	·	·
elevator-AJ	·	·	·	·	·	3	·	1	2	·
jcurzez1	·	·	1	·	·	8	1	·	1	·
jcurzez1-AJ	·	·	1	·	·	7	1	·	1	·
jcurzez2	·	·	2	·	·	6	1	·	2	1
jcurzez2-AJ	·	·	2	·	·	6	1	·	2	1
tsp2	·	·	·	·	·	1	·	1	2	·
tsp2-AJ	·	·	·	·	·	1	·	1	2	·
weblech	1	·	·	·	·	3	1	1	·	·
weblech-AJ	1	·	·	·	·	5	1	1	·	·

Table 3. *Number of classes with differing alias annotations.* The rows and columns have the same meaning as in Table 2.

Aliases	*Missed*					*Added*				
Program	*A*	*B*	*S*	*T*	*W*	*A*	*B*	*S*	*T*	*W*
collections	1	·	·	4	3	·	·	·	·	·
elevator	·	·	·	·	1	2	·	·	·	·
elevator-AJ	·	·	·	·	1	·	·	·	·	·
jcurzez1	·	1	·	·	·	4	·	·	·	·
jcurzez1-AJ	·	1	·	1	·	3	·	·	·	·
jcurzez2	·	1	·	·	·	4	·	·	·	·
jcurzez2-AJ	·	2	·	·	·	4	·	·	·	·
tsp2	·	·	·	·	·	1	1	·	·	·
tsp2-AJ	·	·	·	·	·	1	1	·	·	·
weblech	·	·	·	·	·	3	·	·	·	·
weblech-AJ	·	·	·	·	·	4	·	·	·	·

Table 4. *Number of classes with differing unit of work declarations.* The rows and columns have the same meaning as in Table 2.

Units of Work	*Missed*					*Added*				
Program	*A*	*B*	*S*	*T*	*W*	*A*	*B*	*S*	*T*	*W*
collections	·	·	6	5	4	·	·	1	·	·
elevator	1	·	·	·	·	·	·	·	·	·
elevator-AJ	1	·	·	·	·	·	·	·	·	·
jcurzez1	·	3	2	1	2	1	·	4	·	·
jcurzez1-AJ	·	1	2	1	1	1	·	4	·	·
jcurzez2	·	3	1	·	1	2	·	4	·	·
jcurzez2-AJ	·	2	1	·	2	2	·	4	·	·
tsp2	·	·	·	·	1	·	·	·	·	·
tsp2-AJ	·	·	·	·	·	·	·	·	·	·
weblech	·	·	·	·	2	·	·	·	·	·
weblech-AJ	·	·	·	·	2	·	·	·	·	·

8.3 Results and Discussion

Table 2 shows the number of classes with differing atomic set annotations. A missing atomic set is the most critical kind of difference because it entails that some fields that were intended to be protected from interleaved access remain unprotected, which may result in race conditions. Except for the Java *collections* framework and *weblech*, all missing atomic sets have secondary causes that either reflect refactorings, or can be fixed through a better implementation or workload. In particular, the fuzzer used to drive the *collections* proves to be incomplete. For example, it does not perform concurrent operations on iterator objects. The *shared objects* heuristic described in section 7 therefore removes annotations for their fields, which results in a missing atomic set compared to the manual annotations. In *collections*, our algorithm infers that the static and final field PRESENT in the class HashSet should *not* be a member of an atomic set. This is correct and cannot lead to race conditions. In contrast, the manual annotations accidentally introduce global synchronization between all HashSet instances through the atomic set membership of the field. The missing atomic set count in *weblech* concerns the omission of a field for unknown reasons. This could be a bug in our implementation. However, because we were unable to rule out an algorithm deficiency, we categorize it as this most severe difference type.

The *Added* half of Table 2 indicates that our algorithm infers highly detailed annotations. While additional locking can lead to deadlock, this is not a severe problem because it can be recognized statically [15]. To the contrary, the higher degree of detail prevents a number of inadvertent race conditions in *jcurzez* and *weblech*.

The major factor driving the suggestion of additional annotations is the documentation of *obvious* behavior; another factor is the algorithm's ignorance of the AJ implementation's limitations. Obvious behavior concerns high-level un-

derstanding of a program's concurrency structure. Developers use this understanding to avoid annotating classes they deem irrelevant for achieving the intended behavior. The inference algorithm lacks this concept of obviousness and generates annotations for all classes. From the perspective of project-external developers, these annotations provide a guard against accidentally violating behavior invariants, while at the same time documenting these invariants. For example, in *tsp2*, the PrioQElement elements of a priority queue are accessible only from within the TourCreator singleton. Because TourCreator synchronizes access to the priority queue, no annotations are necessary for the queue elements. In contrast, the algorithm suggests an atomic set containing all fields of PrioQElement, as well as an alias from the respective field in TourCreator. The suggested annotations document the concurrency structure of the program. They are neutral to performance because the alias unifies the atomic sets of the two classes and thus avoids introducing synchronization overhead when the queue is accessed. Other examples include detailed array aliases in *elevator*, as well as unit of work declarations for printing methods in the variants of *jcurzez*.

Tables 3 and 4 show the number of classes with differing aliases and unit of work declarations. Missing aliases only result in additional synchronization overhead and do not affect the program's correctness. Missing unit of work declarations can lead to race conditions. However, all of these differences have secondary causes, or highlight bugs in the manual annotations (discussed below). Additional aliases and unit of work declarations reduce the potential for concurrency in the program, but cannot lead to errors like race conditions or deadlocks. The high number of additional aliases documents the importance of choosing a workload that exerts as much concurrency in the program as possible.

Behavior. The effects of the annotations inferred for *collections*, the *jcurzez* variants, and *tsp2* match the effects of the manual annotations. For *elevator* and *weblech*, the behavior differs: using the suggested annotations with the current lock-based implementation of atomic sets effectively imposes a global lock which removes all concurrency from the program. In both cases, the over-restrictive effects come from our algorithm's inability to change the class structure of the program.

In *elevator*, the different threads synchronize using the elements of a globally shared array as monitors. No interleaved access occurs for a single array element, and thus the algorithm includes the array and all its elements in a single atomic set. Within its limitations, this is the correct behavior because excluding the elements would allow data races. Regaining concurrent execution would require to split the array or a similar refactoring. The developers of AJ use a generalized **unitfor** annotation to circumvent global locking. However, while achieving the right effects in the AJ implementation, their annotation is illegal according to the typing rules because it contains non-final segments in its atomic set designator.

Like *elevator*, the annotations inferred for *weblech* impose a global lock: the download threads in *weblech* execute a single shared Runnable object; adding an atomic set to this object effectively prevents any concurrent execution, that

is, downloads. A solution to this problem is to split the Runnable object. The manually ported AJ variant of *weblech* follows this approach, but the refactoring leaves the crucial blocking network access inside a unit of work for the Runnable's atomic set, and thereby still prevents concurrent downloading.

The inferred annotations for *jcurzez1* reveal race conditions in the classes Cell, Cursor, and Pen. In *jcurzez1-AJ*, the racing fields of Cursor and Pen are protected by an atomic set. Since both the library's control- and data-centric synchronization was added by the AJ developers, this documents that the race is unintended, which points at the difficulty of defining control-centric synchronization. The malign race in class Cell and the lack of a manual atomic set definition for this class are proof that understanding the concurrency structure of other people's programs is hard, which supports our case for automating the necessary reasoning. The case of the PRESENT field in the class HashSet of *collections* yields additional support.

Inferring annotations for the manually ported AJ variants of the programs yielded results similar to the those of the original variants. This indicates that the manual annotations capture the original program's behavior to a high degree. It also shows that the manual refactorings influence the inference very little and cannot be used to guide the derived annotations.

9 Related Work

The automatic inference of a program's concurrency structure has been treated in the context of data race detection. There, the structure is used to warn about violations of the likely *intended* atomicity semantics of variables.

A dynamic approach that learns the atomicity intentions for shared variables from execution traces is the *AVIO* system of Lu et al. [14,13]. *AVIO* observes the read and write operations on a shared variable and treats it as atomic if all operations were serializable. Observing each variable in isolation, *AVIO* can only detect low-level data races. In contrast, Artho et al. [1] introduce the notion of high-level data races and explicitly design their dynamic algorithm to consider races on sets of semantically related variables. Both methods are similar to our algorithm in that they work without user annotations. The *AssetFuzzer* algorithm of Lai et al. [11] likewise works without annotations. It additionally uses partial order relaxation to detect potential, but unmanifested, violations in the execution trace. The *Atomizer* system of Flanagan and Freund [7] also considers *windows of vulnerability*, but requires a few source code annotations.

The *MUVI* tool of Lu et al. [12] follows a static approach to inferring atomicity intentions by mining the program source code and computing variable correlations. The static heuristic [9,18] of defining one atomic set per class that contains all non-static fields has also been proposed in the context of race detection. Targeting race detection, none of the aforementioned approaches considers aliasing information, which is essential for our use case.

Huang and Milanova propose a static inference system for AJ types that significantly reduces the number of annotations that a developer has to write [10].

While simplifying the use of AJ, it needs a set of foundational annotations. Hence, their and our methods complement each other: the static inference rules propagate the base annotations inferred by our analysis, yielding a complete set of AJ annotations.

Atomic sets take a declarative approach to synchronization. *Synchronizers* [8,4] provide a similar notion in the context of Actor systems, where they constrain the message dispatch in a group of Actors. The available constraints differ from atomic sets in that *atomicity constraints* provide *temporal* atomicity—messages arrive at the same time—, not the *spatial* atomicity offerred by atomic sets. Furthermore, Synchronizers cannot easily express the non-interleaving of message sequences, which is the Actor equivalent of non-interleaved access to shared data, and do not support transitive extension similar to aliases in atomic sets.

The algorithm introduced in this paper follows the approach of *accentuating the positive* [20,13]: by assuming atomicity for all operations unless witnessing interleaved access, it suppresses rarely observed Heisenbugs. While leading to coarser concurrency structure, the experiments of Weeratunge et al. [20] show that a low runtime overhead of 15 % can be achieved using this method.

10 Conclusions

This paper introduces an algorithm that infers annotations for data-centric synchronization based on atomic sets (section 2). The algorithm automates the reasoning about a program's concurrency structure that is necessary for converting the program from control-centric synchronization, for example using locks, to synchronization using atomic sets. Thus, it marginalizes the effort necessary to have existing programs benefit from the improved robustness against concurrency bugs of data-centric synchronization.

The fundamental idea behind the algorithm is that the methods of a program perform semantically meaningful actions. The fields a method accesses atomically therefore suggest an atomic set for which the method should be a unit of work (section 3). The algorithm records the dynamic behavior of a program (section 4), extracts the field access patterns from the trace (section 5), and infers the annotations by aggregating the patterns (section 6). A version for Java programs has been implemented (section 7) with support for arrays, synchronized blocks, and wait–notify synchronization.

Discussion and Future Work

The algorithm we presented infers atomic set annotations from the execution traces of a program; if these are not available they have to be generated by executing the program. In particular, converting isolated modules of a large code base requires unit tests which execute these modules. The algorithm further assumes that all observed execution traces are correct, that is, reflect programmer

intent. This assumption can hold even if a program contains bugs: schedule-dependent *Heisenbugs* that never (or very rarely) appear during testing will likely not be observed. In this case, the inferred annotations will prevent the bugs in future executions.

The degree of concurrency in a program with inferred annotations depends on the concurrency manifest in the execution traces that are used. It is therefore important to collect traces using workloads that trigger as much correct concurrent behavior as possible. A direction for future work is to automate the generation of workloads, for example using concolic execution to explore thread scheduling [17].

Another factor limiting concurrency is the current lock-based implementation of atomic sets. Our algorithm treats read–read sharing of fields as non-interleaved access and therefore includes these fields in atomic sets. In the converted program, the fields are therefore protected by locks, preventing the concurrent reading observed in the execution traces. Although unnecessarily restrictive, the resulting behavior will be correct. Better implementation of atomic sets, for example, by using software transactional memory, or by inferring advanced annotations such as *fastread*, partial *unitfor*, and *internal*, would improve the degree of concurrency. As demonstrated by Dolby et al. [6], these annotations may have a dramatic effect on a program's performance.

Finally, our inference is based on simple set-membership and ignores how often and how far from the current scope the events in the set occurred. This makes the inference brittle. A probabilistic reasoning method, for example using Bayesian networks, would be more robust against noise from *Heisenbugs* and could thus allow relaxed synchronization during trace generation. Probabilistic inference could also help overcome the alias resolution problem described in section 6.

Acknowledgments

This publication was made possible in part by sponsorships from the Army Research Office under award number W911NF-09-1-0273, as well as the Air Force Research Laboratory and the Air Force Office of Scientific Research under agreement number FA8750-11-2-0084. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright notation thereon.

References

1. Cyrille Artho, Klaus Havelund, and Armin Biere. High-level data races. In Pedro T. Isaías, Florence Sedes, Juan Carlos Augusto, and Ulrich Ultes-Nitsche, editors, *NDDL/VVEIS*, pages 82–93. ICEIS Press, 2003.
2. Michael Burrows and K. Rustan M. Leino. Finding stale-value errors in concurrent programs. *Concurrency - Practice and Experience*, 16(12):1161–1172, 2004.

3. Luis Ceze, Christoph von Praun, Calin Cascaval, Pablo Montesinos, and Josep Torrellas. Concurrency control with data coloring. In Emery D. Berger and Brad Chen, editors, *MSPC*, pages 6–10. ACM, 2008.
4. Peter Dinges and Gul Agha. Scoped synchronization constraints for large scale actor systems. In Marjan Sirjani, editor, *COORDINATION*, volume 7274 of *Lecture Notes in Computer Science*, pages 89–103. Springer, 2012.
5. Julian Dolby, Stephen J. Fink, and Manu Sridharan. T. J. Watson libraries for analysis (WALA). `http://wala.sf.net`.
6. Julian Dolby, Christian Hammer, Daniel Marino, Frank Tip, Mandana Vaziri, and Jan Vitek. A data-centric approach to synchronization. *ACM Trans. Program. Lang. Syst.*, 34(1):4, 2012.
7. Cormac Flanagan and Stephen N. Freund. Atomizer: A dynamic atomicity checker for multithreaded programs. *Sci. Comput. Program.*, 71(2):89–109, 2008.
8. Svend Frølund and Gul Agha. A language framework for multi-object coordination. In Oscar Nierstrasz, editor, *ECOOP*, volume 707 of *Lecture Notes in Computer Science*, pages 346–360. Springer, 1993.
9. Christian Hammer, Julian Dolby, Mandana Vaziri, and Frank Tip. Dynamic detection of atomic-set-serializability violations. In Wilhelm Schäfer, Matthew B. Dwyer, and Volker Gruhn, editors, *ICSE*, pages 231–240. ACM, 2008.
10. Wei Huang and Ana Milanova. Inferring AJ types for concurrent libraries. In *FOOL 2012: 19th International Workshop on Foundations of Object-Oriented Languages*, pages 82–88, 2012.
11. Zhifeng Lai, Shing-Chi Cheung, and Wing Kwong Chan. Detecting atomic-set serializability violations in multithreaded programs through active randomized testing. In Jeff Kramer, Judith Bishop, Premkumar T. Devanbu, and Sebastián Uchitel, editors, *ICSE (1)*, pages 235–244. ACM, 2010.
12. Shan Lu, Soyeon Park, Chongfeng Hu, Xiao Ma, Weihang Jiang, Zhenmin Li, Raluca A. Popa, and Yuanyuan Zhou. MUVI: automatically inferring multi-variable access correlations and detecting related semantic and concurrency bugs. In Thomas C. Bressoud and M. Frans Kaashoek, editors, *SOSP*, pages 103–116. ACM, 2007.
13. Shan Lu, Soyeon Park, and Yuanyuan Zhou. Detecting concurrency bugs from the perspectives of synchronization intentions. *IEEE Trans. Parallel Distrib. Syst.*, 23(6):1060–1072, 2012.
14. Shan Lu, Joseph Tucek, Feng Qin, and Yuanyuan Zhou. AVIO: Detecting atomicity violations via access-interleaving invariants. *IEEE Micro*, 27(1):26–35, 2007.
15. Daniel Marino, Christian Hammer, Julian Dolby, Mandana Vaziri, Frank Tip, and Jan Vitek. Detecting deadlock in programs with data-centric synchronization. Research Report RC25300 (WAT1208-051), IBM, 2012.
16. J. Gregory Morrisett and Simon L. Peyton Jones, editors. *Proceedings of the 33rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2006, Charleston, South Carolina, USA, January 11-13, 2006*. ACM, 2006.
17. Koushik Sen and Gul Agha. A race-detection and flipping algorithm for automated testing of multi-threaded programs. In Eyal Bin, Avi Ziv, and Shmuel Ur, editors, *Haifa Verification Conference*, volume 4383 of *Lecture Notes in Computer Science*, pages 166–182. Springer, 2006.
18. William N. Sumner, Christian Hammer, and Julian Dolby. Marathon: Detecting atomic-set serializability violations with conflict graphs. In Sarfraz Khurshid and Koushik Sen, editors, *RV*, volume 7186 of *Lecture Notes in Computer Science*, pages 161–176. Springer, 2011.

19. Mandana Vaziri, Frank Tip, and Julian Dolby. Associating synchronization constraints with data in an object-oriented language. In Morrisett and Jones [16], pages 334–345.
20. Dasarath Weeratunge, Xiangyu Zhang, and Suresh Jagannathan. Accentuating the positive: atomicity inference and enforcement using correct executions. In Cristina Videira Lopes and Kathleen Fisher, editors, *OOPSLA*, pages 19–34. ACM, 2011.

A Annotation Comparisons

This section discusses in detail the differences between the manually added and the inferred atomic set, alias, and unit of work annotations for each program used in the evaluation (section 8). The discussion is highly specific to details of the compared programs. It should be read with the source code available for reference.

The margin notes next to the discussion of each class describe how the annotation differences of this class were counted when compiling Tables 2, 3, and 4. A leading "+" denotes an *additional* annotation (inferred, but not manually added); a leading "−" denotes a *missing* annotation (manually added, but not inferred). The subsequent letter, A, L, or U, describes the type of annotation: atomic set, alias, or unit of work declaration. In parentheses follows the category of the cause:

- Algorithm deficiencies—actual problems with our approach (A).
- Bugs in the manual annotations (B).
- Structural differences (that is, refactorings) between the original and the AJ version (S).
- Tool deficiencies like ignorance of **final** attributes for fields, which can be fixed in a better implementations (T).
- Workload insufficiencies that omit relevant behavior, which prevents observation by the tool (W).

A.1 Java Collections Framework

Workload. The execution traces used as input for the inference tool were generated by driving the collections with a custom fuzzing tool. The fuzzing tool exercises random operation like element insertion and deletion on shared collection objects. Its source code is included in the tool chain's source distribution.

Runs. Despite fixing the (pseudo-) random number generator seeds of the fuzzer, the non-deterministic thread scheduling leads to operations being executed in different interleavings. This causes the random numbers (whose sequence is fixed by the seed) to be used in different contexts: fuzzing operations iterating over the elements in the collection vary between the runs as the collection contents depend on the thread schedule. Fixing the random seed therefore does

not lead to identical operation sequences. The differences between the annotations inferred for each of the three runs that originate in schedule-randomization are as follows:

- The atomic set inferred for class HashMap$KeySet in run 3 additionally aliases the this$0 field with the atomic set containing field size.
- The array alias of the field table in class LinkedHashMap of runs 1 and 3 not only covers the array elements (as in run 2), but also includes the atomic set containing the field after.
- The atomic set inferred from run 1 for the iterator class LinkedList$ListItr includes the lastReturned field with an alias to the entry's atomic set, which is not present in run 2. Furthermore, the alias for the reference to the containing object, this$0, additionally includes the atomic set containing the header field.
- The field m in the atomic set of TreeMap$KeySet of run 3 is not aliased with the atomic set containing field modCount.
- In the atomic set of class TreeMap$PrivateEntryIterator, the alias of the field lastReturned includes the field color in runs 1 and 3. Similarly, the color field is added in the unit of work declaration for parameter 1 in method access$0. Likely, no rotation of the tree was necessary in run 2.
- The unit of work declaration of method deleteEntry() in TreeMap contains more aliases in runs 1 and 3 than in run 2, and the method fixAfterDelection() is declared a unit of work in run 1. Likely, a complex deletion was executed in run 1 but not in run 2.

The following witnesses of interleaved access to fields originate in iterator operations. Iterators must be manually synchronized by the user (see the comment in method iterator() of class Collections$SynchronizedCollection in Collections.java). The fuzzing tool does not do this, and hence spurious race conditions occur. We discard these races as having a known origin that could be fixed through synchronization in the fuzzing tool, and instead use data from a run without the witness.

- Run 3 contains a witness for the field value in the HashMap$Entry class (via the setValue() method).
- The field root of class TreeMap is witnessed as being non-atomic in run 1 (in method getCeilingEntry() via an iterator). Runs 2 and 3 do not contain the witness.

The following differences between the runs concern classes that have no manually annotated counterparts in *collections-AJ*. They can be safely ignored.

- Run 2 does not attribute the m field from Collections$SynchronizedMap to AbstractMap; runs 1 and 3 do this.
- The atomic set inferred for Hashtable$EntrySet additionally aliases the field this$0 (the reference to the containing instance) with the atomic set containing count and modCount in run 2. In run 1, the fields are missing because they were witnessed as being non-atomic in the class's method iterator().

– The field count in Hashtable was witnessed as being non-atomic in run 1 in the method getIterator().
– In the atomic set of IdentityHashMap$EntrySet in run 2, modCount is missing from the aliases of field this$0.
– The atomic set inferred for class Vector of run 3 is missing the capacityIncrement field compared to the other runs.
– The field size is not included in the atomic set aliased by the field this$0 in the atomic set of class WeakHashMap$EntrySet.

Furthermore, differences concerning compiler generated methods are ignored.

– The access$1() method in ArrayList is not a unit of work for the atomic set containing size in run 2. This method is added by the compiler to give nested classes access to the fields of their containing class. The difference thus does not matter.
– The compiler-generated method access$2() in LinkedList is only declared an external unit of work in run 1, not in run 2. Apparently, it was not executed in run 2.

Annotation Comparison Between *collections* and *collections-AJ*. The AJ-annotated version of the Java collections framework is called *JUtil_All*, but will be henceforth referred to as *collections-AJ*.

The tool follows the Java compiler's decision about the containing class for fields. Thus, if the bytecode contains a get statement for field f in class c, it considers f as belonging to c, regardless of the fact that f was declared in a superclass of c. This sometimes leads to false attribution, in particular with abstract classes. For example, the fields keySet and values of the abstract class AbstractMap are missing in the tool's output for this class. They are, however, present in the concrete subclass HashMap.

The collections code has been refactored to obtain *collections-AJ*. For example, the list iterator handling has been completely removed from the concrete subclasses. Instead, the subclasses rely on the generic iterators AbstractList$Itr, AbstracList$ListItr, and so on that are defined in AbstractList. No inferred annotations are available for these classes because the concrete classes used to generate the field access traces provide their own optimized versions like ArrayList$Itr. Consequently, the classes are invisible to the inference tool. We instead compare the inferred annotations of the specialized classes against the generic implementations.

The same is true for interfaces like Collection. In *collections-AJ*, some of the defined methods are declared to be units of work for the atomic sets of their arguments. Because the methods have no implementation, they are hidden from the inference tool. However, the tools infers the correct annotations for the interface method implementations. A unification post-processing step in the inference tool could reconciliate these results and derive the most specific unit of work definition for the interface and abstract class definitions. Similarly, the alias type annotations of method parameters (for example setFoo(Foo other/∗this.F=F∗/)) can be derived by unification from the inferred field types.

The abstract classes and interfaces that are missing in the inference tool output, but serve as reference for the annotations of concrete subclasses, are:

- AbstractList
- AbstractSequentialList
- Collection
- List
- Map
- Set

The copy constructors of classes are not explicitly fuzzed. Inferred annotations for evaluation are available only where the constructors have been invoked by other methods.

$-U\ (T)$

AbstractCollection: The atomic sets and the units of work definitions are identical. An atomic set is neither inferred, nor manually added. The **unitfor** annotations on the bulk operations (like addAll()) match.

AbstractList: The algorithm does not infer the unit of work annotation for the argument to the addAll() method because all the concrete subclasses used in the evaluation re-implement the method. Consequently, the method is never called in any run and thus invisible to the inference tool. The manual and inferred atomic sets match; both consist of the modCount field

$-A\ (T)$
$-L\ (W)$

AbstractMap: The defined units of work are identical. The algorithm fails to infer an atomic set for this class because the fields are attributed to the concrete subclasses HashMap and TreeMap. However, the atomic sets inferred for these classes include the fields. In the case of TreeMap, the field keySet does not occur in the inferred annotations because the implementation instead uses a field navigableKeySet. The inferred aliasing does not match the manual annotations in AbstractMap because the fuzzing tool performs no operations on the respective objects.

$4 \times -A\ (T)$
$4 \times -L\ (T)$
$4 \times -U\ (T)$

The class AbstractMap_1 in *collections-AJ* is the key set belonging to an AbstractMap; class AbstractMap_2 models the respective iterator. The classes AbstractMap_3 and AbstractMap_4 do the same, but for the map's values. Because the all concrete classes used in the evaluation re-implement the respective methods and classes (for example, HashMap$EntrySet), the tool does not infer any annotations for these classes in collections. The structure of these classes, however, is equivalent to the concrete implementations. If the classes were used during execution, the inferred annotations would therefore likely match the manual annotations.

$-U\ (T)$

AbstractSequentialList: The unit of work annotation for addAll() is missing because the method is never called. (Its only concrete subclass used in the evaluation is LinkedList, and this class overrides the method.)

AbstractSet: Neither set of annotations defines an atomic set for the class. Both contain identical unit of work annotations.

$-A\ (S)$

ArrayList$Itr: Compared to the atomic set of AbstractList_Itr in *collections-AJ*, the atomic set inferred for the class ArrayList$Itr is missing all fields local to the iterator object. The reason for this is that the fuzzing tool does not share

iterators between the fuzzing threads. Consequently, the fields are purged from the inferred atomic set. Fuzzing iterators from both threads would fix this omission.

The only field present in the inferred atomic set is the implicit reference field this$0 to the containing ArrayList instance. The alias inferred for this field matches the alias manually defined for the (explicit) reference field abstractList to the containing list in AbstractList_Itr of *collections-AJ*.

ArrayList: Modulo the false attribution of the modCount field to the class, the manually added and the inferred atomic sets are identical. This includes the alias adding the elements of the elementData array to the atomic set. $+A\ (T)$ $+U\ (A)$

The list of inferred units of work contains both addAll() variants, which have manual annotations. The other methods were either removed in *collections-AJ* (like removeAll()), were generated by the compiler (like access$1()), or are private (like batchRemove()). Simple filtering of the units of work output by the inference tool could remove the last two categories and improve the quality of the inferred annotations.

HashMap$Entry: In *collections-AJ*, this class has been moved out of HashMap and is called HashMap_Entry. The inferred atomic set is identical to the manually defined one. The alias annotations to the types of the getNext() and setNext() methods can be inferred statically from the type of the next field (for which the right aliases are inferred).

HashMap$EntryIterator: Neither atomic sets nor units of work are manually defined, or inferred for this class (called HashMap_EntryIterator in *collections-AJ*). The manual alias-type annotations to the constructor argument and the next() return type can be inferred from the types in the super class HashMap$HashIterator. (See below.)

HashMap$EntrySet: In *collections-AJ*, the factored out counterpart—the class HashMap_EntrySet—contains an atomic set including the class's only field hashMap. This field is the explicit reference to the containing HashMap object, which aliases the HashMap's atomic set. The inferred atomic set matches this definition modulo the reference field this$0 being implicit. $-A\ (S)$

HashMap$HashIterator The inferred atomic set is missing the fields index and expectedModCount, which are included in the manually defined atomic set. This is due to the non-sharing of iterators. The remaining fields, however, are included in the inferred atomic set; their aliases match the manually defined aliases. $-A\ (W)$

HashMap$KeyIterator: Equivalent to HashMap$EntryIterator.

HashMap$KeySet: The only field in this nested class is the implicit reference to the containing object. As in the corresponding class, HashMap_KeySet, in *collections-AJ*, the reference is included in the atomic set of the class; the defined aliases match. $-A\ (S)$

HashMap$Values: This class is apparently never used during any of the three runs. It does not appear in the inference tool's output. $-U\ (W)$

HashMap: The loadFactor field is missing from the inferred atomic set. All other fields, including the alias definitions, are identical. The inference tool omits the loadFactor field because the only non-constructor method in the class that $-A\ (W)$

accesses the field is putAll(), and this method is used sparingly by the fuzzing tool. (It is guarded by the allowed flag.) The field is present in the inferred atomic set for Hashtable.

$+A$ (T)
$-L$ (A) In the inferred annotations, the fields keySet and values are wrongly attributed to this class. They instead belong to the AbstractMap class. The manual annotations put the fields in the same atomic set (potentially because of the limitation to one atomic set per class). The defined aliases are not inferred.

$-U$ (W) The package private method putAllForCreate() is not inferred as a unit of work for the atomic set of its argument because the fuzzer does not invoke clone() or the copy constructor of HashMap (which are the only callers of putAllForCreate()).

$-A$ (B) HashSet: The inferred atomic set omits the inclusion of the PRESENT field and otherwise matches the manual specification. The inferred atomic set is nevertheless correct because the field is declared to be final and *static*. In contrast, including the PRESENT field in the atomic set for objects severely limits the concurrent use of HashSet objects because every thread using such an object must acquire the (implied global) lock that protects the field. Consequently, no two threads can execute code within HashSet objects at the same time.

$-U$ (W) No unit of work annotation is inferred for the copy constructor's argument because it is never invoked by the fuzzer.

$+A$ (T) LinkedHashMap$Entry: Modulo the wrongly attributed fields key and value, the inferred atomic set matches the manually defined atomic set, including the correct alias definitions.

$+U$ (A) The tool furthermore infers the methods addBefore() and recordAccess() to be external units of work for the atomic sets of their arguments. No such annotation was added manually. The definition has no effect on the program behavior because all calls to the methods of this private class come from contexts where the atomicity of the arguments is already ensured. However, the annotations clarify the invariant holding in the method without introducing locking overhead.

LinkedHashMap$EntryIterator: No atomic set or unit of work is inferred or defined manually for this class. The respective alias type annotations can be inferred from the aliases of the LinkedHashMap$LinkedHashIterator fields.

LinkedHashMap$KeyIterator: See LinkedHashMap$EntryIterator.

$-A$ (W) LinkedHashMap$LinkedHashIterator: The inferred atomic set contains only the implicit reference to the parent object and the lastReturned field. Both have aliases that match the manual annotations in the corresponding *collections-AJ* class LinkedHashMap_LinkedHashIterator. The nextEntry and expectedModcount fields are missing because the fuzzer does not access iterator objects from different threads. The header field was duplicated from LinkedHashMap in *collections-AJ* to avoid direct access (see the comment in the source code in LinkedHashMap_LinkedHashIterator.java).

$+A$ (T)
$+U$ (T) LinkedHashMap: Ignoring the wrongly attributed fields, the inferred and manually added atomic sets match. The only inferred units of work concern com-

piler generated functions and can be safely disregarded. No units of work were defined manually.

LinkedList$Entry: The manually added and the inferred annotations are identical.

LinkedList$ListItr: The inferred atomic set lacks several of the fields contained in the manually defined atomic set. The differences can be explained as follows: the header field has been duplicated in LinkedList_ListItr; fields in the iterator that do not reference shared objects (nextIndex and expectedModCount) are invisible to the inference algorithm. Disregarding the renamed field referring to the containing LinkedList object, the remaining fields and their alias definitions are identical. $-A\ (A)$

LinkedList: Both atomic sets are identical modulo the falsely attributed field modCount. The inferred units of work do not include the copy constructor, but cover the manually annotated variants of the addAll() method. The remaining inferred units of work concern either compiler generated or private methods and could be easily removed by an output filter in the inference tool. $+A\ (T)$ $-U\ (W)$ $+U\ (T)$

TreeMap$AscendingSubMap$AscendingEntrySetView: See TreeMap$AscendingSubMap.

TreeMap$AscendingSubMap: This class belongs to a part of the NavigableSet interface that is not implemented by TreeMap in *collections-AJ*.

TreeMap$Entry: The manually added and the inferred atomic sets are identical. The equals() method is not inferred to be a unit of work for its argument because the fuzzer does not invoke the method (only hashCode() is called). $-U\ (W)$

TreeMap$EntryIterator: Equivalent to LinkedHashMap$EntryIterator.

TreeMap$EntrySet: The corresponding class in *collections-AJ* is TreeMap_3. As in the manually defined atomic set, the inferred atomic set consists of the reference to the containing TreeMap object. The inferred alias for this field would yield the same results as the manual annotation. However, only two fields are listed for resolving the aliased atomic set because the fuzzing tool does not mutate the retrieved enty set, but only its entries (see method mutateEntrySet in class MapFuzzer). Improving the fuzzer would yield stronger support for the alias resolution. $-A\ (S)$ $-L\ (W)$

TreeMap$KeytIterator: Equivalent to LinkedHashMap$EntryIterator.

TreeMap$KeySet: The inferred annotations for this class reflect the introduction of, and delegation to, NavigableSet in Java 1.6. Since the corresponding TreeMap_1 class in *collections-AJ* belongs to an older version of Java, the annotations cannot be directly compared.

TreeMap$NavigableSubMap$SubMapEntryIterator: See TreeMap$AscendingSubMap.

TreeMap$NavigableSubMap$SubMapIterator: See TreeMap$AscendingSubMap.

TreeMap$NavigableSubMap: See TreeMap$AscendingSubMap.

TreeMap$PrivateEntryIterator: The inferred atomic set matches the manually defined atomic set except that the expectedModCount field is missing due to non-sharing of iterators in the fuzzing tool. No units of work are declared in either variant. $-A\ (W)$

TreeMap: The algorithm infers three atomic sets for this class. All fields contained in the manually added atomic set are present in the largest inferred atomic set. The only difference between the annotations is the missing aliasing of the field entrySet, which can be explained by the fuzzing tool's limited use of the entry set. (See the discussion of TreeMap$EntrySet above.) $+A\ (T)$ $-L\ (W)$

The two additional atomic sets are each singletons which consist of the navigableKeySet and values fields respectively. The first one belongs to the NavigableSet modifications of the collections framework that is not included in *collections-AJ*. It is thus safe to ignore in the comparison. The field values can be disregarded because it belongs to the class AbstractMap and was moved to TreeMap by the compiler.

$-U$ (W) The manually added definition of the method putAll() as a unit of work for its argument is also automatically inferred. However, the annotations defining the methods addAllForTreeSet() and buildFromSorted() as units of work are missing. Both methods are only invoked under special and rare circumstances (see method addAll() in TreeSet and, for example, method putAll() in TreeMap), which explains why they are not executed under random fuzzing. They are hence invisible to the inference tool. The remaining inferred units of work concern auxiliary private methods for tree operations. These annotations are correct, but superfluous.

$+U$ (A) Note that in *collections-AJ*, TreeMap_1 is the key set implementation of TreeMap (see method keySet()). TreeMap_2 is the value set implementation, and TreeMap_3 is the entry set.

The remaining classes are not present in *collections-AJ* and have no corresponding counterpart. Consequently, the atomic sets obtained for the following classes cannot be evaluated:

- Hashtable
- IdentityHashMap
- Vector
- WeakHashMap

A.2 Elevator

Workload. The traces were collected while running the program with 8 threads on the input file data3, which is included in its source distribution.

Runs. All three runs yield the same annotations.

Annotation Comparison Between *elevator* and *elevator-AJ*.

$-U$ (B) Controls: In the manually converted *elevator-AJ*, the Controls object contains no atomic sets. However, the methods operating on the floors[] array are declared to be external units of work for the atomic sets floors[onFloor].F of individual array elements (Floor objects). This declaration is a violation of the AJ type system because the generalized form of **unitfor** requires that the segments of the access path leading to the atomic set are final [6]. While the floors field itself could be annotated as final, Java lacks such an annotation for array *elements*.

$+A$ (A) The inferred atomic set A consists of the floors array with an alias including

its Floor elements, as well as the atomic set F of the elements: **this**.A[]F=**this**.A. This atomic set ensures the correct program behavior of atomic access to the Floor objects from the Lift threads. However, the coarse granularity effectively imposes a single global lock guarding all operations on the floors array. Since performance is not a consideration for the *elevator* program, this is not a problem.

Note that the algorithm is precise in its array element access handling. It infers atomic access for the Floor objects because the Lift threads always operate on *different* array elements. The inferred atomic set therefore correctly reflects the programmer intentions. Furthermore note that the Floor class contains *two* atomic sets, both of which should be aliased to the atomic set in Control. AJ currently does not support this, but, just like multiple **unitfor** declarations, there are no fundamental reasons for this limitation. As a workaround, the two atomic sets of Floor could be merged into a single atomic set. With all access being serialized by Control's atomic set, no concurrency would be lost through the merge.

Elevator: The algorithm infers a single atomic set containing the controls field, which holds the reference to the Controls singleton shared by all Lift threads and the Elevator main object. It furthermore adds an alias to the floors[] array. This alias to the *shared* array is the reason for adding the controls field. The Elevator object itself never uses this field in a concurrent setting. $+A\ (A)$ $+L\ (A)$

Floor: The algorithm infers two (disjoint) atomic sets. The first atomic set consists of the list of simulated people who wish to use the elevator to go up, and the flag signaling that an elevator is coming to take them upwards. The second atomic set is identical to the first, except for covering the downwards direction. Evidently, these inferred atomic sets represent semantic units.

Likely due to the AJ compiler limitation of supporting only one atomic set per class, the Floor class in *elevator-AJ* contains a single atomic set that consists of the two lists and the two flags. This atomic set furthermore aliases the contents of the lists of waiting people (using an AJ-converted variant of java.util.ArrayList). $+A\ (S)$

The alias is unnecessary for ensuring the correct program behavior because Floor structs are modified only by the Controls singleton c, which never passes them outside its scope, and never sets the Floor fields to shared objects. The atomic set inferred for the Controls singleton covering the whole floors[] array, including its Floor elements and their atomic sets, therefore guarantees atomicity. $-L\ (W)$

The correctness, however, depends on the particular use and ownership of the Floor objects by the Controls singleton c, as well as the global locking behavior of c. Race conditions could be accidentally introduced by changing Controls to use more fine-grained locking and to re-use lists for waiting people. The aliasing information in the manually added atomic sets prevents this scenario. The inference algorithm cannot infer a similar aliasing information because the used Vector containers belong to the Java runtime library, which is not instrumented.

Lift: Identical to Elevator. The aliasing of the floor array in Controls effectively $+A\ (A)$ $+L\ (A)$

imposes a single global lock that any Lift object must hold to make progress. Thus, in an AJ implementation using locks, all Lifts operate *sequentially*. An AJ implementation using transactional memory would not suffer from this restriction.

Runner: The class contains no fields. Consequently, both atomic sets are identically empty.

A.3 Elevator-AJ

Workload. The AJ variant of *elevator* uses the same workload as the original *elevator* program.

Runs. All three runs yield the same annotations.

Annotation Comparison between *elevator-AJ-Translated* and *elevator-AJ*. The inferred annotations differ from those of the *elevator* runs just by the lock fields inserted by the AJ compiler. In the Floor class, this overlap triggers the merging of the atomic sets.

$2 \times +A\ (T)$

Additionally, the addPeople() method and the copy constructor of the List class are made units of work for the (now instrumented) ArrayList that stores the people waiting to go either up or down. Since the overall structure of aliasing between the atomic sets of Control, Floor, Lift, and Elevator is the same as in *elevator*, this annotation is, again, unnecessary because all interleaving is prevented by the imposed global lock in Control (which is required by Lift via the aliasing of its controls field). However, the annotation is correct and adds documentation to the intended behavior that the list of people is accessed atomically.

$2 \times +L\ (A)$ removed

Because the program makes only very limited use of the collection classes, the inferred atomic sets for these classes are incomplete and are thus not considered as valid output. See subsection A.1 for a detailed comparison of the inferred annotations for the collections framework.

A.4 JCurzez1

Workload. A custom fuzzing tool drives the generation of execution traces. The fuzzer uses multiple threads to randomly execute operations on shared objects from the library. Its source code is contained in the tool chain distribution.

Runs. To check the access to buffer[], a print statement was inserted into the method _deleteLine() in class AbstractWindow, which apparently triggers an interleaved thread schedule by forcing the fuzzing threads to synchronize on the standard output stream.

The annotations inferred from runs 1, 2, and 3 differ only in their witnesses.

- Run 1 contains witnesses of interleaved access to the fields attribute and foreground in class Cell, and fields background and foreground in class Pen.
- Run 3 contains witnesses for the field attribute in class Cell, and the field x in class Cursor.

Annotation Comparison Between *jcurzez1* and *jcurzez1-AJ*. The basic version of *jcurzez* with coarse synchronization is called *jcurzez-redo-simple*. For simplicity, it will be henceforth be referred to as jcurzez1. The AJ variant of *jcurzez1* is *jcurzez1-AJ*. Besides adding annotations that define atomic sets and units of work, it also introduces the following refactorings:

- Direct field access across object boundaries is replaced by calls to getter and setter methods. (This is a trivial refactoring to fulfill the requirement that *fields belonging to atomic sets must be accessed through the (implicit)* **this** *reference* [6].)
- The Cursor class is annotated as internal (only in *jcurzez1-AJ*, not in *jcurzez2-AJ*), which means that its atomic set must always be aliased by the atomic set of the creating object, and references to Cursor objects may not leave this context.
  The getCursor() method is therefore removed from the AbstractWindow class. Indirect getters and setters for cursor properties take over its functionality (for example method setCursorXY()). For the same reason, the AbstractWindow constructors now accept a Rectangle argument in place of a Cursor.
- A set of methods is introduced to allow unit of work definitions:
  - A method move() with Rectangle arguments is added to class Window.
  - A method initPens() is introduced in class Rectangle.
  - The shutdown logic in class Screen$JCurzezHook is extracted into the method shutdownPeer().
- Constructors of the Area and Window classes have been removed to prevent undeclared sharing. (See the comment at the top of class Area.)
- The field aw, which refers to an AbstractWindow instance used for synchronization, was removed in several classes. The Field completed was introduced in ansi.PeerScreen.

Trivial differences between the inferred and the manually defined annotations that originate from these refactorings will be ignored in the discussion. In detail, the inferred annotations differ from the manual annotations as follows:

AbstractWindow: The manually added atomic set consists of only the buffer field, which aliases its entries (**this**.A[]). The inferred atomic set extends this alias to the entries in the *second* dimension, as well as their atomic sets. This alias structure (**this**.A[][]B=**this**.A) cannot be expressed in AJ; an output filter in the inference tool could easily prune the alias to the maximal expressible variant.
The inferred atomic set furthermore contains the final fields parent and cursor. These fields have been manually annotated with aliases, which match the inferred aliases. The other final fields included in the inferred atomic set are linewrap, scroll, id, isBufferShared, and bufferLock. Except bufferLock, which has been removed in jcurzez1−AJ, all are of primitive type and therefore immutable. With an output filter, the inference tool could remove these entries.
The field stressConcurrency serves for debugging. Its inclusion in the inferred atomic set does no harm. $+A\ (A)$

Area: No atomic set has been manually defined. However, an alias in the constructor merges the parent's atomic set into the instance's atomic set. The inferred atomic set for the super class AbstractWindow includes this alias. A type inference mechanism could generate this annotation in a post-processing step from field annotations. +A (A) +L (A)

−U (T) +U (A) Additionally, the method printCell is inferred to be a unit of work for its Cell argument; no such manual annotation exists. The argument is declared as final, but this does not prevent concurrent modification. The inferred annotation documents the intended behavior of the cell staying constant during printing.

+A (T) +U (T) Attribute: The class has no manual annotations. The inferred atomic set consists of the value field, which is final and of a primitive type. The atomic set could be removed in a post-processing step. All of the inferred units of work cover other Attribute arguments and would be eliminated by the same filter.

+A (B) +L (A) Cell: Neither an atomic set, nor units of work have been manually defined for this class. The inferred atomic set contains all of the class's fields. For the attribute, background, and foreground fields, this includes aliases to the respective atomic sets.

Runs 1 and 3 contain witnesses for interleaved access to some of the fields. Since thread-safety was added to jcurzez by the AJ authors (and not the original library developer), this is likely a synchronization error.

+U (T) The comparison method hasSameDecoration() is inferred to be a unit of work for the atomic set of the other Cell. No such manual annotation can exist because no atomic set has been manually defined for Cell. The same is true for the copy constructor.

+A (A) Color: No manual annotations have been added this this class. The algorithm infers two atomic sets for the class, one for each of the two fields. This documents that color objects are shared between multiple threads, but not modified concurrently. The definition of the atomic sets allows, for example, the Cell class to include the color properties in its atomic set and convey the intended semantics that no other thread changes the colors.

Cursor: The manual annotations define this class as an internal class. Its atomic set must therefore always be aliased by the atomic set of the creating object. References to Cursor objects may not leave this context. This leads to several refactorings; see the discussion above.

+A (A) Th first inferred atomic set contains all the manually annotated fields with the respective aliases. A separate atomic set is inferred for the field visibility, which has no manual annotations.

−U (T) The class's constructor is defined to be a unit of work of its Rectangle argument. This definition is missing from the automatically added annotations because the argument is not shared at the time of construction and is thus disregarded by the tool.

+A (A) +L (A) FillingPen: This class contains no manual annotations. The inference tool suggests an atomic set consisting of the three (wrongly attributed) fields of the super class Pen. The tool furthermore suggests an atomic set consisting of just the blankCell field with an alias to the atomic set of Cell. This alias is

correct and supplies additional information because the color and attribute fields of the cell can be updated via the set...() methods in FillingPenn.

Pen: The manually added atomic set covers all fields of the class. The inferred atomic set matches the definition and furthermore adds aliases for the atomic set in Color class that contains the field name. This may lead to locking overhead if Color instances are shared between Pen instances. $+L\ (A)$

The copy constructor is not invoked by the fuzzing tool, which explains why it is not inferred as a unit of work for its Pen argument. $-U\ (W)$

Rectangle: The manual annotations put all fields of the class into an atomic set. The fields fillingPen and drawingPen have aliases to the atomic set of Pen.

The inference algorithm derives the atomicity of all fields. It splits the single atomic set into three, putting the Pen fields into atomic sets of their own. While the alias for fillingPen is derived, no such alias exists for drawingPen because it was witnessed as non-atomic (via the method _printChar() in class AbstractWindow in all three runs). Thus, the original synchronization is incomplete, allowing this unintended interleaving. $-L\ (B)$ $-U\ (S)$

The private method initPens() is not inferred as a unit of work because it has been freshly introduced into jcurzez1−AJ during the refactoring.

Screen: An atomic set is neither manually defined, nor inferred for this class. The manual annotations define the method shutdownPeer() as a unit of work for its argument. This annotation is not inferred because the method does not exist in jcurzez1. It is introduced during the refactoring into jcurzez1−AJ. $-U\ (S)$

The manual annotations do not define the method printCell() as a unit of work for its argument because the class Cell contains no atomic set in jcurzez1−AJ. However, the method is protected by a lock in jcurzez1, which indicates that the inferred annotation reflects the programmer's intention. $+U\ (A)$

Window$Parent: The class contains no manually defined atomic set. The algorithm infers one atomic set consisting of the fields cursor and frame. The field cursor is wrongly attributed to this class. $+A\ (A)$

Defining the field frame as atomic is correct. However, the field is never written. Since Frame objects seem to never be shared, this annotation is unnecessary.

The inferred units of work only concern the compiler generated access methods for fields in the containing class Window. They can be safely ignored and could be removed by an output filter in the inference tool. $+U\ (T)$

Window: No atomic set is defined manually for this class. The inferred atomic set contains the field moved alongside the wrongly attributed fields cursor and parent, which are already covered in their defining class AbstractWindow. $+A\ (A)$

The method move() is not inferred as a unit of work for its Rectangle arguments because this method is not present in jcurzez1. $-U\ (S)$

ansi.PeerScreen: The manually defined atomic set mostly matches the larger of the inferred atomic sets. The completed field is missing because it was introduced in jcurzez1−AJ. The inferred atomic set additionally contains the fields columns and lines. These private fields are never written and could be declared final, whereby an output filter could remove them from the $-A\ (S)$ $+A\ (A)$

suggested annotations. Because lines is in a separate atomic set, a slight additional locking overhead is incurred.

$+U$ (T) The class contains no manually defined units of work. The inferred units of work concern methods whose argument types do not contain manually defined atomic sets.

$+U$ (B) The printCell() method accesses the properties of the Cell argument several times. Since the cell could be modified concurrently, the inferred annotation clarifies the concurrency invariants in the method.

A.5 JCurzez1-AJ

Workload. The same fuzzing tool as with *jcurzez1* is used to generate the execution traces.

Runs. Runs 1 and 2 yield identical annotations. Run 3 contains a witness of interleaved access to field attribute in the Cell class (in method copyInto()). This witness is not present in the other runs.

Annotation Comparison Between *jcurzez1-AJ-Translated* and *jcurzez1-AJ*. The inferred annotations for jcurzez1−AJ−Translated are equivalent to those inferred for jcurzez1 (*jcurzes-redo-simple*), modulo the refactoring from *jcurzez1* to *jcurzez1-AJ* and the compiler generated methods and fields. For a discussion of the differences between the manual and the inferred annotations, see the evaluation of the annotations inferred for jcurzez1.

The most salient changes in the inferred annotations are:

Now also $-L$ (T) AbstractWindow: The atomic set requested as alias for the field parent no longer contains the parent field (that is, parent.parent). However, the Window subclass still introduces this alias, which suggests that the effect is the result of the compiler attributing the field to the wrong class.

$+A$ (A) removed Area: The inferred atomic set is now empty, matching the manual annotations.

$+L$ (A) removed The parent alias is inferred in the Window class.

FillingPen: The atomic set consisting of the wrongly attributed fields is gone. The atomic set containing blankCell (which does not exist in the manual annotations) is still inferred.

$-U$ (S) removed JCurzezHook: The method shutdownPeer() is present in jcurzez1−AJ−Translated and is correctly inferred to be a unit of work for its PeerScreen argument.

Parent: No annotations are inferred for this class. It appears that the parent objects are never shared in jcurzez1−AJ−Translated.

Rectangle: The alias for drawingPen is now present. However, the alias for fillingPen changed to blankCell only.

$+U$ (A) removed Screen: The printCell() units of work disappeared. Since the method is just calling method printCell() in ScreenPeer (which still is a unit of work for its argument), this is irrelevant.

$-U$ (S) removed Window: The inferred atomic set now contains the parent alias, but frame and cursor vanished. The method move_internal() is correctly identified as unit of work for its Rectangle arguments.

A.6 JCurzez2

Workload. The same fuzzing tool as with *jcurzez1* is used to generate the execution traces.

Runs. Compared to the other runs, run 1 contains a witness for the field x in class Cursor. Furthermore, the fields background and attribute are missing from the alias of field drawingPen in class Rectangle. Run 2 contains a witness of interleaved access to the field background in class Pen, and the field moved in class Window. Run 3 contains witnesses for the fields background and foreground in class Cell, the field y in class Cursor, and the fields background and foreground in class Pen.

Annotation Comparison Between *jcurzez2* and *jcurzez2-AJ*. The *jcurzez-redo* project (henceforth called *jcurzez2*) is a variant of the jcurzez library that intends to offer a higher degree of concurrency than *jcurzez1* [6, sec. 7.2]. The structural differences between *jcurzez1* and *jcurzez2* are as follows:

- The coupling between AbstractWindow and Cursor objects has been loosened. It appears that the intended effect is to allow concurrent access to the Cursor object while operating on the AbstractWindow object. Several fields were introduced in class AbstractWindow to buffer the values from the Cursor object.
- The PeerScreen instance is always protected by an external lock. (See the comment in PeerScreen.java about the removed **synchronized** modifier.)
- Class Rectangle now uses the fields width and height instead of the fields right and bottom. The aw field has been removed from all classes.

The modifications have little effect. Ignoring the addition and removal of fields, as well as the wrong attribution by the compiler, the following differences between the annotations inferred for *jcurzez1* and *jcurzez2* remain:

Cell: Method copyInto() is now a unit of work for its Cell argument. The runs of *jcurzez2* contain more witnesses of interleaved access to the class's fields. (The same holds for the classes Cursor and Pen.)

FillingPen: The field hasChanged is now included in the alias of field blankCell.

Rectangle: The field drawingPen gained an alias to foreground; however, the field fillingPen lost its aliases to the atomic set containing attribute, background, and foreground.

Window$Parent: The field frame was witnessed as being non-atomic (and is consequently missing from the inferred atomic sets and aliases). Now $+A$ (T)

Considering the high degree ofsimilarity, the discussion of the annotations inferred for *jcurzez1* stays valid. The following list considers only the differences between the manual AJ annotations of *jcurzez-aj1* and *jcurzez2-AJ*. (Recall that the changes were made to achieve a higher degree of concurrency).

Now $+A$ (W)

AbstractWindow: The alias merging the atomic field of parent with the class's atomic set has been removed. This change implies that the constructor is no longer a unit of work for its parent argument. (The same is true for the subclasses Area and Window.) Furthermore, the alias to the elements of the buffer array were removed. Also, the atomic set of the Rectangle object returned by the getRectangle() method is no longer aliased with the atomic set of AbstractWindow.

None of these changes appear in the inferred annotations. The reason could be an insufficient degree of concurrency in the fuzzing tool, making all operations appear coarsely atomic.

Now $+A$ (A)

Now $+L$ (A)

Area: The constructor is no longer a unit of work for the atomic set of its Cursor argument.

Cursor: The class is no longer internal. The alias annotation of its field rectangle has been removed. None of the changes is reflected by the inferred annotations. As with AbstractWindow, a reason for not dropping the alias annotation of field rectangle could be a coarse grained schedule of the fuzzing threads. Another reason could be that the locking structure implicitly prevents concurrent access to the contents of the field rectangle.

Now also $+A$ (A) and $+L$ (A)

$-U$ (W) removed

Pen: The class no longer has an atomic set. This change implies that the constructor of class Rectangle can no longer be a unit of work for its Pen arguments. The inference tool still suggests an atomic set for this class consisting of all fields. In contrast to *jcurzez1*, the tool furthermore infers that the copy constructor is a unit of work for its argument Pen.

Now $-A$ (S) and $+U$ (B)

ansi.PeerScreen: The fields columns and lines have been added to the atomic set. This makes the manually defined atomic set identical to the inferred atomic set (which has not changed from *jcurzez1*).

A.7 JCurzez2-AJ

Workload. The same fuzzing tool as with *jcurzez1* is used to generate the execution traces.

Runs. Run 1 contains a witness of interleaved access to the field background in class Cell, and the field foreground in class Pen. Run 2 contains a witness for the field frame in class Parent. Run 3 differs from run 2 in additionally containing a witness for the field attribute in class Cell.

Annotation Comparison Between *jcurzez2-AJ-Translated* and *jcurzez2*. The inferred annotations for *jcurzez2-AJ-Translated* are mostly identical to those inferred for *jcurzez-redo* (*jcurzez2*). Ignoring the fields and methods generated by the compiler, the only differences are as follows:

Now also $+U$ (A)

AbstractWindow: The method updateCursorIfUnchanged() is inferred to be a unit of work for its Cursor argument.

$-U$ (S) removed
$-L$ (B)

Rectangle: The methods contains_internal() and move_internal() are inferred as units of work for their Rectangle arguments, which matches the manual annotations.

Window: Method move_internal() is inferred as unit of work for its Rectangle arguments, which is precisely what the manual annotations define. $-U$ (S) removed

ansi.PeerScreen: The field lines is merged into the main atomic set.

A.8 TSP2

Workload. The *tsp2* program is executed using 8 threads on the file map14 to generate the traces.

Runs. Runs 1 and 3 yield identical annotations. Run 2 does not suggest the method less_than() in class TourCreator as unit of work for priority queue elements.

Annotation Comparison Between *tsp2* and *tsp2-AJ*

Minimizer: The atomic sets are identical modulo the added lock field MinLock, which is not present in *tsp2-AJ*. $+A$ (T)

PrioQElement: This class has no atomic set in *tsp2-AJ*, whereas the inference algorithm adds both of its fields to a single atomic set. Both variants result in the same program behavior because PrioQElements are used exclusively by the TourCreator singleton, which synchronizes all operations on the priority queue elements. $+A$ (A)

However, the elements *are* accessed by multiple threads (via TourCreator). Adding the atomic set documents this behavior and is furthermore a requirement for the aliasing annotations in the TourCreator class. (See below.)

TourCreator: The inferred atomic set includes all fields of the manually added atomic set. It furthermore contains the TourLock, which was removed in *tsp2-AJ*, as well as the minimizer field. The minimizer field is final, and thus the annotation is unnecessary. The inference algorithm currently ignores the access flags of fields, but an enhanced version could remove unaliased final fields from the inferred atomic set. $+A$ (S) $+U$ (A)

The automatically derived annotations furthermore contain additional aliasing information for the PrioQ[] and Tours[] arrays. For the Tours[] array, the atomic set is extended to its elements through the alias **this**.TC[]. The alias emphasizes that not only the array itself is used atomically within TourCreator, but that TourCreator synchronizes updates of the array's elements. While the annotation does not change the program's behavior (the manual annotation without the alias has the right effect), it clearly documents an additional concurrency invariant without introducing runtime overhead. $+L$ (A)

The inferred aliasing information for the PrioQ[] array is even more detailed and includes the atomic set of PrioQElements. The original *tsp2* program allows unsynchronized access to the PrioQ[] array in the DumpPrioQ() method of TourCreator. This method is called by the TspSolver threads if the debug

flag is set, thereby creating a race condition. However, if the debugging information is disabled, the access to PrioQ[] *and* its elements is atomic:

- The initialize() method is invoked by the main() method before the TspSolver threads start.
- DumpPrioQ() is only called if the debug flag is set (which was not the case in our runs). (This method is missing synchronization using TourLock.)
- Every other access to PrioQ is synchronized using TourLock.

+*A* (*B*) The manual annotations in *tsp2-AJ* reflect the program's behavior with debugging disabled. By adding PrioQ to the atomic set TC of the object, the annotation fixes the race condition in DumpPrioQ(). The inferred atomic set additionally includes the alias **this**.TC[]I=**this**.TC for PrioQ, which documents the atomic handling of priority queue elements (without introducing locking overhead). Both atomic sets have the same effect, but the inferred variant conveys more information about the intended program behavior.

TourElement and Tsp: The manually added and inferred atomic sets are both empty.

+*A* (*T*) TspSolver: This class—the program's worker thread class—received no atomic set during the manual conversion. The inference algorithm introduces an atomic set for the shared singletons minimizer and tourCreator. Both fields are final, which means that the atomic set could be omitted. The atomic set introduces, if at all, minimal runtime overhead because the lock for the set is permanently held by the TspSolver instance thread itself.

A.9 TSP2-AJ

Workload. The annotations are inferred from traces generated by the same workload as *tsp2*.

Runs. Runs 1 and 2 yield identical atomic sets. Run 3 adds an alias for including the elements of the Tours array in the atomic set of TourCreator. The missing alias in runs 1 and 2 most likely originates in scheduling that always accesses the array elements from the same thread. (The array is then treated as *not shared* by the inference algorithm and hence removed from the atomic sets.) As the increased runtime indicates, the program uses a schedule different from *tsp2* because of the changed lock structure.

The alias is *not* missing due to interleaved access: the compiled program (*tsp2-AJ-Translated*) ensures atomic access to the array; the debug log contains no witness for an interleaved update of the array elements.

Annotation Comparison Between *tsp2-AJ-Translated* and *tsp2-AJ*. The inferred atomic sets for *tsp2-AJ-Translated* almost match those inferred for *tsp2*. The discussion of the differences to the manual annotations in *tsp2-AJ* consequently applies to *tsp2-AJ-Translated* as well. The only differences between the inferred atomic sets for *tsp2-AJ-Translated* and *tsp2* are as follows:

Now just $+L$ (B)

The Tours field in TourCreator lacks the aliasing to its elements in two of the three runs of *tsp2-AJ-Translated*. (See above.)

No run of *tsp2-AJ-Translated* infers TourCreator.lessThan() as an external unit of work. As with the Tours[] array, this is likely due to a more monolithic locking structure that reduces the shared access between the worker threads. $+U$ (A) removed

A.10 Weblech

Workload. The program collected files from a host-local webserver for 4 minutes.

Runs. In runs 1 and 3, the basicAuthPassword and basicAuthUser fields are in separate atomic sets of class SpiderConfig, while in run 2 they are in a single atomic set. Run 1 suggests the field spiderThreads in class SpiderConfig as atomic, while runs 2 and 3 ignore this field. Likely, the field was not shared in these runs and therefore filtered by the inference algorithm. There is *no* witness of interleaved access to the field spiderThreads.

Annotation Comparison Between *weblech* and *weblech-AJ*. *weblech-AJ* slightly differs from *weblech* in that it has been refactored to better fit the AJ concurrency control. The inferred annotations are therefore only partially comparable. *weblech-AJ* introduces an additional intermediate SpiderRunnable class. While in *weblech*, the Spider class itself is a Runnable that is executed by the worker threads (instantiated in Spider.start()), this behavior is extracted and moved to the SpiderRunnable class in *weblech-AJ*. In both variants, all worker threads execute a single Runnable instance. This object coordinates the threads with its download queues etc.

The URLGetter object, which is invoked via the downloadUrl() method, contains blocking calls to fetch the content located at the URL. In *weblech*, the thread (correctly) holds no locks during these blocking calls. Multiple threads can thus concurrently block for input, as can be verified by printing the thread ids at the beginning and end of getUrl(). However, the atomic set structure suggested by the inference algorithm would impose a global lock with the current, lock-based, implementation of atomic sets. This global lock (for the atomic set of the shared Spider instance) must be held by any thread operating on the Spider instance. This includes the call to downloadUrl(). Consequently, only a single thread can make downloads, eliminating all concurrency from the program.

This behavior is safe, but to allow concurrent downloads, the program must be refactored (as is the case with *weblech-AJ*). However, the refactoring in *weblech-AJ* does *not* achieve this goal: the downloadUrl() method is still located in the shared Spider instance and is consequently an internal unit of work for the atomic set S. The method should instead be moved to SpiderRunnable, which does not protect its fields with atomic sets. A temporary workaround to enable concurrent downloads in the translated version of *weblech-AJ* is to close the

synchronized(**this**.$lock_S) block in the downloadUrl() method of class Spider just before the calls to urlGetter.getUrl(url), and to re-open right after the calls.

The manually added and the inferred annotations of the individual classes differ as follows:

$+A$ (A) DownloadQueue: This class does not contain a manually added atomic set. The algorithm infers an atomic set containing all of the class's fields, which are collections for organizing the queued URLs in different categories. Since the Spider singleton is the only object that uses DownloadQueue (the queue field), the synchronization through the atomic set S in *weblech-AJ*'s Spider class suffices to guarantee atomic access as in *weblech*. The (missing) manual annotation is therefore correct. The inferred atomic set documents the exclusive access to the fields in DownloadQueue. Because of the inferred alias from the queue field in Spider (see below), no locking overhead is introduced.

HTMLParser: Both the manually added and automatically inferred atomic sets are empty.

$+L$ (A) Spider: The inferred atomic set contains all fields in the class and furthermore aliases the fields of SpiderConfig and DownloadQueue. Because the worker threads operate on a single Spider instance, they must synchronize via the lock associated with the atomic set, and therefore execute in a *purely sequential* order. In particular, threads can no longer wait concurrently for downloads to finish. As discussed above, this behavior is correct; enabling concurrent downloads with atomic sets *requires* a refactoring of the class structure. Since the run() method invokes all operations in the class, all fields are always accessed together, resulting in a single inferred atomic set.

$+A$ (B) The *weblech-AJ* version defines an atomic set that contains only three fields responsible for managing downloads. The fourth field related tod downloads, urlsDownloadedOrScheduled, and the methods operating on it have been moved to the UDorSWrapper class to work around the limitation of one atomic set per class (see comments in UDorSWrapper.java). The fields related to checkpointing are omitted from the atomic set. This introduces an, albeit benign, race condition regarding checkpoints that *weblech* prevents by synchronizing on queue (see method checkpointIfNeeded()). The fields related to thread management (running and quit) have been moved to SpiderRunnable in *weblech-AJ* and are not members of an atomic set.

$+A$ (A) SpiderConfig: No atomic set is defined for this class in *weblech-AJ*. Because the fields are only read after the configuration has been loaded, no data races can originate from this lack of synchronization. Nevertheless, the fields are accessed from multiple threads, and consequently, the inference algorithm defines an atomic set for each field. The atomic sets document the intended (read–read sharing) behavior of the fields. With the central Spider class defining aliases for all the atomic sets, no locking overhead is introduced.

$+L$ (A) URLGetter: The algorithm infers an alias for the userAgent, basicAuthUser, and
$-A$ (A) basicAuthPassword fields in the shared SpiderConfig object. This documents that the configuration value stays constant, but no more. It does not change the program's behavior compared to the empty atomic set in *weblech-AJ*.

+U (A) The constructor is furthermore inferred to be a unit of work for the atomic set containing the basicAuthPassword and basicAuthUser fields in the SpiderConfig class.

URLObject: Similar to URLGetter. +L (A)

URLToDownload: The class lacks an atomic set in both program versions. +U (A)

TextSpider: See URLToDownload.

Log4j: See URLToDownload.

A.11 Weblech-AJ

Workload. Like the non-AJ variant, the program collected files from a host-local webserver for 4 minutes.

Runs. The schedule of run 3 does not include shared access to the UDorSWrapper (via SpiderRunnable), which causes the inference algorithm to remove the field urlsDownloadedOrScheduled from the atomic set of UDorSWrapper. Runs 1 and 2 do not have this issue.

Annotation Comparison Between *weblech-AJ-Translated* and *weblech-AJ*. The atomic sets inferred from the field access traces of *weblech-AJ-Translated* virtually match those of *weblech*. The differences arise from the refactorings that were applied to *weblech-AJ*: the Spider class was split and several of its fields were moved to the new classes SpiderRunnable and UDorSWrapper. See the evaluation of *weblech* for a detailed discussion of these refactorings.

SpiderRunnable: The manual annotations define no atomic set for this class. In contrast, the algorithm infers an atomic set consisting of all fields of the class and including aliases to the atomic sets of Spider and SpiderConfig. This atomic set again introduces a global lock that forces all threads to execute in a sequential order because all threads operate on a SpiderRunnable singleton, and the run() method forces the executing thread to obtain the lock for the atomic set. As discussed in the evaluation of *weblech*, this structure of atomic sets is safe, but it eliminates all concurrency from the program. +A (A)

One aspect that leads to inferring this restrictive combination of atomic sets is the limited number of thread switches in the execution trace, which partially originates in the accidentally coarse concurrency structure of the manual AJ annotations (see subsection A.10). Sprinkling Thread.yield() statements over the run() method of class SpiderRunnable has the desired effect of removing the aliases from the inferred atomic set of SpiderRunnable.

Unfortunately, this does not remove the imposed global lock. The reason that the algorithm infers an atomic set for the fields of SpiderRunnable is that the threads executing the SpiderRunnable singleton never write the object's fields. The algorithm regards read–read sharing as atomic access because it best captures the programmer's intended semantics of the shared object. In

the lock-based implementation of atomic sets, this results in a (correct but) restrictive locking structure.

Concurrency can be restored by either moving to a different implementation of atomic sets, or by refactoring the application. In general, the lock-based atomic set implementation seems problematic for architectures in which multiple worker threads execute a *single* shared runnable object.

$+A$ (A)
$+L$ (A)

UDorSWrapper: The inferred atomic set includes the manually specified field urlsDownloadedOrScheduled and furthermore adds the lock field introduced by the AJ compiler, as well as the config field. The config field introduces an alias to the atomic set in the SpiderConfig class that contains the urlMatch field. This documents that the configuration option does not change when accessed by the UDorSWrapper object. Since no other class makes use of this configuration field, the alias does not introduce locking overhead.