Automated Inference of Atomic Sets for Safe Concurrent Execution

Peter Dinges
Dept. of Computer Science
University of Illinois
Urbana–Champaign, USA
pdinges@acm.org

Minas Charalambides
Dept. of Computer Science
University of Illinois
Urbana–Champaign, USA
charala1@illinois.edu

Gul Agha
Dept. of Computer Science
University of Illinois
Urbana–Champaign, USA
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 enables safe concurrency by preventing unobserved interleavings which may harbor latent *Heisenbugs*.

Categories and Subject Descriptors

D.3.3 [**Programming Languages**]: Language Constructs and Features—*Concurrent programming structures*; D.2.3 [**Software Engineering**]: Coding Tools and Techniques—*Object-oriented programming*

General Terms

Languages, Algorithms

Keywords

Atomic Sets, Data-Centric Synchronization

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 [16, 11]. 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 [16, 5], 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 that 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 [5] and deadlock-freedom [14].

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 [5]. 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 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 that 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

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
PASTE'13, June 20, 2013, Seattle, WA USA
Copyright 2013 ACM 978-1-4503-2128-0/13/06 ...$15.00.

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. However, it does not infer alias annotations for local variables. The algorithm 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 [17].

2. BACKGROUND: ATOMIC SETS

The algorithm presented in this paper infers annotations for data-centric synchronization that is based on *atomic sets* [5]. 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 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 12 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. Hence, 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.

```
 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   public void addAll(unitfor(L) List other) {
13     length = length + other.length;
14     /*...*/
15   }
16 }
17
18 class DownloadManager {
19   atomicset U;
20   atomic(U) List urls|L=this.U|;
21
22   public void downloadNext() {
23     Object u = urls.get(0);
24     if (u != null) {
25       urls.remove(0);
26       download(u);
27     }
28   }
29 }
```

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

Additionally, AJ recognizes partial **unitfor** declarations and fastread annotations. These simplify the specification of units of work and help the AJ compiler generate more efficient concurrency-control code. These annotations are secondary; we do not consider their inference in this paper.

3. ALGORITHM SYNOPSIS

The assumption behind our algorithm for inferring the aforementioned annotations for data-centric synchronization 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 (section 4). 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 (section 5). 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 (section 6).

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.

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.

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.

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

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

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

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

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

Figure 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.

The enter, get, and put events record values. These values are either object identities or the special symbol $\bot$ 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).

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.

Abstract Machine

The observation phase extracts the sets of accessed fields by replaying the field access trace on an abstract machine that monitors which fields each method accesses. The abstract machine executes the events in the trace and applies their effects to its *configuration*. A configuration

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

consists of the global timestamp γ, the state of the thread-owned stacks, the access history of the heap shared by all threads, and the collected observations. The initial configuration has timestamp 0, and empty stacks, heap, and

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

$$intSugg(c) = \bigcup_{m \in methods(c)} \big(\{ f \mid \big(obs(m)\big)(\textsf{this}.f) = (\text{atomic}, \text{direct}) \} \setminus nonAtomic(c) \big)$$

$$extSugg(c) = \bigcup_{m \in \text{Method}} \big(\{ f \mid \exists p : p \neq \textsf{this}, \big(obs(m)\big)(p.f) = (\text{atomic}, _), \text{ and } scope(f) = c \} \setminus nonAtomic(c) \big)$$

Figure 3: Aggregation functions for suggesting atomic sets. For a class c, *nonAtomic* computes the set of fields excluded from any atomic set, and *intSugg* and *extSugg* compute the sets of suggested atomic sets. While *intSugg* aggregates observations from the methods of c, *extSugg* uses observations from all methods of all classes.

observations. 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 trace event. For brevity, the remainder of this section contains only a high-level description of the machine's operation. For a complete formalization, please see the full version of this paper [4].

Field Access Observation

The abstract machine maintains a stack of method calls for each thread to track the syntactic scope of field access events. Within each stack frame, it records which fields of which objects the method reads and writes. When the method exits, the machine records for each accessed field whether the access was interleaved between concurrent threads, or atomic within the method's scope. To detect interleaved access, it compares a timestamp representing the latest thread-local version of a field's value against the timestamp of the field's value in the model of the shared heap. Threads update the global timestamp only on write events; read–read sharing therefore does not count as interleaved access. To record the fields accessed by a method's callees, as motivated in section 3, the abstract machine merges the access information of popped stack frames into the frame beneath. Thus, access to a field can be observed as a combination of direct or indirect, and atomic or interleaved access. The algorithm resolves the access paths of fields by maintaining a directed graph of objects that were accessed in the scope. An edge in the graph signifies that the target object is the value of a field in the source object; the edge label stores the field's name.

While replaying the input trace, the abstract machine aggregates the observations for each method m. At the end, the *obs* component of the configuration contains the observations for each method.

$$obs(m) : \text{AccsPath} \to \text{AccsStat} \times \{\text{direct}, \text{indirect}\}$$

where $\text{AccsStat} = \{\text{none}, \text{read}, \text{written}\}$ and AccsPath = Parameter ("." Field)*.

6. ATOMIC SET FORMATION

This section describes how atomic sets, aliases, and units of work are derived from the collected observations.

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.

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 suggestions for internal and external atomic sets are formed using the following inference rules:

- Fields with internal witnesses are *non-atomic* and are therefore excluded from all atomic sets. Limiting the witnesses to *direct* observations—made in the method's scope, not one of its callees—prevents overly emphasizing witnesses from scopes 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.
- 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 *non-atomic* fields are computed by the function *nonAtomic*, which is defined in Figure 3. The function *intSugg* returns the internal suggestions for atomic sets; the function *extSugg* returns the external suggestions.

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

$$succ(f, s) = \bigcup_{m \in \text{Method}} \{g \mid \exists p : \big(obs(m)\big)(p.f.g) = (s, _)\}$$
$$units(m, q, s) = \{A \in atomicSets\big(type(q)\big) \mid A \cap paramFields(m, q, s) \neq \emptyset\}$$
$$paramFields(m, q, s) = \{f \mid \big(obs(m)\big)(q.f) = (s, _)\}$$

Figure 4: Auxiliary functions for alias and unit of work computation. Function *succ* returns the set of fields that are access path successors of f which have been observed as $s \in \{\text{atomic}, \text{interleaved}\}$ in any method. Function *units* computes the set of atomic sets containing a field accessed via parameter q of method m.

After pruning, the observations of all non-empty prefixes indicate atomic access.

To obtain the final atomic sets, the 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 in the function *merge*. 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 $atomicSets(c)$ for a class c inferred by the algorithm are

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

Aliases

The rules for inferring atomic sets focus on the observations about the final segments of access 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 access 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 access 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

$$succ(f, \text{atomic}) \setminus succ(f, \text{interleaved})$$

with function *succ* defined in Figure 4.

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.

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:

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

where the function *units* is defined in Figure 4.

Computing the units 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[2] 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. To allow the tool to process realistic programs, we have extended it with the ability to handle arrays, synchronized blocks, and wait–notify synchronization. In addition, we have included heuristics that add special handling of monitor variables and constructors, and remove thread-local fields from the output.

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. Additionally, the tool chain currently does not export the access flags of fields. It hence cannot remove final primitive fields from the output.

[1]Available at `http://osl.cs.illinois.edu/software/`
[2]T. J. Watson Libraries for Analysis. `http://wala.sf.net`

8. EVALUATION

This section discusses the performance of our algorithm measured by the quality of the inferred annotations. The complete discussion and detailed results are available in the full version of this paper [4].

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 it contained too few AJ annotations to justify the effort of creating a fuzzing tool for it. 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. [5]. Archives containing the source code of the conversions are available on the *Data-Centric Concurrency Control* project website[3]. The AJ variant of the Java collections framework was kindly provided by Frank Tip.

Table 1: Programs used to evaluate the inference algorithm. 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. In parentheses follows the number of classes that contain at least one manual AJ annotation (atomicset, atomic, or unitfor).

Program	*Description*	*kLoC*	*Classes*
collections	OpenJDK 1.6 collections	11.1	171 (43)
elevator	Elevator simulation	0.3	6 (2)
jcurzez1	UI library (low concur.)	2.7	78 (9)
jcurzez2	UI library (high concur.)	2.8	79 (6)
tsp2	Traveling salesman	0.5	6 (2)
weblech	Web site mirror tool	1.3	12 (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. Creating the fuzzing programs took about half a day per library. However, this effort could be automated, or unit tests could be used instead where available. All 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.

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 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.

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 code. 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 and sometimes even incorrect. Second, using other quantitative measures like execution speed is infeasible 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 flattening nested classes and splitting classes to achieve concurrent execution. Simplifications 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.

Results

The inferred annotations can differ from the manual annotations in both missed and added atomic set, alias, and unit of work definitions. Missing annotations are those that were manually added, but not (completely) inferred. Added annotations are those that were inferred, but not manually added.

- The most critical kind of difference is a missing or incomplete atomic set, implying that some fields that were intended to be protected from interleaved access remain unprotected, which may result in a race condition. Additional atomic sets can lead to deadlock, but this is not a severe problem because deadlock caused by atomic sets can be statically recognized [14].
- Missing aliases result in synchronization overhead without affecting the program's correctness. Additional aliases reduce the potential for concurrency in the program. However, extraneous aliases cannot lead to errors like race conditions or deadlocks.
- Missing unit of work declarations can lead to race conditions. Additional declarations may reduce the concurrency in the program and lead to (statically recognizable) deadlock.

Overall, the inferred annotations mostly agree with the manual annotations. The inferred annotations are missing an atomic set for only two classes; one of these cases documents a mistake in the manual annotations of the *collections* framework. Additional atomic sets are inferred for 24 classes; in three classes, they prevent inadvertent race

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

conditions. The algorithm fails to infer aliases for three classes in total. Two of these belong to the *jcurzez* variants and originate in race conditions in the Java version. New aliases are added to 15 classes overall, fixing one race condition in *tsp2*. Finally, only one class misses a (incorrect) unit of work declaration. Additional unit of work declarations are introduced to 13 classes in total.

These counts ignore secondary causes of differences that either reflect refactorings, or can be fixed through a better tool implementation or workload. In particular, the fuzzer used to drive the *collections* is incomplete. For example, it does not access iterators from different threads. The *shared objects* heuristic described in section 7 therefore removes annotations for their fields, which results in (uncounted) missing atomic sets in three classes.

The one class missing an atomic set is part of *weblech*. A field is omitted for unknown reasons. This could be a bug in our implementation, but because we were cannot rule out an algorithm deficiency, we categorize it as this most severe difference type.

The high number of additional aliases documents the importance of choosing a workload that exerts as much concurrency in the program as possible.

Inferring annotations for the manually ported AJ variants of the programs yields results similar to the those of the original variants. This indicates that the manual annotations capture most of the original program's behavior. It also shows that the manual refactorings influence the inference very little.

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

In *elevator*, the 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 having the right effects in the AJ implementation, this annotation violates the atomic set 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 fails to enable concurrent downloading.

The inferred annotations for *jcurzez1* reveal race conditions in the classes Cell, Cursor, and Pen. In its AJ variant, 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 underlines 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 mistakenly added atomic set in *collections* adds further support.

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. [13, 12]. *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. [10] 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 [6] also considers *windows of vulnerability*, but requires a few source code annotations.

The *MUVI* tool of Lu et al. [11] follows a static approach to inferring atomicity intentions. It computes variable correlations by mining the program source code. The static heuristic [8, 15] 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 [9]. 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* [7, 3] 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 Synchronizers can 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 extensions similar to aliases in atomic sets.

The algorithm introduced in this paper follows the approach of *accentuating the positive* [17, 12]: 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. [17] show that a low runtime overhead of 15 % can be achieved using this method.

10. DISCUSSION

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 rarely) appear during testing will likely not be observed. In this case, the inferred annotations will prevent the bugs in future executions.

The major factor driving the suggestion of additional annotations is the documentation of *obvious* behavior. Obvious behavior concerns high-level understanding 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.

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. It would be feasible to automate the generation of workloads, for example using concolic execution to explore thread scheduling.

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 *unit-for*, and *internal*, would improve the degree of concurrency. As demonstrated by Dolby et al. [5], 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 W911NF-09-1-0273, as well as the Air Force Research Laboratory and the Air Force Office of Scientific Research under agreement FA8750-11-2-0084. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright notation thereon.

11. REFERENCES

[1] C. Artho, K. Havelund, and A. Biere. High-level data races. In *NDDL '03*, pages 82–93. ICEIS Press, 2003.

[2] M. Burrows and K. R. M. Leino. Finding stale-value errors in concurrent programs. *Concurrency - Practice and Experience*, 16(12):1161–1172, 2004.

[3] P. Dinges and G. Agha. Scoped synchronization constraints for large scale actor systems. In *COORDINATION '12*, pages 89–103. Springer, 2012.

[4] P. Dinges, M. Charalambides, and G. Agha. Automated inference of atomic sets for safe concurrent execution. Technical report, UIUC, April 2013. `http://hdl.handle.net/2142/43357`.

[5] J. Dolby, C. Hammer, D. Marino, F. Tip, M. Vaziri, and J. Vitek. A data-centric approach to synchronization. *ACM TOPLAS*, 34(1):4, 2012.

[6] C. Flanagan and S. N. Freund. Atomizer: A dynamic atomicity checker for multithreaded programs. *Sci. Comput. Program.*, 71(2):89–109, 2008.

[7] S. Frølund and G. Agha. A language framework for multi-object coordination. In *ECOOP '93*, pages 346–360. Springer, 1993.

[8] C. Hammer, J. Dolby, M. Vaziri, and F. Tip. Dynamic detection of atomic-set-serializability violations. In *ICSE '08*, pages 231–240. ACM, 2008.

[9] W. Huang and A. Milanova. Inferring AJ types for concurrent libraries. In *FOOL '12*, pages 82–88, 2012.

[10] Z. Lai, S.-C. Cheung, and W. K. Chan. Detecting atomic-set serializability violations in multithreaded programs through active randomized testing. In *ICSE '10*, pages 235–244. ACM, 2010.

[11] S. Lu, S. Park, C. Hu, X. Ma, W. Jiang, Z. Li, R. A. Popa, and Y. Zhou. MUVI: Automatically inferring multi-variable access correlations and detecting related semantic and concurrency bugs. In *SOSP '07*, pages 103–116. ACM, 2007.

[12] S. Lu, S. Park, and Y. Zhou. Detecting concurrency bugs from the perspectives of synchronization intentions. *IEEE Trans. Parallel Distrib. Syst.*, 23(6):1060–1072, 2012.

[13] S. Lu, J. Tucek, F. Qin, and Y. Zhou. AVIO: Detecting atomicity violations via access-interleaving invariants. *IEEE Micro*, 27(1):26–35, 2007.

[14] D. Marino, C. Hammer, J. Dolby, M. Vaziri, F. Tip, and J. Vitek. Detecting deadlock in programs with data-centric synchronization. Research Report RC25300 (WAT1208-051), IBM, 2012.

[15] W. N. Sumner, C. Hammer, and J. Dolby. Marathon: Detecting atomic-set serializability violations with conflict graphs. In *RV*, pages 161–176. Springer, 2011.

[16] M. Vaziri, F. Tip, and J. Dolby. Associating synchronization constraints with data in an object-oriented language. In *POPL '06*, pages 334–345. ACM, 2006.

[17] D. Weeratunge, X. Zhang, and S. Jagannathan. Accentuating the positive: Atomicity inference and enforcement using correct executions. In *OOPSLA '11*, pages 19–34. ACM, 2011.