Automatic Synchronization Correction

Cormac Flanagan
Department of Computer Science
University of California, Santa Cruz
Santa Cruz, CA 95064

Stephen N. Freund
Department of Computer Science
Williams College
Williamstown, MA 01267

ABSTRACT

Multithreaded programs are notoriously prone to synchronization errors. Much prior work has tackled the problem of detecting such errors. This paper focuses on the subsequent problem of *synchronization correction.* We present a constraint-based analysis that, given an erroneous program, automatically infers (where possible) what additional locking operations should be inserted in order to yield a correctly-synchronized program. For performance reasons, our algorithm also attempts to minimize the number of additional lock acquires and the duration for which the acquired locks are held. We present experimental results that validate this approach on a number of standard Java library classes.

1. INTRODUCTION

Multithreaded programs are notoriously prone to errors due to incorrect synchronization. Earlier work in this area focused on detecting synchronization errors that cause race conditions [24, 23, 28, 29, 1, 4, 9, 11], atomicity violations [14, 8, 10, 12, 19, 31], and other consistency violations [2, 30]. Of course, *detecting* defects is useful only if it is followed by a second step of *defect correction.* In this work, we focus on this subsequent defect correction phase, and in particular on the problem of providing automated support for correcting synchronization errors.

The goal of this work is, given a program that fails to satisfy its atomicity specification, to modify the program (for example, by introducing additional synchronization operations) so that it satisfies its specification. In addition, we try to minimize the number of additional locks required (to reduce synchronization overhead) and to only hold these locks for short durations (to reduce lock contention [3]). We assume some methods in the original program are documented as `atomic`, which means that they should include sufficient synchronization so that their execution is *serializable* (i.e., they can be considered to execute without interleaved actions of concurrent threads).

Inferring which additional synchronization operations are necessary to satisfy this atomicity specification requires performing a deep analysis on the program's synchronization structure. In previous work, we (1) developed *Rcc/Sat*, an analysis for inferring protecting locks for each field [11], and (2) a constraint-based framework for inferring the most precise atomicity for each method [12]. In this paper, we extend this line of research to tackle the more difficult problem of synchronization correction.

Figure 1: Class `Stack` and Inferred Locking Annotations

```
class Elem⟨ghost x⟩ {
  int num guarded_by x;
  Elem⟨x⟩ next guarded_by x;
}

class List {
  Elem⟨this⟩ elems guarded_by this;

  void add(int v) {
    this.elems = new Elem⟨this⟩(v,this.elems);
  }

  int removeFirst() {
    sync(this) {
      let int x = this.elems.num in {
        this.elems = this.elems.next;
        return x;
      }
    }
  }
}

class Stack {
  final List data;

  atomic void push(int x) {
    this.data.add(x);
  }

  atomic int dup() {
    let int x = this.data.removeFirst() in {
      this.data.add(x);
      this.data.add(x);
    }
  }
}
```

Inserting additional synchronization can introduce the potential for deadlock. We do not explicitly reason about deadlock, and instead assume the programmer ensures deadlock-freedom by code inspection or via some other static or dynamic analysis.

We present preliminary results that validate this analysis on three standard Java library classes that are known to have synchronization errors. For each class, our analysis can automatically infer the additional synchronization operations that are necessary to correct these defects. We

Presented at the Workshop on Synchronization and Concurrency in Object-Oriented Languages (SCOOL), October, 2005.

also performed defect-injection experiments, which demonstrated that the analysis is capable of correcting the vast majority of randomly-inserted synchronization defects in Java library classes.

2. MOTIVATING EXAMPLE

We illustrate the behavior of our analysis on the example shown in Figure 1, in which a `Stack` is represented as a linked `List` of `Elements`. The underlined annotations specify the program's locking structure. For example, the field `List.elems` is guarded by `this`, the implicit lock of the `List` object. This lock also guards the fields `List.elems.num` and `List.elems.next`, since `elems` has type $\texttt{Elems}\langle\texttt{this}\rangle$, and in `Elems` the ghost parameter `x` guards the `num` and `next` fields. These underlined type annotations can be automatically inferred by the *Rcc/Sat* type inference algorithm [11].

The class `Stack` is intended to be thread-safe, and so its `push` and `dup` methods are declared as `atomic`. However, the given program violates this atomicity specification. For example, `push` and `dup` call `add` without acquiring any locks, resulting in race conditions. Synchronizing on `this` inside `add` corrects this race condition and ensures that `push` is atomic, but `dup` is still incorrect, since a concurrent threads could modify the stack between `dup`'s two calls to `add`. Thus, inferring the necessary synchronization operations requires reasoning about both race conditions and atomicity violations.

Given this incorrect program, our algorithm automatically generates the corrected program of Figure 2(b), where the methods `add` and `dup` contain additional synchronization. (Alternatively, inserting synchronization into `push` and `dup` would also suffice.) Our algorithm also infers the most precise atomicity for each unannotated method. For example, the method `add` is assigned the conditional atomicity "`this?mover:atomic`", which states that if the lock `this` is already held, then `add` is a `mover` (that is, its execution commutes with steps of concurrent threads); if `this` is not held, then `add` is `atomic` (that is, it can be assumed to execute in a serial manner, without interleaved steps of concurrent threads).

Our analysis works in four phases.

1. The first phase encloses each program statement e in the *tagged synchronization operations*:

   $$\texttt{sync?}t_1\ (l_1)\ \{\ \ldots\ \{\ \texttt{sync?}t_n\ (l_n)\ \{\ e\ \}\ \}\ \ldots\}$$

   where $l_1, \ldots, l_n$ are all locks in scope, and the t_i are tags that uniquely identify each inserted synchronization operation. This phase also inserts atomicity variables α_i for methods without declared atomicities, such as `add`, producing the program shown in Figure 2(a).

   The goal of our analysis is then to determine a set T of tags denoting which of these tagged synchronization operations are necessary and sufficient to yield a correctly synchronized program. As part of its reasoning, the analysis also needs to infer an *assignment* A mapping atomicity variables α_i to atomicities.

2. The second phase of the analysis translates the program with tagged synchronization operations (as in Figure 2(a)) into a collection of constraints $\bar{C}$ over the tag set T and assignment A.

3. The third phase solves these constraints using an iterative least fixed point algorithm, to yield the program of Figure 2(a) with the grayed-out synchronization operations removed. This program is correctly synchronized, but still contains some unnecessary synchronization operations.

4. The fourth phase then identifies and removes these redundant synchronization operations, yielding the final program of Figure 2(b).

This constraint-based approach extends our earlier work on detecting synchronization errors [12], but for synchronization correction the addition of tagged synchronization operations requires an extended constraint language and new constraint solving algorithms.

The presentation of our results proceeds as follows. Section 3 describes the idealized Java subset that we used to formalize our analysis. Section 4 presents our constraint language. Section 5 describes how to generate constraints, which are then solved by the algorithm of Section 6. This solution is then optimized in Section 7. Our implementation and experiments are described in Section 8. Section 9 discusses related work, and Section 10 concludes.

A preliminary version of this work will be presented at the SCOOL Workshop, Oct. 2005.

3. THE SOURCE LANGUAGE AJC

We formalize our ideas in terms of the idealized language AJC (Atomic Java with Correction), whose syntax is summarized in Figure 3. For simplicity, the idealized language AJC does not support subclassing, although it is supported in our implementation [12].

As in Java, each field declaration includes the name and type of the field. Additionally, in AJC, each field also has an associated *guard* g, which states that the field is either (1) `final`, (2) `unguarded`, or (3) `guarded_by` some lock l, which must be held at each access (*i.e.*, read or write) to that field. Sometimes the fields of a class need to be protected by a lock external to the class. For this purpose, each AJC class declaration includes a binding for a sequence of *ghost variables* denoting locks that can be used to protect fields of the class. Ghost variables are only used during type checking and do not exist at run time. A class type $cn\langle l^*\rangle$ includes a class name cn and an appropriate number of lock parameters for that class. The *Rcc/Sat* type inference algorithm can infer appropriate guards and lock parameters for unannotated programs [11].

A method declaration can also include a number of ghost variable bindings, for which corresponding lock expressions must be provided at call sites. Each method declaration also includes an *atomicity specification* s, such as `atomic` or `this?mover:atomic`, as described in Section 4.4.

AJC expressions include object allocation, field access and update, method invocation, variable binding and reference, conditionals, and while loops. Object allocation $\texttt{new}_y\ c(e^*)$ includes a sequence of expressions used to initialize the object fields. For technical reasons, the `new` keyword is subscripted by `y`, which is a ghost variable bound to the object being created while evaluating the field initialization expressions. This enables the types of the initialization expressions to refer to the new object. We omit the binding from examples when it is not needed.

Figure 2: Class Stack with Atomicity Annotations and Corrected Synchronization

a) Program with Atomicity Annotations

```
class Elem⟨ghost x⟩ {
  int num guarded_by x;
  Elem⟨x⟩ next guarded_by x;
}

class List {
  Elem⟨this⟩ elems guarded_by this;

  α1 void add(int v) {
    sync?t1 (this)
      this.elems = new Elem⟨this⟩(v,this.elems);
  }

  α2 int removeFirst() {
    sync?t2 (this)
      sync(this) {
        sync?t3 (this)
          let int x = this.elems.num in {
            sync?t4 (this)  {
              sync?t5 (this)
                this.elems = this.elems.next;
              sync?t6 (this)
                return x;
            }
          }
      }
  }
}

class Stack {
  final List data;

  atomic void push(int x) {
    sync?t7 (this)
      sync?t8 (this.data)
        this.data.add(x);
  }

  atomic int dup() {
    sync?t9 (this)
      sync?t10 (this.data)
        let int x = this.data.removeFirst() in {
          sync?t11 (this)
            sync?t12 (this.data)  {
              sync?t13 (this)
                sync?t14 (this.data)
                  this.data.add(x);
              sync?t15 (this)
                sync?t16 (this.data)
                  this.data.add(x);
            }
        }
  }
}
```

b) Program with Correct Synchronization

```
class Elem⟨ghost x⟩ {
  int num guarded_by x;
  Elem⟨x⟩ next guarded_by x;
}

class List {
  Elem⟨this⟩ elems guarded_by this;

  this?mover:atomic void add(int v) {
    sync(this)
      this.elems = new Elem⟨this⟩(v,this.elems);
  }

  this?mover:atomic int removeFirst() {

      sync(this) {

          let int x = this.elems.num in {


                this.elems = this.elems.next;

                return x;

          }
      }
  }
}

class Stack {
  final List data;

  atomic void push(int x) {


        this.data.add(x);
  }

  atomic int dup() {

      sync(this.data)
        let int x = this.data.removeFirst() in {




                  this.data.add(x);


                  this.data.add(x);

        }
  }
}
```

Figure 3: AJC Syntax

$$
\begin{array}{rcll}
P & ::= & \mathit{defn}^* \ e & \text{(program)} \\
\mathit{defn} & ::= & \texttt{class}\ cn\langle\texttt{ghost}\ x^*\rangle\ \mathit{body} & \text{(class decl.)} \\
\mathit{body} & ::= & \{\ \mathit{field}^*\ \mathit{meth}^*\ \} & \text{(class body)} \\
\mathit{field} & ::= & c\ \mathit{fn}\ g & \text{(field decl.)} \\
g & ::= & \texttt{final} \mid \texttt{guarded\_by}\ l & \\
 & & \mid \texttt{unguarded} & \text{(field guards)} \\
\mathit{meth} & ::= & s\ c\ mn\langle\texttt{ghost}\ x^*\rangle\texttt{(}arg^*\texttt{)}\ \{\ e\ \} & \text{(method decl.)} \\
\mathit{arg} & ::= & c\ x & \text{(arg. decl.)} \\
c & ::= & cn\langle l^*\rangle & \text{(class type)} \\
l & ::= & e & \text{(lock expr.)} \\
 & & & \\
e & ::= & x \mid \texttt{null} \mid \texttt{new}_y\ c\texttt{(}e^*\texttt{)} & \text{(expressions)} \\
 & & \mid e.\mathit{fd} \mid e.\mathit{fd}\ \texttt{=}\ e \mid e.mn\langle l^*\rangle\texttt{(}e^*\texttt{)} & \\
 & & \mid \texttt{let}\ c\ x\ \texttt{=}\ e\ \texttt{in}\ e \mid \texttt{while}\ e\ e \mid \texttt{if}\ e\ e\ e & \\
 & & \mid \texttt{sync}\ l\ e \mid e.\texttt{fork} \mid \texttt{sync?}t\ l\ e &
\end{array}
$$

$$
\begin{array}{ll}
x, y \in \mathit{Var} & t \in \mathit{Tag} \\
cn \in \mathit{ClassName} & fn \in \mathit{FieldName} \\
mn \in \mathit{MethodName} &
\end{array}
$$

The language supports multiple threads of control via the construct e.`fork`. Here, e should evaluate to an object with a nullary `run` method, which is called by a newly-spawned thread.

Synchronization between threads is achieved via the construct `sync` l e, which executes by first evaluating l to yield an object reference; the implicit lock associated with that object is then acquired; the expression e is then evaluated, and finally the lock is released. AJC also contains a tagged synchronization construct `sync?`t l e, where t is a unique tag identifying that operation.

Although omitted from the formal system for simplicity, our examples use integers and sequential composition, which we treat in the expected fashion.

4. ATOMICITY CONSTRAINTS

4.1 Basic Atomicities

Our approach to verifying atomicity is based on classifying expressions according to what actions they may perform, and in particular how these actions may interact with actions of concurrent threads. For this purpose, we introduce the following five *basic atomicities*:

$$b \quad ::= \quad \texttt{const} \mid \texttt{mover} \mid \texttt{atomic} \mid \texttt{cmpd} \mid \texttt{error}$$

The informal meaning of these basic atomicities is as follows:

- `const` expressions do not access any mutable state, and hence always yield the same result when evaluated in the same environment. Such expressions may include calls to `const` methods, etc.

- `mover` expressions can access mutable state, but only if that mutable state is either local to the current thread or protected by a lock held by the current trend. In addition, `mover` expressions cannot acquire or release locks. Thus, `mover` expressions commute with actions of concurrent threads.

- `atomic` expressions, in addition to accessing thread-local or protected state, may also acquire and release locks according to the two-phase locking discipline. That is, nested synchronization, as in:

  `sync` l_1 { ... `sync` l_2 { ... } ... }

  is `atomic`, but the sequential composition of synchronization operations, as in:

  `sync` l_1 { ... } ; `sync` l_2 { ... }

  is not `atomic`. By Lipton's theory of reduction [20], `atomic` expressions are serializable, and are therefore amenable to sequential reasoning techniques, which significantly facilitates subsequent formal and informal reasoning [14]. Our previous investigation showed that the vast majority of methods in multithreaded Java programs are atomic [10, 12].

- `cmpd` expressions do not follow the two-phase locking discipline, but which always hold the correct protecting lock when accessing any guarded field. Such expressions are not serializable and are therefore not amenable to sequential reasoning.

- `error` expressions violate the program's synchronization discipline by accessing a field without holding the guarding lock. Programs with `error` expressions do not type check.

Basic atomicities are ordered by the subatomicity relation $\sqsubset_b$:

$$\texttt{const} \sqsubset_b \texttt{mover} \sqsubset_b \texttt{atomic} \sqsubset_b \texttt{cmpd} \sqsubset_b \texttt{error}$$

Let $\sqcup_b$ denote the corresponding join operator for basic atomicities. Suppose that the basic atomicities b_1 and b_2 reflect the behavior of e_1 and e_2 respectively. Then:

- The atomicity $b_1 \sqcup_b b_2$ reflects the non-deterministic choice between executing either e_1 or e_2.

- The *sequential composition* $b_1; b_2$ reflects the behavior of executing $e_1; e_2$, and is defined as:

  $$b_1; b_2 \quad = \quad \begin{cases} \texttt{cmpd} & \text{if } b_1 = b_2 = \texttt{atomic} \\ b_1 \sqcup_b b_2 & \text{otherwise} \end{cases}$$

- The *iterative closure* ${b_1}^*$ reflects the behavior of executing e_1 an arbitrary number of times, and is defined as:

  $${b_1}^* \quad = \quad \begin{cases} \texttt{cmpd} & \text{if } b_1 = \texttt{atomic} \\ b_1 & \text{otherwise} \end{cases}$$

LEMMA 1. *The operations $b_1 \sqcup_b b_2$ and $b_1; b_2$ and ${b_1}^*$ are monotonic in both b_1 and b_2.*

4.2 Atomicity Expressions

We follow a constraint-based approach to atomicity inference and synchronization correction. For each method body, we generate an *atomicity expression* d that encodes the various operations performed by that method body. The syntax of atomicity expressions (see Figure 4) includes the following constructs:

- Basic atomicities.

Figure 4: Atomicity Expressions

$$
\begin{array}{rll}
d ::= & b \mid \alpha \mid d\mathbf{;}d \mid d\sqcup d \mid d^{*} \mid l\,\mathbf{?}\,d\,\mathbf{:}\,d & \text{(AtomExp)} \\
 & \mid \mathcal{S}(l,d) \mid \mathcal{R}(t,l,d) & \\
 & \mid d\cdot\theta \mid \mathit{wfa}(P,E,d) & \\
\theta ::= & [\vec{x} := \vec{l}\,] & \text{(substitution)}
\end{array}
$$

- Atomicity variables α, which support atomicity inference.

  An *assignment* maps atomicity variables to closed atomicity expressions (that is, to atomicity expressions that do not contain atomicity variables):

  $$
  \begin{array}{lcl}
  \alpha & \in & \mathit{AtomVar} \\
  A & \in & \mathit{Assignment} \;=\; \mathit{AtomVar} \rightarrow \mathit{ClosedAtomExp}
  \end{array}
  $$

- Sequential composition, join, and iterative closure of atomicity expressions (which correspond to sequential composition, branching, and looping operations in the original code).

- The *conditional atomicity* l **?** d_1 **:** d_2, which is equal to d_1 if the lock l is held, and is equal to d_2 otherwise. For example, the atomicity of an access to a field protected by lock l is l **?** `mover` **:** `error`, formalizing that an access to the field has atomicity `mover` if l is held, and has atomicity `error` otherwise.

- The atomicity expression $\mathcal{S}(l,d)$ yields the atomicity for a synchronized expression `sync` l e, where d is the atomicity expression for e.

- The construct $\mathcal{R}(t,l,d)$ is generated for each tagged synchronization operation `sync?`t l e. If the inferred tag set T includes the tag t, then this synchronization operation is chosen for insertion, and $\mathcal{R}(t,l,d)$ is equivalent to $\mathcal{S}(l,d)$. If the inferred tag set does not include t, then $\mathcal{R}(t,l,d)$ is equivalent to d. More formally, the application of a tag set T to an atomicity expression is the compatible closure of the following function:

  $$
  T(\mathcal{R}(t,l,d)) \quad = \quad \begin{cases} \mathcal{S}(l,d) & \text{if } t \in T \\ d & \text{otherwise} \end{cases}
  $$

- The *delayed substitution* operation $d\cdot\theta$ is used when changing scopes, where the substitution θ is a finite map from variables to lock expressions. For example, if a method's atomicity refers to a formal method parameter, then this formal parameter must be replaced by the corresponding actual parameter to derive the correct atomicity for a call to that method.

- The construct $\mathit{wfa}(P,E,d)$ yields the smallest (*i.e.*, most precise) atomicity that is at least as large as d and that only depends on locks that are in scope in the environment E. This construct is used when d may refer to a variable that is going out of scope, such as at the end of a `let` construct.

An atomicity expression is *closed* if it does not contain atomicity variables. An atomicity expression is *tag-free* if it does not contain the construct $\mathcal{R}(t,l,d)$.

Figure 5: Atomicity Meaning Function

$$
\begin{array}{rcl}
[\![\cdot]\!] & : & \mathit{ClosedAtomExp} \rightarrow \mathit{Atomicity} \\
[\![b]\!]_L & = & b \\
[\![d_1\mathbf{;}d_2]\!]_L & = & [\![d_1]\!]_L\,;[\![d_2]\!]_L \\
[\![d_1\sqcup d_2]\!]_L & = & [\![d_1]\!]_L \sqcup_a [\![d_2]\!]_L \\
[\![d^{*}]\!]_L & = & ([\![d]\!]_L)^{*} \\
[\![l\,\mathbf{?}\,d_1\,\mathbf{:}\,d_2]\!]_L & = & \begin{cases} [\![d_1]\!]_L & \text{if } l \in L \\ [\![d_2]\!]_L & \text{otherwise} \end{cases} \\
[\![\mathcal{S}(l,d)]\!]_L & = & \begin{cases} [\![d]\!]_L & \text{if } l \in L \\ [\![d]\!]_{L\cup\{l\}} & \text{if } \texttt{atomic} \sqsubseteq_a [\![d]\!]_{L\cup\{l\}} \\ \texttt{atomic} & \text{otherwise} \end{cases} \\
[\![d\cdot\theta]\!]_L & = & [\![d]\!]_{L'} \quad \text{where } L' = \{x \mid \theta(x) \in L\} \\
[\![\mathit{wfa}(P,E,d)]\!]_L & = & \text{(see Section 5)}
\end{array}
$$

4.3 Atomicities

We formalize the meaning of a closed atomicity expression as a map from the set of lock held by the current thread to a basic atomicity, and we refer to this map as an *atomicity.*

$$
\begin{array}{lcll}
L & \in & \mathit{LockSet} & = \; 2^{\mathit{Lock}} \\
a & \in & \mathit{Atomicity} & = \; \mathit{LockSet} \rightarrow \mathit{BasicAtomicity}
\end{array}
$$

The function $[\![\cdot]\!]$, defined in Figure 5, maps closed, tag-free atomicity expressions to atomicities. (For clarity, we write "$[\![d]\!]_L = \ldots$" to abbreviate "$[\![d]\!] = \lambda L.\ \ldots$").

We order atomicities according to the point-wise extension $\sqsubseteq_a$ of the ordering relation $\sqsubseteq_b$ on basic atomicities, with corresponding minimal element $\bot_a$ and join operation $\sqcup_a$. We also extend the sequential composition and iterative closure operations to atomicities in a point-wise manner.

$$
\begin{array}{rcl}
a_1 \sqsubseteq_a a_2 & \text{iff} & \forall L \subseteq \mathit{Lock}.\ (a_1(L) \sqsubseteq_b a_2(L)) \\
\bot_a & \stackrel{\text{def}}{=} & \lambda L.\ \texttt{const} \\
a_1 \sqcup_a a_2 & \stackrel{\text{def}}{=} & \lambda L.\ (a_1(L) \sqcup_b a_2(L)) \\
a^{*} & \stackrel{\text{def}}{=} & \lambda L.\ (a(L))^{*} \\
a_1 ; a_2 & \stackrel{\text{def}}{=} & \lambda L.\ (a_1(L); a_2(L))
\end{array}
$$

We order assignments according to the point-wise extension $\sqsubseteq_A$ of the ordering relation $\sqsubseteq_a$ on atomicities, with corresponding minimal element $\bot_A$ and join operation $\sqcup_A$:

$$
\begin{array}{rcl}
A_1 \sqsubseteq_A A_2 & \text{iff} & \forall \alpha.\ ([\![A_1(\alpha)]\!] \sqsubseteq_a [\![A_2(\alpha)]\!]) \\
A_1 =_A A_2 & \text{iff} & \forall \alpha.\ ([\![A_1(\alpha)]\!] = [\![A_2(\alpha)]\!]) \\
\bot_A & \stackrel{\text{def}}{=} & \lambda \alpha.\ \texttt{const} \\
A_1 \sqcup_A A_2 & \stackrel{\text{def}}{=} & \lambda \alpha.\ (A_1(\alpha) \sqcup A_2(\alpha))
\end{array}
$$

We extend assignments in a compatible manner to be maps from atomicity expressions to atomicity expressions. Given an A and a tag-free atomicity expression d, the atomicity expression $A(d)$ is then closed and tag-free, and its meaning is the atomicity $[\![A(d)]\!]$. Furthermore, this meaning function is monotonic in A (a necessary prerequisite for performing our least fixpoint analysis).

LEMMA 2 (MONOTONICITY). *For all tag-free atomicity expressions d, the function $\lambda A.\ [\![A(d)]\!]$ is monotonic.*

PROOF. By induction on the structure of d. □

4.4 Atomicity Constraints

A *syntactic atomicity* s is either an atomicity variable or a closed, tag-free atomicity expression. Each method declaration includes a corresponding syntactic atomicity. For example, a programmer could specify a precise atomicity for a method, such as `atomic`. More commonly, the programmer might omit this specification, in which case the type checker uses a fresh atomicity variable as the method's declared atomicity.

Given a program P, for each method with declared atomicity s and method body e, we generate an atomicity expression d for the method body e (as described in the following section) and produce the *constraint* $d \sqsubseteq s$. Thus, a constraint is a subatomicity relation between an atomicity expression and a syntactic atomicity. An assignment A *satisfies* a constraint $d \sqsubseteq s$ with respect to a tag set T (written $A; T \models C$) if

$$[\![A(T(d))]\!] \sqsubseteq_a [\![A(s)]\!]$$

Generating a constraint for each method yields a constraint set $\bar{C}$ for the entire program. The assignment A *satisfies* a set $\bar{C}$ with respect to T (written $A; T \models \bar{C}$) if A satisfies each constraint in $\bar{C}$. Thus, our goal is to find an assignment A and a tag set T such that A satisfies $\bar{C}$ with respect to T. The tag set T then specifies which tagged synchronization operations need to be inserted into the program in order to yield a corrected program that satisfies the desired atomicity specifications.

Since the type checker introduces a fresh atomicity variable for each unannotated method, each atomicity variable α annotates exactly one method, and so α has a unique lower bound d such that the constraint $d \sqsubseteq \alpha$ occurs in $\bar{C}$. We use the notation $\bar{C}(\alpha)$ to refer to this lower bound d. We refer to such constraints with a variable as an upper bound as *propagation constraints*, and we refer to all other constraints (with a closed atomicity expression as upper bound) as *checking constraints*.

5. PHASE 2: CONSTRAINT GENERATION

We express the algorithm for converting a program with previously-inserted tagged synchronization operations into a collection of constraints $\bar{C}$ as a set of rules for reasoning about the judgment:

$$P; E \vdash e : c \cdot d \cdot \bar{C}$$

Here, c is the type inferred for the expression e; d is the atomicity expression generated for e; and $\bar{C}$ is the set of constraints generated from this expression. The program P is included to provide access to class declarations, and E is an environment providing types for the free program and ghost variables of the expression e:

$$E ::= \epsilon \mid E, c\ x \mid E, \texttt{ghost}\ x$$

The complete set of type judgments and rules is contained in Figure 6. We briefly describe some of the more important rules.

[LOCK EXP] The judgment $P; E \vdash_{\text{lock}} l : \bar{C}$ checks that l is a well-formed lock expression in environment E. The lock expression l can be either a ghost variable or a program expression e. In the latter case, e must denote a fixed lock throughout the execution of the program to ensure soundness. Thus, we require that e has atomicity `const`.

In addition, we require the size $|e|$ of the lock expression to be bounded by the constant *MaxLockSize*. This requirement ensures that there is only a finite number of valid lock expressions at any program point, which in turn bounds the size of conditional atomicities and number of possible tagged synchronization statements. This ensures termination of our correction algorithm.

[EXP VAR] A variable access has `const` atomicity, since all variables are immutable in AJC.

[EXP IF] The atomicity of a conditional expression is the atomicity of the *test* subexpression, sequentially composed with the join of the atomicities of the *then* and *else* branches.

[EXP LET] This rule for `let` x `=` e_1 `in` e_2 infers atomicity expressions d_1 and d_2 for e_1 and e_2, respectively. Since the atomicity expression d_2 may refer to the let-bound variable x, we apply the substitution $\theta = [x := e_1]$ to yield a corresponding atomicity that does not mention x. Several complications arise here.

First, since d_2 may include an atomicity variable α, we cannot apply the substitution θ immediately because α may later resolve to x. Instead, we use the *delayed substitution form* $d_2 \cdot \theta$ to delay this substitution until after atomicity variables are resolved.

Second, e_1 may not be `const` (in general, we cannot determine which expressions are `const` until after type inference), in which case $d_2 \cdot \theta$ may not be a valid atomicity. Therefore, we use the well-formed atomicity construct $\mathit{wfa}(P, E, d_2 \cdot \theta)$ to yield a valid atomicity for e_2 that is well-formed in environment E. The meaning of this construct is defined via:

$$\begin{aligned} [\![\mathit{wfa}(P, E, d)]\!]_L &= [\![d]\!]_{L'} \\ \text{where } L' &= \left\{ l \in L \;\middle|\; \begin{array}{l} P; E \vdash_{\text{lock}} l : \bar{C} \\ \text{and } \bot_A; \emptyset \models \bar{C} \end{array} \right\} \end{aligned}$$

As described above, the judgment $P; E \vdash_{\text{lock}} l$ checks that l is a well-formed lock expression in an environment E and program P provided the constraints $\bar{C}$ holds. Since the meaning function $[\![\cdot]\!]_L$ is defined only on closed, tag-free atomicity expressions, $\bar{C}$ will also be closed and tag-free, and so we check that it satisfiable via $\bot_A; \emptyset \models \bar{C}$. Thus, L' contains only locks that are held and that are valid in the environment E, and d is evaluated in the context of these held and valid locks.

[EXP REF] The rule for a field access $e.\mathit{fn}$ first checks that e is of some type $cn\langle l_{1..n}\rangle$, and that cn is a class parameterized by n ghost variables, say $x_{1..n}$, that declares a field fn of some type t. The type t may refer to the variables `this` and $x_{1..n}$ in scope at the field declaration. Since these variables are not in scope at the field access, the type rule introduces a substitution θ that replaces them with the corresponding expressions e and $l_{1..n}$, and ensures that $\theta(t)$ is a well-formed type.

The [EXP REF] rule performs a case analysis on the field's guard. If the field is `final`, then the read operation has atomicity `const`, since there can be no concurrent writes. If the field is `unguarded`, then the read operation is `atomic`, since it may not commute with concurrent writes. If the field is `guarded_by` l, then the lock $\theta(l)$ must be held and the read operation is a `mover`.

Figure 6: AJC Constraint Generation Rules

$$\boxed{P; E \vdash e \mathbf{:} c \cdot d \cdot \bar{C}}$$

[EXP NULL]
$$\frac{P; E \vdash c : \bar{C}}{P; E \vdash \texttt{null} \mathbf{:} c \cdot \texttt{const} \cdot \bar{C}}$$

[EXP VAR]
$$\frac{\begin{array}{c} P \vdash E : \bar{C} \\ E = E_1, c\ x, E_2 \end{array}}{P; E \vdash x \mathbf{:} c \cdot \texttt{const} \cdot \bar{C}}$$

[EXP SYNC]
$$\frac{\begin{array}{c} P; E \vdash l \mathbf{:} c_l \cdot d_l \cdot \bar{C} \\ P; E \vdash e \mathbf{:} c \cdot d \cdot \bar{C}' \\ \bar{C}'' = \bar{C} \cup \bar{C}' \cup \{d_l \sqsubseteq \texttt{const}\} \end{array}}{P; E \vdash \texttt{sync}\ l\ e \mathbf{:} c \cdot \mathcal{S}(l, d) \cdot \bar{C}''}$$

[EXP SYNC-OPT]
$$\frac{\begin{array}{c} P; E \vdash l \mathbf{:} c_l \cdot d_l \cdot \bar{C} \\ P; E \vdash e \mathbf{:} c \cdot d \cdot \bar{C}' \\ \bar{C}'' = \bar{C} \cup \bar{C}' \cup \{d_l \sqsubseteq \texttt{const}\} \end{array}}{P; E \vdash \texttt{sync?}t\ l\ e \mathbf{:} c \cdot \mathcal{R}(t, l, d) \cdot \bar{C}''}$$

[EXP REF]
$$\frac{\begin{array}{c} P; E \vdash e \mathbf{:} cn\langle l_{1..n} \rangle \cdot d' \cdot \bar{C} \\ \texttt{class}\ cn\langle \texttt{ghost}\ x_{1..n} \rangle\ \{ \ldots c\ \mathit{fn}\ g \ldots \} \in P \\ \theta = [\texttt{this} := e, x_j := l_j\ ^{j \in 1..n}] \\ P; E \vdash \theta(c) : \bar{C}' \\ (g \equiv \texttt{guarded\_by}\ l) \quad \Rightarrow \quad (d = \theta(l)\,\mathbf{?}\,\texttt{mover}\,\mathbf{:}\,\texttt{error}) \\ (g \equiv \texttt{final}) \quad \Rightarrow \quad (d = \texttt{const}) \\ (g \equiv \texttt{unguarded}) \quad \Rightarrow \quad (d = \texttt{atomic}) \end{array}}{P; E \vdash e.\mathit{fn} \mathbf{:} \theta(c) \cdot (d' \mathbf{;} \mathit{wfa}(P, E, d)) \cdot (\bar{C} \cup \bar{C}')}$$

[EXP ASSIGN]
$$\frac{\begin{array}{c} P; E \vdash e_1 \mathbf{:} cn\langle l_{1..n} \rangle \cdot d_1 \cdot \bar{C}_1 \\ \texttt{class}\ cn\langle \texttt{ghost}\ x_{1..n} \rangle \{ \ldots c\ \mathit{fn}\ g \ldots \} \in P \\ \theta = [\texttt{this} := e_1, x_j := l_j\ ^{j \in 1..n}] \\ P; E \vdash e_2 \mathbf{:} \theta(c) \cdot d_2 \cdot \bar{C}_2 \\ (g \equiv \texttt{guarded\_by}\ l) \quad \Rightarrow \quad (d = \theta(l)\,\mathbf{?}\,\texttt{mover}\,\mathbf{:}\,\texttt{error}) \\ (g \equiv \texttt{final}) \quad \Rightarrow \quad (d = \texttt{error}) \\ (g \equiv \texttt{unguarded}) \quad \Rightarrow \quad (d = \texttt{atomic}) \end{array}}{P; E \vdash (e_1.\mathit{fn} = e_2) \mathbf{:} \theta(c) \cdot (d_1 \mathbf{;} d_2 \mathbf{;} \mathit{wfa}(P, E, d)) \cdot (\bar{C}_1 \cup \bar{C}_2)}$$

[EXP NEW]
$$\frac{\begin{array}{c} \theta = [x_j := l_j\ ^{j \in 1..n}, \texttt{this} := y] \\ P; E, \texttt{ghost}\ y \vdash e_i \mathbf{:} \theta(c_i) \cdot d_i \cdot \bar{C}_i \qquad \forall i \in 1..k \\ \texttt{class}\ cn\langle \texttt{ghost}\ x_{1..n} \rangle\ \{\ \mathit{field}_{1..k}\ \mathit{meth}_{1..m}\ \} \in P \\ \mathit{field}_i = c_i\ \mathit{fn}_i\ g_i \qquad \forall i \in 1..k \\ P; E \vdash cn\langle l_{1..n} \rangle : \bar{C}' \\ \bar{C}'' = \bar{C}_{1..k} \cup \bar{C}' \end{array}}{P; E \vdash \texttt{new}_y\ cn\langle l_{1..n} \rangle \texttt{(} e_{1..k} \texttt{)} \mathbf{:} cn\langle l_{1..n} \rangle \cdot (d_1 \mathbf{;} \cdots \mathbf{;} d_k) \cdot \bar{C}''}$$

[EXP INVOKE]
$$\frac{\begin{array}{c} P; E \vdash e \mathbf{:} cn\langle l_{1..n} \rangle \cdot d \cdot \bar{C} \\ \texttt{class}\ cn\langle \texttt{ghost}\ x_{1..n} \rangle\ \{ \ldots \mathit{meth} \ldots \} \in P \\ \mathit{meth} = s\ c\ mn\langle \texttt{ghost}\ y_{1..k} \rangle \texttt{(} c_j\ z_j^{\ j \in 1..r} \texttt{)}\ \{\ e'\ \} \\ \theta = [\ \texttt{this} := e,\ x_i := l_i\ ^{i \in 1..n}, y_i := l'_i\ ^{i \in 1..k},\ z_i := e_i\ ^{i \in 1..r}\ ] \\ P; E \vdash e_j \mathbf{:} \theta(c_j) \cdot d_j \cdot \bar{C}_j \qquad \forall j \in 1..r \\ P; E \vdash \theta(t) : \bar{C}' \\ P; E \vdash_{\text{lock}} l'_i \mathbf{:} \bar{C}'_i \qquad \forall i \in 1..k \\ d' = (d \mathbf{;} d_1 \mathbf{;} \cdots \mathbf{;} d_r \mathbf{;} \mathit{wfa}(P, E, s \bullet \theta))) \end{array}}{P; E \vdash e.mn\langle l'_{1..k} \rangle \texttt{(} e_{1..r} \texttt{)} \mathbf{:} \theta(c) \cdot d' \cdot (\bar{C} \cup \bar{C}_{1..r} \cup \bar{C}' \cup \bar{C}'_{1..k})}$$

[EXP WHILE]
$$\frac{\begin{array}{c} P; E \vdash e_i \mathbf{:} c_i \cdot d_i \cdot \bar{C}_i \qquad \text{for } i = 1, 2 \\ d \ = \ d_1 \mathbf{;} ((d_2 \mathbf{;} d_1)^{*}) \end{array}}{P; E \vdash \texttt{while}\ e_1\ e_2 \mathbf{:} c_2 \cdot d \cdot (\bar{C}_1 \cup \bar{C}_2)}$$

[EXP IF]
$$\frac{\begin{array}{c} P; E \vdash e_1 \mathbf{:} c_1 \cdot d_1 \cdot \bar{C}_1 \\ P; E \vdash e_i \mathbf{:} c \cdot d_i \cdot \bar{C}_i \qquad \text{for } i = 2..3 \\ d \ = \ d_1 \mathbf{;} (d_2 \sqcup d_3) \\ \bar{C}' = (\bar{C}_1 \cup \bar{C}_2 \cup \bar{C}_3) \end{array}}{P; E \vdash \texttt{if}\ e_1\ e_2\ e_3 \mathbf{:} c \cdot d \cdot \bar{C}'}$$

[EXP FORK]
$$\frac{\begin{array}{c} P; E \vdash e \mathbf{:} cn\langle l_{1..n} \rangle \cdot d \cdot \bar{C} \\ \texttt{class}\ cn\langle \texttt{ghost}\ x_{1..n} \rangle\ \{ \ldots \mathit{meth} \ldots \} \in P \\ \mathit{meth} = a\ c'\ \texttt{run}\langle \texttt{ghost tll} \rangle \texttt{()}\ \{\ e'\ \} \\ a = (\texttt{tll}\,\mathbf{?}\,\texttt{cmpd}\,\mathbf{:}\,\texttt{error}) \\ P; E \vdash c : \bar{C}' \end{array}}{P; E \vdash e.\texttt{fork} \mathbf{:} c \cdot (d \mathbf{;} \texttt{atomic}) \cdot (\bar{C} \cup \bar{C}')}$$

[EXP LET]
$$\frac{\begin{array}{c} P; E \vdash e_1 \mathbf{:} c_1 \cdot d_1 \cdot \bar{C}_1 \\ P; E, t_1\ x \vdash e_2 \mathbf{:} c_2 \cdot d_2 \cdot \bar{C}_2 \\ \theta = [x := e_1] \\ P; E \vdash \theta(c_2) : \bar{C}_3 \\ d = (d_1 \mathbf{;} \mathit{wfa}(P, E, d_2 \bullet \theta)) \end{array}}{P; E \vdash \texttt{let}\ c_1\ x\ \texttt{=}\ e_1\ \texttt{in}\ e_2 \mathbf{:} \theta(c_2) \cdot d \cdot (\bar{C}_1 \cup \bar{C}_2 \cup \bar{C}_3)}$$

$$\boxed{P \vdash \mathit{defn} : \bar{C}}$$

[CLASS]
$$\frac{\begin{array}{c} \mathit{garg}_i = \texttt{ghost}\ x_i \\ E = \mathit{garg}_{1..n}, cn\langle x_{1..n} \rangle\ \texttt{this} \\ P; E \vdash \mathit{field}_i : \bar{C}_i \qquad \forall i \in 1..j \\ P; E \vdash \mathit{meth}_i : \bar{C}'_i \qquad \forall i \in 1..k \end{array}}{P \vdash \texttt{class}\ cn\langle \texttt{ghost}\ x_{1..n} \rangle\ \{\ \mathit{field}_{1..j}\ \ \mathit{meth}_{1..k} \} : (\bar{C}_{1..j} \cup \bar{C}'_{1..k})}$$

$$\boxed{P; E \vdash_{\text{lock}} e \mathbf{:} \bar{C}}$$

[LOCK EXP]
$$\frac{P; E \vdash e \mathbf{:} c \cdot d \cdot \bar{C} \qquad |e| \leq \mathit{MaxLockSize}}{P; E \vdash_{\text{lock}} e \mathbf{:} (\bar{C} \cup \{d \sqsubseteq \texttt{const}\})}$$

[LOCK GHOST]
$$\frac{P \vdash E : \bar{C} \qquad E = E_1, \texttt{ghost}\ x, E_2}{P; E \vdash_{\text{lock}} x \mathbf{:} \bar{C}}$$

$$\boxed{P; E \vdash t : \bar{C}}$$

[TYPE C]
$$\frac{\begin{array}{c} P \vdash E : \bar{C} \\ \texttt{class}\ cn\langle \texttt{ghost}\ x_{1..n} \rangle\ \ldots \in P \\ P; E \vdash_{\text{lock}} l_i \mathbf{:} \bar{C}_i \qquad \forall i \in 1..n \end{array}}{P; E \vdash cn\langle l_{1..n} \rangle : (\bar{C} \cup \bar{C}_{1..n})}$$

$$\boxed{P \vdash E : \bar{C}}$$

[ENV EMPTY]
$$\frac{}{P \vdash \epsilon : \emptyset}$$

[ENV X]
$$\frac{P; E \vdash c : \bar{C} \qquad x \notin \mathit{Dom}(E)}{P \vdash (E, c\ x) : \bar{C}}$$

[ENV GHOST]
$$\frac{P \vdash E : \bar{C} \qquad x \notin \mathit{Dom}(E)}{P \vdash (E, \texttt{ghost}\ x) : \bar{C}}$$

$$\boxed{P; E \vdash \mathit{field} : \bar{C}}$$

[FIELD]
$$\frac{\begin{array}{c} P; E \vdash c : \bar{C}_1 \\ (g \equiv \texttt{guarded\_by}\ l) \ \Rightarrow \ P; E \vdash_{\text{lock}} l \mathbf{:} \bar{C}_2 \\ (g \equiv \texttt{final}) \ \Rightarrow \ \bar{C}_2 = \emptyset \\ (g \equiv \texttt{unguarded}) \ \Rightarrow \ \bar{C}_2 = \emptyset \end{array}}{P; E \vdash c\ \mathit{fn}\ g : (\bar{C}_1 \cup \bar{C}_2)}$$

$$\boxed{P; E \vdash \mathit{meth} : \bar{C}}$$

[METHOD]
$$\frac{\begin{array}{c} \mathit{meth} = s\ c\ mn\langle \texttt{ghost}\ x_{1..n} \rangle \texttt{(} \mathit{arg}_{1..d} \texttt{)}\ \{\ e\ \} \\ \mathit{garg}_i = \texttt{ghost}\ x_i \qquad \forall i \in 1..n \\ E' = E, \mathit{garg}_{1..n}, \mathit{arg}_{1..d} \\ P; E' \vdash e \mathbf{:} c \cdot d \cdot \bar{C} \end{array}}{P; E \vdash \mathit{meth} : (\bar{C} \cup \{d \sqsubseteq s\})}$$

$$\boxed{P \vdash \bar{C}}$$

[PROG]
$$\frac{\begin{array}{c} \mathit{ClassOnce}(P) \quad \mathit{FieldsOnce}(P) \\ \mathit{MethodsOnce}(P) \\ P = \mathit{defn}_{1..n}\ e \\ P \vdash \mathit{defn}_i : \bar{C}_i \qquad \forall i \in 1..n \\ P; \epsilon \vdash e \mathbf{:} c \cdot d \cdot \bar{C} \end{array}}{P \vdash \bar{C}_{1..n} \cup \bar{C} \cup \{d \sqsubseteq \texttt{cmpd}\}}$$

Figure 7: Constraints for Stack Program

$$
\begin{array}{rll}
\mathcal{R}(\texttt{t1},\texttt{this},(\texttt{const};(\texttt{const};(\texttt{const};\mathit{wfa}(P,E_1,\texttt{this}\,?\,\texttt{mover}:\texttt{error})));\mathit{wfa}(P,E_1,\texttt{this}\,?\,\texttt{mover}:\texttt{error}))) & \sqsubseteq & \alpha_1 \\
\mathcal{R}(\texttt{t2},\texttt{this},\mathcal{S}(\texttt{this},\mathcal{R}(\texttt{t3},\texttt{this},\texttt{const};\mathit{wfa}(P,E_2,\texttt{this}\,?\,\texttt{mover}:\texttt{error});\mathit{wfa}(P,E_2,\texttt{this}\,?\,\texttt{mover}:\texttt{error}); & & \\
\mathit{wfa}(P,E_2,(\mathcal{R}(\texttt{t4},\texttt{this},\mathcal{R}(\texttt{t5},\texttt{this},\texttt{const};\texttt{const};\mathit{wfa}(P,E_3,\texttt{this}\,?\,\texttt{mover}:\texttt{error}); & & \\
\mathit{wfa}(P,E_3,\texttt{this}\,?\,\texttt{mover}:\texttt{error}); & & \\
\mathit{wfa}(P,E_3,\texttt{this}\,?\,\texttt{mover}:\texttt{error})); & & \\
\mathcal{R}(\texttt{t6},\texttt{this},\texttt{const}))\cdot\theta_1))))) & \sqsubseteq & \alpha_2 \\
\mathcal{R}(\texttt{t7},\texttt{this},\mathcal{R}(\texttt{t8},\texttt{this.data},(\texttt{const};\texttt{const});\texttt{const};\mathit{wfa}(P,E_4,\alpha_1\cdot\theta_2))) & \sqsubseteq & \texttt{atomic} \\
\mathcal{R}(\texttt{t9},\texttt{this},\mathcal{R}(\texttt{t10},\texttt{this.data},(\texttt{const};\texttt{const};\mathit{wfa}(P,E_5,\alpha_2\cdot\theta_3)); & & \\
\mathit{wfa}(P,E_5,\mathcal{R}(\texttt{t11},\texttt{this},\mathcal{R}(\texttt{t12},\texttt{this.data}, & & \\
(\mathcal{R}(\texttt{t13},\texttt{this},\mathcal{R}(\texttt{t14},\texttt{this.data},(\texttt{const};\texttt{const});\texttt{const};\mathit{wfa}(P,E_6,\alpha_1\cdot\theta_2))); & & \\
\mathcal{R}(\texttt{t15},\texttt{this},\mathcal{R}(\texttt{t16},\texttt{this.data},(\texttt{const};\texttt{const});\texttt{const};\mathit{wfa}(P,E_6,\alpha_1\cdot\theta_2))))))\cdot\theta_4))) & \sqsubseteq & \texttt{atomic}
\end{array}
$$

$$
\begin{array}{lcl@{\qquad}lcl@{\qquad}lcl}
E_1 & = & \texttt{List this, int v} & E_4 & = & \texttt{Stack this, int v} & \theta_1 & = & [\texttt{x} := \texttt{this.elems.num}] \\
E_2 & = & \texttt{List this} & E_5 & = & \texttt{Stack this} & \theta_2 & = & [\texttt{this} := \texttt{this.data}, \texttt{v} := \texttt{x}] \\
E_3 & = & \texttt{List this, int x} & E_6 & = & \texttt{Stack this, int x} & \theta_3 & = & [\texttt{this} := \texttt{this.data}] \\
 & & & & & & \theta_4 & = & [\texttt{x} := \texttt{this.data.removeFirst()}]
\end{array}
$$

[EXP SYNC] The rule for the synchronized statement `sync` l e checks that l has atomicity `const`, and so always denotes the same lock. The rule then yields the atomicity expression $\mathcal{S}(l,d)$, where d is the atomicity of e. The meaning $[\![\mathcal{S}(l,d)]\!]_L$ of this atomicity expression is either (1) $[\![d]\!]_L$, if the lock is already held; (2) $[\![d]\!]_{L\cup\{l\}}$, if d is non-atomic; or (3) `atomic` otherwise.

[EXP SYNC-OPT] The rule for the tagged synchronized statement `sync?`t l e is similar, and yields the atomicity expression $\mathcal{R}(t,l,d)$, which either behaves like $\mathcal{S}(l,d)$ or d, depending on whether this synchronization statement is enabled or disabled by the tag set T.

[PROG] This rule defines the top-level judgment $P \vdash \bar{C}$, where $\bar{C}$ is the generated set of constraints for the program P. This rule uses three predicates defined as follows. (See [15] for their precise definition.)

- *ClassOnce*(P): no class is declared twice in P.
- *FieldsOnce*(P): no field name is declared twice in a class.
- *MethodsOnce*(P): no method name is declared twice in a class.

The constraints for our example program are shown in Figure 7. (For simplicity, we omit several trivial checking constraints.) The first constraint is generated for the method `add`. The various `const` atomicities in that constraint correspond to variable accesses; the two conditional atomicities `this ? mover : error` correspond to the access and update of `this.elems`, which is guarded by `this`; the enclosing $\mathcal{R}(\texttt{t1},\texttt{this},\dots)$ corresponds to the tagged synchronization construct `sync?t1 (this) ...`; and the upper bound α_1 is the atomicity specification for `add`. The third constraint is for `push`, and includes a reference to the atomicity specification α_1 of the callee `add`, with a substitution θ_2 that maps formal to actual parameters for this call site. The remaining constraints are similar, although more verbose.

6. PHASE 3: CONSTRAINT SOLVING

Having generated a constraint set $\bar{C}$ over the tags and atomicity variables in the program, the next step is to solve these constraints. If the program does not contain any tagged synchronization operations, then the generated constraints are tag-free, and our earlier work presents an algorithm for solving such constraints [12]. In this paper, we now tackle the harder problem of solving a constraint set $\bar{C}$ in the case where some constraints may include the tagged synchronization construct $\mathcal{R}(t,l,d)$, which corresponds to an automatically-inserted tagged synchronization operation `sync?`t l e in the source program.

The behavior of these tagged synchronization constructs depends on the chosen tag set T. If $t \in T$, then a real synchronization operation is inserted into the original program at this point, and $\mathcal{R}(t,l,d)$ behaves exactly like $\mathcal{S}(l,d)$. If $t \notin T$, then this potential synchronization point is ignored, and $\mathcal{R}(t,l,d)$ behave like d.

We wish to find a suitable choice of tag set T and assignment A such that $A; T \models \bar{C}$. We use an iterative least fixed point algorithm to compute the minimal assignment A that satisfies $\bar{C}$, but this assignment will of course depend on the chosen tag set. Hence, on each iteration of the algorithm, we choose the tag set that yields the smallest possible assignment for use in the next iteration.

Choosing a tag set in this manner is possible if atomicity expressions are *well-formed*. An atomicity expression d is well-formed if (1) each tag occurs at most once in d, and (2) whenever d contains a conditional atomicity $l\,?\,d_1 : d_2$ then

1. $[\![d_1]\!] \sqsubseteq [\![d_2]\!]$,
2. `atomic` $\sqsubseteq [\![d_2]\!]$, and
3. d_1 and d_2 mention the same lock expressions and atomicity variables.

The constraint generation rules only generate well-formed atomicity expressions, and well-formedness is preserved by the various operations we perform on atomicity expressions.

The function *min*(d), defined in Figure 8 returns a tag set T that minimizes $[\![d]\!]$, where d is a closed, well-formed atomicity expression.

LEMMA 3. *Suppose that d is closed and well-formed, and let $T = min(d)$. Then T only contains tags that occur in d, and for all T', $[\![T(d)]\!] \sqsubseteq_a [\![T'(d)]\!]$.*

PROOF. By structural induction on d. □

Figure 8: Tag Minimization Function

$$\begin{array}{rcl}
\mathit{min} & : & \mathit{ClosedAtomExpr} \to 2^{\mathit{Tag}} \\
\mathit{min}(b) & = & \emptyset \\
\mathit{min}(d_1\mathbf{;}d_2) & = & \mathit{min}(d_1) \cup \mathit{min}(d_2) \\
\mathit{min}(d_1\sqcup d_2) & = & \mathit{min}(d_1) \cup \mathit{min}(d_2) \\
\mathit{min}(d^*) & = & \mathit{min}(d) \\
\mathit{min}(l\,\mathbf{?}\,d_1\mathbf{:}\,d_2) & = & \mathit{min}(d_1) \cup \mathit{min}(d_2) \\
\mathit{min}(d\cdot\theta) & = & \mathit{min}(d) \\
\mathit{min}(\mathcal{S}(l,d)) & = & \mathit{min}(d) \\
\mathit{min}(\mathit{wfa}(P,E,d)) & = & \mathit{min}(d) \\
\mathit{min}(\mathcal{R}(t,l,d)) & = & \begin{cases} T \cup \{t\} & \text{if } [\![T(d)]\!] \text{ depends on } l \\ T & \text{otherwise} \end{cases} \\
 & & \text{where } T = \mathit{min}(d)
\end{array}$$

The following function $f_{\bar{C}}$ describes each iteration of our algorithm. For each variable α, the function $f_{\bar{C}}(A)$ computes the closed atomicity expression $d = A(\bar{C}(\alpha))$, computes the tag set T that minimizes d, and then returns this minimal atomicity $[\![T(d)]\!]$:

$$\begin{array}{rcl}
f_{\bar{C}} & : & \mathit{Assignment} \to \mathit{Assignment} \\
f_{\bar{C}}(A) & = & \lambda\alpha.\ [\![T(d)]\!], \text{ where } d\ = A(\bar{C}(\alpha)) \\
 & & \qquad\qquad\quad \text{and } T = \mathit{min}(d)
\end{array}$$

Suppose A is the least fixpoint of $f_{\bar{C}}$, that is, $A = \mathit{fix}(f_{\bar{C}}, \bot_A)$, where we define the fixpoint operator to terminate once it reaches assignments that are semantically equivalent with respect to $=_A$:

$$\mathit{fix}(F, X) \overset{\text{def}}{=} \text{if } X =_A F(X) \text{ then } X \text{ else } \mathit{fix}(F, F(X))$$

The tag set T defined by

$$T \quad = \quad \cup\{\mathit{min}(A(d)) \mid (d \sqsubseteq s) \in \bar{C}\}$$

minimizes all atomicity expressions in $\bar{C}$. Now consider any propagation constraint $d \sqsubseteq \alpha$ in $\bar{C}$. We have that

$$A(\alpha) = f_{\bar{C}}(A)(\alpha) = [\![T(A(\bar{C}(\alpha)))]\!] = [\![T(A(d))]\!]$$

and so $A; T \models d \sqsubseteq \alpha$. Thus, A satisfies all propagation constraints in $\bar{C}$ with respect to T, and it simply remains to check if A also satisfies the checking constraints in $\bar{C}$. The following function *solve* performs this analysis:

$$\mathit{solve}(\bar{C}) = \begin{cases} \langle A, T\rangle & \text{if } A = \mathit{fix}(f_{\bar{C}}, \bot_A) \\ & \wedge\ A; T \models \bar{C} \\ & \wedge\ T = \cup\{\mathit{min}(A(d)) \mid (d \sqsubseteq s) \in \bar{C}\} \\ & \\ \texttt{undef} & \text{otherwise} \end{cases}$$

Lemma 4.

1. *If* $\mathit{solve}(\bar{C}) = \texttt{undef}$ *then* $\bar{C}$ *is unsatisfiable.*

2. *If* $\mathit{solve}(\bar{C}) = \langle A, T\rangle$ *then* $A; T \models \bar{C}$.

Proving that the *solve* algorithm terminates is non-trivial, because delayed substitutions could lead to arbitrarily large lock expressions and infinite ascending chains of atomicities and assignments. We bound the size of lock expressions to exclude this possibility. A lock expression l is *bounded* if $|l| < \mathit{MaxLockSize}$. Similarly, an atomicity is *bounded* if it only contains bounded lock expressions, and an assignment is *bounded* if it only yields bounded atomicities. An atomicity expression or constraint is *bounded* if it is only conditional on bounded lock expressions, and every delayed substitution occurs inside the construct $\mathit{wfa}(P, E, \cdot)$. The *solve* algorithm terminates on the bounded constraint systems produced by the constraint generation rules.

Theorem 5 (Termination). *The constraint solving algorithm terminates on bounded constraint systems.*

Proof. See [12]. □

The algorithm computes the following solution for our Stack example:

$$\begin{array}{rcl}
A(\alpha_1) & = & \texttt{this}\,\mathbf{?}\,\texttt{mover}\,\mathbf{:}\,\texttt{atomic} \\
A(\alpha_2) & = & \texttt{this}\,\mathbf{?}\,\texttt{mover}\,\mathbf{:}\,\texttt{atomic} \\
T & = & \{\texttt{t1}, \texttt{t2}, \texttt{t3}, \texttt{t4}, \texttt{t5}, \texttt{t8}, \texttt{t10}, \texttt{t12}, \texttt{t14}, \texttt{t16}\}
\end{array}$$

In particular, the algorithm concludes that the tagged synchronization operations `t6`, `t7`, `t9`, `t11`, `t13`, and `t15` do not decrease the atomicity of their containing methods, and so do not contribute to yielding a correctly-synchronized program. These omitted tagged synchronization operations are grayed-out in Figure 2(a). The remaining tagged synchronization operations in T yield a correctly synchronized program, but incur an unnecessarily-large synchronization overhead, since many of them are redundant. The next section describes how to eliminate the redundant operations.

7. PHASE 4: SYNCHRONIZATION OPTIMIZATION

Having computed a tag set T and assignment A such that $\langle A, T\rangle = \mathit{solve}(\bar{C})$, it remains to relax the synchronization (by reducing T) while preserving the satisfiability of the constraints. For this purpose, we use a greedy algorithm, where the tags T are ordered by some heuristic into a worklist Z and are iteratively removed:

Figure 9: Phase 4 (version 1)

```
$Z := T$;
foreach $z \in Z$ do
    $T' := T \setminus \{z\}$;
    if $T'(\bar{C})$ is satisfiable then $T := T'$;
end foreach
```

The correctness of this basic algorithm is fairly straightforward to verify. However, the algorithm performs a lot of redundant computation in checking the satisfiability of $\bar{C}$ for various tag sets. As a first step towards optimizing this algorithm, we inline some of the operations being performed by this routine:

Figure 10: Phase 4 (version 2)

```
$Z := T$;
foreach $z \in Z$ do
    $T' := T \setminus \{z\}$;
    $\bar{D} := T'(\bar{C})$;
    $A' := \mathit{fix}(f_{\bar{D}}, \bot_A)$;
    if $\forall (d \sqsubseteq s) \in \bar{D}.\ [\![A'(d)]\!] \sqsubseteq_a [\![s]\!]$ then $T := T'$;
end foreach
```

It is now clear that the main performance overhead is in the repeated iterative fixpoint computation of $\mathit{fix}(f_{\bar{D}}, \bot_A)$, which always begins with the least assignment $\bot_A$. However, there is a key monotonicity property that we can exploit. By maintaining an assignment A such that $A =$

$\mathit{fix}(f_{T(\bar{C})}, \bot_A)$, we can more efficiently compute the fixpoint $\mathit{fix}(f_{T'(\bar{C})}, \bot_A)$, where $T' \subseteq T$, by starting from the current assignment A rather than the minimal assignment $\bot_A$. The following lemma ensures the correctness of this optimization.

Lemma 6. *Suppose*

$$\begin{array}{rcl@{\qquad\qquad}rcl} \langle A_0, T_0 \rangle &=& \mathit{solve}(\bar{C}) & T' &=& T \setminus \{z\} \\ A_0 &\sqsubseteq_A& A & A &=& \mathit{fix}(f_{T(\bar{C})}, \bot_A) \\ T &\subseteq& T_0 & \bar{D} &=& T'(\bar{C}) \end{array}$$

Then $A \sqsubseteq_A \mathit{fix}(f_{\bar{D}}, \bot_A)$ *and* $\mathit{fix}(f_{\bar{D}}, \bot_A) =_A \mathit{fix}(f_{\bar{D}}, A)$.

Our optimized algorithm is then:

Figure 11: Phase 4 (version 3)

```
$Z := T$;
foreach $z \in Z$ do
    // invariant: $A = \mathit{fix}(f_{T(\bar{C})}, \bot_A)$;
    $T' := T \setminus \{z\}$;
    $\bar{D} := T'(\bar{C})$;
    $A' := \mathit{fix}(f_{\bar{D}}, A)$;
    if $\forall (d \sqsubseteq a) \in \bar{D}.\ \llbracket A'(d) \rrbracket \sqsubseteq_a \llbracket a \rrbracket$ then
        $A := A'$;
        $T := T'$;
    end if
end foreach
```

This algorithm computes the following assignment and tag set for the Stack program.

$$\begin{array}{rcl} A(\alpha_1) &=& \texttt{this?mover:atomic} \\ A(\alpha_2) &=& \texttt{this?mover:atomic} \\ T &=& \{\texttt{t1}, \texttt{t10}\} \end{array}$$

All redundant synchronization operations are now eliminated, and Figure 2(b) shows the code with the two synchronization operations that fix the program. Different orderings for the work list may yield different results. For example, the following is also a solution:

$$\begin{array}{rcl} A(\alpha_1) &=& \texttt{this?mover:error} \\ A(\alpha_2) &=& \texttt{this?mover:atomic} \\ T &=& \{\texttt{t8}, \texttt{t10}\} \end{array}$$

One interesting aspect of this algorithm is that we can order the work list to achieve specific effects. Trying to remove tags for outermost synchronization statements first may help reduce the size of critical sections, whereas removing tags for innermost statements first may improve performance by reducing the number of synchronization operations.

8. IMPLEMENTATION

We have implemented synchronization correction in *Bohr*, our tool for detecting and correcting errors in Java programs [12]. *Bohr* takes as input source code that may optionally contain atomicity specifications in stylized comments starting with "#", as in "/*# mover */". *Bohr* runs in two phases. The first phase uses the *Rcc/Sat* tool to infer appropriate guards for each field and appropriate formal and actual ghost parameters for class and method declarations and uses. *Rcc/Sat* is somewhat resilient to errors and will infer the most likely synchronization information, even if a small number of race conditions do exist. For more details on *Rcc/Sat*, we refer the interested reader to our earlier paper [11].

Our present work is included in the second phase of *Bohr*, which computes any unspecified atomicities and inserts synchronization operations to correct errors. *Bohr* outputs a fully annotated version of the source code, including any synchronization statements necessary to satisfy the specified atomicity requirements. If errors cannot be fixed by our algorithm, the checker prints warning messages for each atomicity violation identified.

We applied *Bohr* to three standard Java 1.4.2 library classes to validate its effectiveness at fixing defects. These three classes are designed to be thread-safe, meaning that all public methods should be `atomic`. The following table summarizes the results of running *Bohr* with and without synchronization correction:

Class	Lines	Atomicity Warnings	
		No Correction	With Correction
`String`	2,307	1	0
`StringBuffer`	1,276	1	0
`Vector`	3,546	3	0

The "Lines" column includes the size of the class of interest and all superclasses. We annotated all referenced library classes with appropriate atomicities, and, for simplicity, we assumed that all subclasses of `Collection` are internally synchronized. *Bohr* successfully corrected for the warnings reported in our previous work [12]. The `StringBuffer` and `Vector` warnings are real defects. The `String` warning is caused by benign races and is spurious, although *Bohr* did add synchronization to remove the races.

To further assess the effectiveness of *Bohr*, we performed an experiment on Java 1.4.2 library classes in which we added atomicity specifications and then randomly removed a small number of synchronization operations and measured how many operations *Bohr* would correctly reinsert. We removed one, two, or five synchronization operations at a time, and repeated each scenario multiple times. The following table summarizes the percentage of inserted defects that were identified and fixed:

Class	Lines	Time	Correction Rate per Number of Defects		
		(sec)	1	2	5
`Observable`	198	0.73	100%	100%	100%
`Inflater`	319	0.60	100%	97%	91%
`Deflater`	384	0.90	100%	95%	88%
`Zipfile`	498	25.0	100%	100%	90%
`StringBuffer`	1,276	41.8	100%	88%	49%
`String`	2,307	26.2	100%	100%	—
`Vector`	3,456	49.6	100%	100%	85%
`SynchronizedList`	3,837	13.8	94%	85%	85%

No result is reported for `String` with five defects since that class has fewer than five synchronization operations. *Bohr* performed well when a small number of defects were introduced. For the larger numbers of defects, the precision declined because *Rcc/Sat* did not always infer the appropriate locking discipline, due to the large number of data races introduced on specific object fields. *Rcc/Sat* can be adjusted to withstand a higher number, but the mere fact that a large number of races exists is often an indication of a fundamental problem with the design of the code (as opposed to more local programming errors). Interestingly, *Bohr* determined that several synchronization operations in these classes are redundant and unnecessary.

Results from applying *Bohr* to small, complete programs are similarly promising. The application of our current implementation to significantly larger programs is limited by the lack of precise atomicity specifications for libraries, and by incomplete support in our current implementation for some synchronization idioms such as protecting locks [12].

9. RELATED WORK

Since an atomicity annotation describes aspects of the behavior or effect of an expression, we are essentially performing a form of effect reconstruction [27, 26]. Our work differs from most of the work on effect systems and dependent types [6] by investigating automated techniques for correcting errors. Sasturkar *et al* [22] have also developed a type inference algorithm for atomicity. Unlike *Bohr*, their system includes a notion of object ownership [4] and uses a dynamic analysis to infer race condition information. They have not explored synchronization correction.

Lipton [20] first proposed reduction as a way to reason about deadlocks without considering all possible interleavings. Partial-order reduction techniques are based on similar ideas [16]. Several papers have have used Lipton's theory of reduction to improve the efficiency of model checking [5, 25, 13].

The use of model checking for verifying atomicity is being explored by Hatcliff *et al* [19]. This model checking approach is more expressive than our type-based analysis, but it is vulnerable to state-space explosion. Their results suggest that verifying atomicity via model-checking is feasible for unit-testing. A more general (but more expensive) technique for verifying atomicity during model checking is *commit-atomicity* [8]. Several tools have explored verifying atomicity dynamically [10, 31], but these tools are sensitive to test case coverage.

View consistency is a another approach to preventing threads interference [2, 30]. A view is the set of variables accessed within a synchronized block. Thread A is view consistent with B if all views from the execution of A, intersected with the maximal view of B, are ordered by subset inclusion. We believe view consistency could be extended with an analogous idea of synchronization correction.

Recent approaches to supporting atomicity include lightweight transactions [18, 32, 21, 17] and automatic generation of synchronization code from high-level specifications [7]. Lightweight transactions in particular seem like a promising, and complementary, approached to providing atomicity guarantees. An interesting avenue of future work is to explore how to best merge synchronization-based and transaction-based approaches.

10. CONCLUSIONS

Synchronization errors are a common source of defects in software systems. Our synchronization correction algorithm enables us to not only identify concurrency errors statically, but also correct them in many situations. Preliminary experiments demonstrate the effectiveness of our approach at correcting existing and randomly-inserted defects. We hope to further validate our approach on larger programs. This would also give us the opportunity to explore different optimization approaches in the final step of the algorithm. In addition, we plan to integrate a deadlock detection analysis into *Bohr* so that the tool can avoid introducing any potential for deadlock when inserting synchronization operations.

11. ACKNOWLEDGMENTS

This work was supported by the National Science Foundation under Grants CCR-0341179 and CCR-0341387.

12. REFERENCES

[1] R. Agarwal and S. D. Stoller. Type inference for parameterized race-free Java. In *Proceedings of the Conference on Verification, Model Checking, and Abstract Interpretation*, pages 149–160, 2004.

[2] C. Artho, K. Havelund, and A. Biere. High-level data races. In *The First International Workshop on Verification and Validation of Enterprise Information Systems*, 2003.

[3] A. D. Birrell. An introduction to programming with threads. Research Report 35, Digital Equipment Corporation Systems Research Center, 1989.

[4] C. Boyapati and M. Rinard. A parameterized type system for race-free Java programs. In *Proceedings of the ACM Conference on Object-Oriented Programming, Systems, Languages and Applications*, pages 56–69, 2001.

[5] D. Bruening. Systematic testing of multithreaded Java programs. Master's thesis, Massachusetts Institute of Technology, 1999.

[6] L. Cardelli. Typechecking dependent types and subtypes. In *Lecture Notes in Computer Science on Foundations of logic and functional programming*, pages 45–57, 1988.

[7] X. Deng, M. Dwyer, J. Hatcliff, and M. Mizuno. Invariant-based specification, synthesis, and verification of synchronization in concurrent programs. In *International Conference on Software Engineering*, pages 442–452, 2002.

[8] C. Flanagan. Verifying commit-atomicity using model-checking. In *Proceedings of the International SPIN Workshop on Model Checking of Software*, pages 252–266, 2004.

[9] C. Flanagan and S. N. Freund. Type-based race detection for Java. In *Proceedings of the ACM Conference on Programming Language Design and Implementation*, pages 219–232, 2000.

[10] C. Flanagan and S. N. Freund. Atomizer: A dynamic atomicity checker for multithreaded programs. In *Proceedings of the ACM Symposium on the Principles of Programming Languages*, pages 256–267, 2004.

[11] C. Flanagan and S. N. Freund. Type inference against races. In *Proceedings of the Static Analysis Symposium*, pages 116–132, 2004.

[12] C. Flanagan, S. N. Freund, and M. Lifshin. Type inference for atomicity. In *Proceedings of the ACM Workshop on Types in Language Design and Implementation*, pages 47–58, 2005.

[13] C. Flanagan and S. Qadeer. Transactions for software model checking. In *Proceedings of the Workshop on Software Model Checking*, 2003.

[14] C. Flanagan and S. Qadeer. A type and effect system for atomicity. In *Proceedings of the ACM Conference*

on Programming Language Design and Implementation, pages 338–349, 2003.

[15] M. Flatt, S. Krishnamurthi, and M. Felleisen. Classes and mixins. In *Proceedings of the ACM Symposium on the Principles of Programming Languages*, pages 171–183, 1998.

[16] P. Godefroid. *Partial-Order Methods for the Verification of Concurrent Systems: An Approach to the State-Explosion Problem*. Lecture Notes in Computer Science 1032. Springer-Verlag, 1996.

[17] T. Harris, S. Marlow, S. Peyton-Jones, and M. Herlihy. Composable memory transactions. In *Proceedings of the ACM Symposium on Principles and Practice of Parallel Programming*, pages 48–60, 2005.

[18] T. L. Harris and K. Fraser. Language support for lightweight transactions. In *Proceedings of the ACM Conference on Object-Oriented Programming, Systems, Languages and Applications*, pages 388–402, 2003.

[19] J. Hatcliff, Robby, and M. B. Dwyer. Verifying atomicity specifications for concurrent object-oriented software using model-checking. In *Proceedings of the International Conference on Verification, Model Checking and Abstract Interpretation*, pages 175–190, 2004.

[20] R. J. Lipton. Reduction: A method of proving properties of parallel programs. *Communications of the ACM*, 18(12):717–721, 1975.

[21] M. F. Ringenburg and D. Grossman. AtomCaml: First-class atomicity via rollback. In *ACM International Conference on Functional Programming*, 2005.

[22] A. Sasturkar, R. Agarwal, L. Wang, and S. D. Stoller. Automated type-based analysis of data races and atomicity. In *Proceedings of the ACM Symposium on Principles and Practice of Parallel Programming*, 2005.

[23] S. Savage, M. Burrows, G. Nelson, P. Sobalvarro, and T. E. Anderson. Eraser: A dynamic data race detector for multi-threaded programs. *ACM Transactions on Computer Systems*, 15(4):391–411, 1997.

[24] N. Sterling. WARLOCK — a static data race analysis tool. In *USENIX Technical Conference Proceedings*, pages 97–106, Winter 1993.

[25] S. D. Stoller. Model-checking multi-threaded distributed Java programs. In *Workshop on Model Checking and Software Verification*, pages 224–244, 2000.

[26] J.-P. Talpin and P. Jouvelot. Polymorphic type, region and effect inference. *Journal of Functional Programming*, 2(3):245–271, 1992.

[27] M. Tofte and J.-P. Talpin. Implementation of the typed call-by-value lambda-calculus using a stack of regions. In *Proceedings of the ACM Symposium on the Principles of Programming Languages*, pages 188–201, 1994.

[28] C. von Praun and T. Gross. Object race detection. In *Proceedings of the ACM Conference on Object-Oriented Programming, Systems, Languages and Applications*, pages 70–82, 2001.

[29] C. von Praun and T. Gross. Static conflict analysis for multi-threaded object-oriented programs. In *Proceedings of the ACM Conference on Programming Language Design and Implementation*, pages 115–128, 2003.

[30] C. von Praun and T. Gross. Static detection of atomicity violations in object-oriented programs. *Journal of Object Technology*, 3(6):103–122, 2004.

[31] L. Wang and S. D. Stoller. Runtime analysis of atomicity for multi-threaded programs. Technical Report DAR 04-14, Computer Science Department, SUNY Stony Brook, July 2004. A preliminary version appeared in *Proceedings of the Workshop on Runtime Verification*, 2003.

[32] A. Welc, S. Jagannathan, and A. L. Hosking. Transactional monitors for concurrent objects. In *Proceedings of the European Conference on Object-Oriented Programming*, pages 519–542, 2004.