Commutativity Race Detection

Dimitar Dimitrov
ETH Zürich
dimitar.dimitrov@inf.ethz.ch

Veselin Raychev
ETH Zürich
veselin.raychev@inf.ethz.ch

Martin Vechev
ETH Zürich
martin.vechev@inf.ethz.ch

Eric Koskinen
New York University
ejk@cims.nyu.edu

Abstract

This paper introduces the concept of a commutativity race. A commutativity race occurs in a given execution when two library method invocations can happen concurrently yet they do not commute. Commutativity races are an elegant concept enabling reasoning about concurrent interaction at the library interface.

We present a dynamic commutativity race detector. Our technique is based on a novel combination of vector clocks and a structural representation automatically obtained from a commutativity specification. Conceptually, our work can be seen as generalizing classical read-write race detection.

We also present a new logical fragment for specifying commutativity conditions. This fragment is expressive, yet guarantees a constant number of comparisons per method invocation rather than linear with unrestricted specifications.

We implemented our analyzer and evaluated it on real-world applications. Experimental results indicate that our analysis is practical: it discovered harmful commutativity races with overhead comparable to state-of-the-art, low-level race detectors.

1. Introduction

In response to the growing complexity of software, common patterns are increasingly being encapsulated into thread-safe libraries (*e.g.* Java collections). At the same time, virtually all modern applications use some form of concurrency and heavily rely on various libraries to accomplish their objective. This raises a key challenge: even though a library is implemented correctly, concurrent threads can invoke its operations in a way which causes undesirable interference at the library interface level, leading to incorrect program behaviors. We observe that such errors are fundamentally caused when concurrent threads perform non-commutative library operations. We refer to this phenomenon as a *commutativity race*. To illustrate the concept, consider the following simple program:

```
        T1:                          T2:

1:  fork T2;              3: int v=m.get(5);
2:  m.put(5,7);
```

Initially, the concurrent hashmap `m` has all keys initialized to the value `1`. Here, we have a commutativity race between `m.put(5,7)` and `m.get(5)`: the two operations do not commute *and* they can happen in any order. As a result of this commutativity race, the values returned by `get` differ in each of the two possible executions, returning `7` if `m.put(5,7)` occurred before `m.get(5)` in the execution or returning `1` otherwise. Such non-determinism can sometimes lead to undesirable behaviors further in the execution. In fact, in Java, if `m` was initially empty, and `get(5)` executes before `put(5,7)`, the program will throw a `NullPointerException` when unboxing the result from `get(5)`. Of course, not all commutativity races are harmful, however the presence of a commutativity race may indicate undesirable interference.

Commutativity race detection. In this paper, we introduce the concept of a *commutativity race* and present a dynamic commutativity race detector with the appropriate formal guarantees. Our approach is *parametric* on a declarative commutativity specification and can be used to analyze any library equipped with such a specification. The analysis is based on a novel combination of commutativity information with classic vector clocks [12, 14]. Conceptually, our work can be seen as a *generalization* of traditional data race detection (*e.g.* [3]) to deal with much richer and abstract notions of conflict, beyond the level of basic reads and writes.

We introduce an expressive logical fragment (ECL) for capturing commutativity conditions as well as an automatic translation procedure from ECL to a structural representation used by the commutativity race detector. ECL admits practical specifications (*e.g.* maps, sets) yet is amenable to efficient analysis: any commutativity condition expressed in ECL can be checked by the analysis via a *constant* number of comparisons per method invocation as opposed to linear with unrestricted specifications. We believe the results in this paper are of interest beyond race detection: the structural representation, ECL and the translation procedure between them can serve as a basis for other concurrency analyzers as well as for optimistic concurrency schemes (*e.g.* transactional memory).

Contributions. The main contributions of this paper are:

- A dynamic commutativity race detector based on a novel combination of classic vector clocks for tracking the happens-before relation with a structural representation of a commutativity specification (Section 5). Our approach generalizes traditional race detection to richer, more abstract notions of conflict.
- A logical fragment, called ECL, which captures useful commutativity conditions (*e.g.* maps), yet allows the analysis to perform a *constant* number of operations per method invocation, as opposed to linear with arbitrary specifications (Section 6.1).
- A translation procedure from ECL to the structural representation used by the analysis (Section 6.2).
- An implementation and an evaluation of our approach on two industrial Java applications (Section 7). Our results indicate that the performance overhead is acceptable and further, our tool discovered harmful commutativity races in these applications when using `ConcurrentHashMap`.

PLDI '14, June 09–11, 2014, Edinburgh, United Kingdom.

http://dx.doi.org/10.1145/2594291.2594322

```
1: var o = dictionary();    // Empty dictionary.
2: var hosts = ReadListOfHosts();
3: for host in hosts {
4:   fork {
5:     o.put(host, createConnection(host));
6:   }
7: }
8: joinall;
9: print(o.size() + " connections established");
```

Figure 1. An example of concurrently establishing connections to a list of hosts and storing them in a shared dictionary `o`.

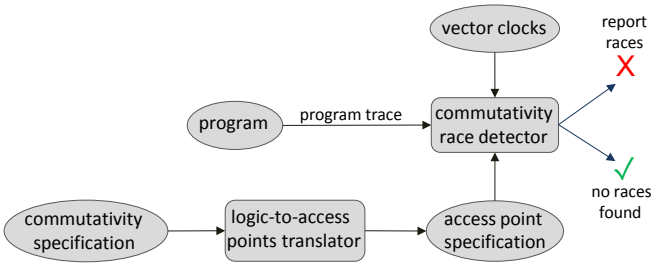

Figure 2. Our approach to commutativity race detection.

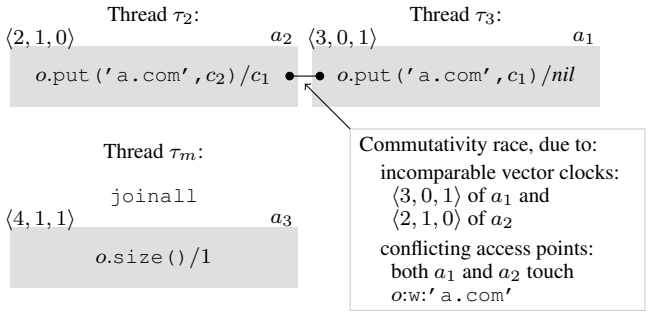

Figure 3. Algorithm operation on an example execution.

2. Overview

Next, we informally introduce and motivate the concepts underlying our approach. Full formal details are provided in later sections.

Consider the example program shown in Fig. 1. The program obtains a list of hostnames, creates a connection to each host and stores these connections in an initially empty associative map (a dictionary for which every key is initially associated with *nil*) such that every hostname is associated with a connection. In Java, the dictionary can be represented by `ConcurrentHashMap`. Connections are established in parallel as the program forks a thread for each connection. The command `joinall` waits for all previously created threads to finish. Once all threads have completed, the program prints the total number of established connections.

Example of a commutativity race. If `hosts` contains duplicates (*e.g.* `hosts = ['a.com', 'a.com']`), more than one thread will attempt to connect to the same host, leading to a *commutativity race* between a successful `o.put` invocation in one thread and an unsuccessful `o.put` operation in another. Such a commutativity race can indicate a potential application error: the developer did not think of the fact that the input list may contain duplicate hosts. Further, with every attempt to create a duplicate connection, there will be an additional short-lived connection object that is created and never used later, overburdening the underlying memory management system.

We next discuss our approach to detecting commutativity races. The approach consists of several ingredients. These ingredients as well as the flow between them are illustrated in Fig. 2. We discuss these ingredients in the context of our example. Later sections provide the appropriate formal treatment.

Step 1: obtain a commutativity specification. We capture commutativity between method invocations by using logical formulas: a natural, declarative way to capture commutativity [8]. That is, for every pair of methods of a given library, we provide a formula which describes when the two methods commute. For instance, the following formula describes when two `put` operations, `put`$(k_1, v_1)/p_1$ and `put`$(k_2, v_2)/p_2$, commute:

$$k_1 \neq k_2 \vee (v_1 = p_1 \wedge v_2 = p_2)$$

Step 2: obtain an access point representation. Next, our translator (discussed in Section 6.2) automatically converts a commutativity specification into what we call an *access point representation*, a form of an intermediate executable representation used by the analysis. Its main purpose is to reduce the number of checks that the analysis needs to perform. An access point representation captures the "micro actions" (called access points) that are relevant to commutativity checking. For instance, a successful o.`put`$(k, v)/nil$ is translated into two access points o:`w`:k and o:`resize` denoting that: i) the value associated with the particular key has changed, and ii) the size of the dictionary o has changed. Conflict checking can then be performed on individual access points as opposed to on entire invocations (if the analysis had worked using the commutativity specification directly). More generally, working with the access point representation of certain commutativity specifications (which we discuss in Section 6) leads to better *asymptotic complexity* of the analysis algorithm (*i.e.* constant over linear).

Step 3: commutativity race detection. Finally, our online dynamic analyzer consumes a program trace and checks the trace for commutativity races. Consider an execution of our running example shown in Fig. 3. First, thread τ_3 associates a connection c_1 with `'a.com'` in the dictionary. As the dictionary is initially empty, `put` returns *nil*. Next, thread τ_2 associates a new connection object c_2 with `'a.com'`, overwriting and returning the previous connection c_1. Finally, the main thread waits for both threads to complete and obtains the size of the dictionary, which is 1.

In the figure, we show the information computed by our algorithm when analyzing the trace, namely the vector clocks which capture the temporal relationship between invocations (also called actions) as well as the access points which capture the commutativity relationship between invocations. Here, each invocation is inside a shaded area (top right names the action) annotated with a vector clock in the top-left corner. A commutativity race occurs when there are two unordered actions (*i.e.* their vector clocks are non-comparable), yet their access points conflict.

In our example, actions a_1 and a_2 participate in a commutativity race. The reason is that: i) both of these actions are unordered as their vector clocks are non-comparable (we compare vector clocks by comparing their respective entries), and ii) when translated to access points, both actions share the same access point o:`w`:`'a.com'` which conflicts with itself.

Actions a_3 and a_1 do not participate in a commutativity race because they are ordered: the vector clock $\langle 3,0,1\rangle$ of action a_1 is less than the vector clock $\langle 4,1,1\rangle$ of action a_3. Note that had there not been a `joinall` statement in the code, then the two actions would conflict because action a_1 resizes the dictionary and touches access point o:`resize` which conflicts with the access point o:`size` touched by the action a_3.

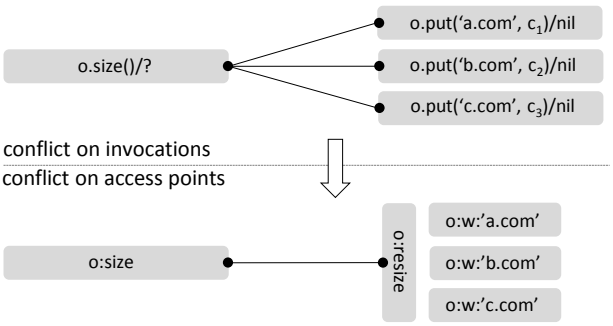

Figure 4. Conflict on invocations vs. conflict on access points

Finally, actions a_3 and a_2 do not participate in a commutativity race because they are ordered according to their vector clocks. Interestingly, had there not been a `joinall` statement in the code then these two actions would still *not* participate in a commutativity race because they touch access points which do not conflict: intuitively, action a_2 is not resizing the dictionary and hence cannot affect the result of a_3.

Motivation for access points. To give an intuition why we introduce access points, consider the example in Fig. 4. Here, in the top part, we have the main thread invoking `size` which conflicts with all three successful `put` invocations of other threads. An analysis which directly uses the logical specification would need to perform three checks for `size`: one with each `put` operation. However, for checking whether `size` and `put` commute, the only relevant information is whether a `put` operation is resizing the map. The bottom part of the figure shows the access points that our translation produces for each invocation. Here, each o.`put(`k`,`$v`)`$/nil$ invocation is translated into two access points: o:`w`:k and o:`resize`.

Interestingly, we see that each of the three `put` operations *share* the access point o:`resize` capturing the fact that the size of the dictionary changed. On the other hand, the `size()` operation generates a single access point `size`. With access points, a commutativity analysis now only needs to check whether the `size` access point conflicts with the resize access point o:`resize`. Importantly, this is a *single* conflict check and not three conflict checks as would be necessary without using access points.

Discussion. Before we proceed further, we discuss three key points. First, the commutativity race detector is orthogonal to how the access point representation is obtained. This representation can be obtained manually or automatically by translating from a commutativity specification. Second, the access point representation can be used to generalize other concurrency analyzers to high-level commutativity notions of conflict (*e.g.* atomicity [5]), or to enable more general optimistic concurrency control schemes (*e.g.* [10]). Similarly, the translator from commutativity specifications to access point representations is useful in those (and potentially other) settings as well. Third, as we will see later, the asymptotic time complexity of the analysis presented in this work differs depending on the different access point representations. To cleanly capture this fact, in Section 6 we present a commutativity logical fragment and a translation of any formula in that fragment to an access point representation which enables the analysis to always lookup a *constant* number of access points instead of linear as would be needed with arbitrary access point representations.

Method	Effect on the dictionary d
`put(`k`,`$v`)`$/p$	$d \longrightarrow d'$ iff $d' = d[k \mapsto v]$ and $p = d(k)$
`get(`k`)`$/v$	$d \longrightarrow d'$ iff $d' = d$ and $v = d(k)$
`size()`$/r$	$d \longrightarrow d'$ iff $d' = d$ and $r = \lvert\{k \mid d(k) \neq nil\}\rvert$

Figure 5. Dictionary methods and their effects

3. Preliminaries

In this section we provide necessary definitions of several terms used later in the paper.

3.1 Execution model

Actions. We consider concurrent programs consisting of threads that communicate via shared *objects*. Each object $o \in Obj$ can be in a set of possible *abstract* states. That is, we are interested in the abstract states of the object as described by its specification and not in the actual implementation details of the object. For example, the abstract state of a dictionary object is a key-value mapping of type $K \to (V \cup \{nil\})$ where K is a set of possible keys, V is the set of possible values and *nil* is a special no-value.

The shared state of a program consists of the abstract states of all shared objects. The space of possible shared states is denoted by H. For example, if a program references only one shared object, a dictionary, then $H = K \to (V \cup \{nil\})$. We assume that object methods are given by specifying their effects on the shared state, *e.g.*, Fig. 5 describes the method effects of the dictionary object. We refer to method invocations as *actions*.

An action $a \in Act$ is denoted by an expression of the form $o.m(\vec{u})/\vec{v}$ where $o \in Obj$ is an object, m is a method of o and $\vec{u}$ and $\vec{v}$ are tuples of concrete arguments and return values that match the signature of m. The effect on the shared state of every $a \in Act$ is given by a partial map $(\!|a|\!) \in H \rightharpoonup H$. For example, for a dictionary object o, the map $(\!|o.\texttt{size}()/n|\!)$ should be the identity on all states in which the dictionary o has size n and undefined otherwise.

We say that two actions commute when independent of application order, their composed effects are the same:

Definition 3.1. *Two actions $a, b \in Act$ commute, denoted by $a \bowtie b$, iff $(\!|a|\!) \circ (\!|b|\!) = (\!|b|\!) \circ (\!|a|\!)$.*

For example, following Fig. 5, two `put` actions commute when they operate on different keys as the two actions modify disjoint parts of the object state. We assume that actions of different objects always commute, that is method invocations of one object do not interfere with the state of another object.

Executions and Events. We model programs as labeled transition systems and their executions as traces in the associated system.

In this work we focus on whether an object is used correctly and assume that it is implemented correctly (*i.e.* the object is linearizable). This allows us to treat method invocations as atomic transitions. We assume that every state transition performs a single action on a shared object with possible effects on the thread-local state. We write $s \xrightarrow{\tau:a} s'$ for a transition from program state s to program state s' accompanied by the execution of an action $a \in Act$ by thread $\tau \in Tid$, where *Tid* denotes the set of thread identifiers.

A program execution is modeled by a sequence of state transitions $\pi = s_0 \xrightarrow{\tau_1:a_1} s_1 \xrightarrow{\tau_2:a_2} \cdots \xrightarrow{\tau_n:a_n} s_n$ that starts in an initial state s_0. Such sequences are called *traces*. An occurrence e_i of a transition label $\tau_i : a_i$ in the trace is called an *event* and thus

the trace π has a multiset[1] of events $\{e_i = \tau_i : a_i\}_{i=1}^n$. Such a multiset is ordered by event position, that is $e_i \leq_\pi e_j$ iff $i \leq j$. The fact that e is an event will be denoted by $e \in \mathit{Evt}$. If e occurs in a trace π we will write $e \in \pi$.

3.2 Happens-before

Happens-before relations [12] are a standard way to encode causality between events. A *happens-before* relation for a trace π is a certain partial ordering $\preceq$ on π's multiset of events. This order records causal relationships between the events. When $e_i \preceq e_j$ then there must be some cause that forces e_i to appear before e_j. If neither $e_i \preceq e_j$ nor $e_j \preceq e_i$ we say that e_i and e_j may happen in parallel and denote it by $e_i \parallel e_j$. The relation $\preceq$ satisfies:

- if $e_i \preceq e_j$ then $e_i \leq_\pi e_j$
- if the program forces e_i to occur before e_j then $e_i \preceq e_j$

For example, one thread awaiting for the termination of another forces subsequent events from the waiting thread to occur after the events from the other thread. Another example is the sequential execution of actions by a single thread. In particular, if $e_i \leq_\pi e_j$ have occurred in the same thread then $e_i \preceq e_j$. In other words, if $e_i \parallel e_j$ then e_i and e_j must have occurred in different threads.

Vector clocks. Vector clocks [14] are a well-known technique for computing the happens-before relation. A *vector clock* is a map $c \in \mathit{Tid} \to \mathbb{N}$ that records a timestamp $c(\tau)$ from the local clock of every thread τ. The set of vector clocks $\mathit{Tid} \to \mathbb{N}$ is denoted by VC. The set of vector clocks VC is ordered pointwise becoming a lattice with a smallest element. The additional operation inc_υ performs one timestep increment of the υ component:

$$\begin{aligned} c_1 \sqsubseteq c_2 &\text{ iff } c_1(\tau) \leq c_2(\tau) \text{ for all } \tau \in \mathit{Tid} \\ c_1 \sqcup c_2 &= \tau \mapsto \max(c_1(\tau), c_2(\tau)) \\ \bot_V &= \tau \mapsto 0 \\ inc_\upsilon(c) &= \tau \mapsto c(\tau) + 1 \text{ if } \tau = \upsilon \text{ else } c(\tau) \end{aligned}$$

A happens-before relation $\preceq$ is represented by associating a vector $vc(e)$ clock with every event e

$$vc : \mathit{Evt} \to VC$$

in a way such that $e_i \preceq e_j$ iff $vc(e_i) \sqsubseteq vc(e_i)$. Informally, the vector clock $vc(e)$ reflects the dependency of e on the progress of every thread.

4. Commutativity Specifications and Their Representation

We next discuss how we capture commutativity specifications as well as how these commutativity specifications can be represented in a form that is usable by a dynamic analysis system. This representation is used later by our commutativity race detector in Section 5 and can potentially be applied to generalize other concurrency analyzers. Hence, the discussion in this section is also of interest beyond race detection.

4.1 Logical Specifications

The conditions under which two actions commute are often conveniently specified in the form of logical formulas. This allows formal treatment of various types of conditions. Briefly, a (sound) logical commutativity specification is a predicate φ over pairs of actions $a, b \in \mathit{Act}$ such that $\varphi(a, b)$ implies that a and b commute. Commutativity specification of a dictionary object is given on Fig. 6.

[1] By a multiset we mean a set of labeled points rather than a set with element multiplicities.

$$\begin{aligned} \varphi^{\texttt{put}(k_1,v_1)/p_1}_{\texttt{put}(k_2,v_2)/p_2} &:= k_1 \neq k_2 \vee (v_1 = p_1 \wedge v_2 = p_2) \\ \varphi^{\texttt{put}(k_1,v_1)/p_1}_{\texttt{get}(k_2)/v_2} &:= k_1 \neq k_2 \vee (v_1 = p_1) \\ \varphi^{\texttt{put}(k_1,v_1)/p_1}_{\texttt{size}()/r} &:= (v_1 = \mathit{nil} \wedge p_1 = \mathit{nil}) \vee (v_1 \neq \mathit{nil} \wedge p_1 \neq \mathit{nil}) \end{aligned}$$

$$\varphi^{\texttt{get}(k_1)/v_1}_{\texttt{get}(k_2)/v_2}, \varphi^{\texttt{get}(k_1)/v_1}_{\texttt{size}()/r}, \varphi^{\texttt{size}()/r_1}_{\texttt{size}()/r_2} := \mathit{true}$$

Figure 6. Commutativity specification of a dictionary object

The kind of commutativity specifications we discuss in this work are those that can be expressed without mentioning the object state. A more relaxed notion of commutativity that can depend on object state may be useful, however this may have significant implications on its mechanized checking and we leave those as future work. It is worth mentioning that in certain cases exposing hidden state as shadow return values may allow obtaining more precise specification. However, this cannot capture the general notion of "two actions commute at a specific state".

Next, we proceed to define logical commutativity specifications, soundness of such specifications and commutativity races.

Definition 4.1. *Given a suitable logic, a logical commutativity specification for a pair of methods m_1, m_2 of the same object is given by a logical formula $\varphi^{m_1}_{m_2}$ with its free variables collected into the list $(\vec{x}_1; \vec{x}_2)$ so the variables $\vec{x}_i$ match the arguments and returns of m_i. A logical commutativity specification Φ for an object with methods M is a set of method specifications $\varphi^{m_1}_{m_2}(\vec{x}_1; \vec{x}_2)$ [2] for each $\{m_1, m_2\} \subseteq M$.*

For a method specification $\varphi^{m_1}_{m_2}(\vec{x}_1; \vec{x}_2)$ and suitable terms $\vec{t}_1, \vec{t}_2$ we will write $\varphi^{m_1}_{m_2}(\vec{t}_1; \vec{t}_2)$ for the substitution of $\vec{t}_i$ for $\vec{x}_i$ into the formula $\varphi^{m_1}_{m_2}$. Similarly, we can evaluate the formula for concrete values $\vec{u}_1, \vec{u}_2$. We require specifications to represent symmetric predicates on actions, that is we require that $\varphi^m_m(\vec{x}_1; \vec{x}_2)$ is logically equivalent to $\varphi^m_m(\vec{x}_2; \vec{x}_1)$.

Example. An example of a commutativity specification for our dictionary object o for the method pair $\texttt{put}(k_1, v_1)/p_1$ and $\texttt{put}(k_2, v_2)/p_2$ would be:

$$\varphi^{\texttt{put}}_{\texttt{put}}(k_1, v_1, r_1; k_2, v_2, r_2) := k_1 \neq k_2 \vee (v_1 = p_1 \wedge v_2 = p_2)$$

Note that when we swap parameters we get the equivalent formula:

$$\varphi^{\texttt{put}}_{\texttt{put}}(k_2, v_2, r_2; k_1, v_1, r_1) := k_2 \neq k_1 \vee (v_2 = p_2 \wedge v_1 = p_1)$$

For a specification $\varphi^{m_1}_{m_2}(\vec{x}_1; \vec{x}_2)$ and two actions $a = o.m_1(\vec{u}_1)/\vec{v}_1$ and $b = o.m_2(\vec{u}_2)/\vec{v}_2$, we write $\varphi^{m_1}_{m_2}(a, b)$ for the evaluation $\varphi^{m_1}_{m_2}(\vec{u}_1\vec{v}_1; \vec{u}_2\vec{v}_2)$.

We would expect commutativity specifications to actually say something about the commutativity properties of the object in question. Sound specifications imply commutativity:

Definition 4.2. *A logical commutativity specification Φ for some object is* sound *iff for every method specification $\varphi^{m_1}_{m_2}(\vec{x}_1; \vec{x}_2) \in \Phi$ and two actions a, b of m_1, m_2 respectively, we have that $\varphi^{m_1}_{m_2}(a, b)$ implies $a \bowtie b$.*

Note that the above definition allows for imprecise commutativity specifications. Even though actions commute, the specification is allowed to say that they do not. One could replace the implication with an if-and-only-if condition but this is unnecessary for our algorithm. We are now ready to define the concept of a *commutativity race*, a central concept of our paper.

[2] The notation $\varphi^{m_1}_{m_2}(\vec{y}_1\vec{r}_1; \vec{y}_2\vec{r}_2)$ is an alternative to $\varphi^{m_1(\vec{y}_1)/\vec{r}_1}_{m_2(\vec{y}_2)/\vec{r}_2}$

$$\mathcal{X}_o = \{o{:}\mathtt{r}{:}k\}_{k\in K} \cup \{o{:}\mathtt{w}{:}k\}_{k\in K} \cup \{o{:}\mathtt{size}, o{:}\mathtt{resize}\}$$

(a) Access points

Action Type	Access Points (η_o)	Condition
$o.\mathtt{put}(k, v)/p$	$o{:}\mathtt{w}{:}k, o{:}\mathtt{resize}$	$v \neq p$ and size changed
	$o{:}\mathtt{w}{:}k$	$v \neq p$ and size unchanged
	$o{:}\mathtt{r}{:}k$	$v = p$
$o.\mathtt{get}(k)/v$	$o{:}\mathtt{r}{:}k$	
$o.\mathtt{size}()$	$o{:}\mathtt{size}$	

(b) Action access points

$\mathcal{C}_o$	$o{:}\mathtt{r}{:}l$	$o{:}\mathtt{w}{:}l$
$o{:}\mathtt{w}{:}k$	$k = l$	$k = l$
$o{:}\mathtt{r}{:}k$	no	$k = l$

$\mathcal{C}_o$	$o{:}\mathtt{size}$	$o{:}\mathtt{resize}$
$o{:}\mathtt{resize}$	yes	no
$o{:}\mathtt{size}$	no	yes

(c) Conflict relation $\mathcal{C}_o$

Figure 7. Access point representation for a dictionary.

Definition 4.3. *For a trace π, a pair of events $e_i, e_j \in \pi$ participate in a* commutativity race *with respect to a partial order $\preceq$ and a commutativity specification of their corresponding methods $\varphi_{m_2}^{m_1}(\vec{x}_1; \vec{x}_2)$ iff $e_i \parallel e_j$ (the events may happen in parallel) and their corresponding actions a, b are not specified to commute, that is, $\varphi_{m_2}^{m_1}(a, b)$ does not hold.*

Note that if the commutativity specification is imprecise, it is possible to report a commutativity race even if actions do commute. To avoid clutter, we use the term commutativity race regardless of whether the specification is precise or not.

4.2 Access point representation

We next introduce a structure which is useful for capturing commutativity specifications in a way that can be used by a dynamic program analyzer. We refer to these structures as *access point representations* (of the object's commutativity properties).

Definition 4.4. *An access point representation for an object $o \in Obj$ is a tuple $\langle \mathcal{X}_o, \eta_o, \mathcal{C}_o \rangle$, where:*

1. *$\mathcal{X}_o$ is a set of access points.*
2. *$\eta_o \in Act_o \to \mathcal{P}(\mathcal{X}_o)$ indicates the finite set of access points touched by each action of the object (Act_o stands for the set of all actions of object o).*
3. *$\mathcal{C}_o \subseteq \mathcal{X}_o \times \mathcal{X}_o$ is a symmetric binary relation describing which access points conflict.*

Example: Dictionary Fig. 7 presents an access point representation for a dictionary object o. Fig. 7(a) shows two groups of access points. The first group contains two access points for every possible key: $o{:}\mathtt{r}{:}k$ indicates that the value for key k has been read; $o{:}\mathtt{w}{:}k$ indicates that the value for key k has changed. The second group has only two access points: $o{:}\mathtt{size}$ indicates that the dictionary size has been observed; $o{:}\mathtt{resize}$ indicates that the dictionary has been resized.

Fig. 7(b) shows the possible types of actions and the access points which are touched upon the execution of that action. For example, the action $o.\mathtt{put}(5, 1)/nil$ changes the value for key 5 and also changes the size of the dictionary. Therefore, $\eta_o(o.\mathtt{put}(5, 1)/nil) = \{o{:}\mathtt{w}{:}5, o{:}\mathtt{resize}\}$. Whether the action $o.\mathtt{put}(k, v)/p$ changes the size is determined by the parameters alone: either $v \neq nil$ and $p = nil$ or $v = nil$ and $p \neq nil$.

Fig. 7(c) shows the conflict relation for each of the two groups of access points (for convenience split in two matrices). The entry in the matrix denotes when the two access points conflict. For example, $o{:}\mathtt{r}{:}5$ and $o{:}\mathtt{w}{:}5$ conflict (as $k = l$) while $o{:}\mathtt{r}{:}5$ and $o{:}\mathtt{w}{:}6$ do not conflict (as $k \neq l$). There are no conflicts between actions from different groups.

We now define what it means for an access point representation to precisely match a logical specification. In Section 6, we show a translation from a logical formula to an equivalent access point representation.

Definition 4.5. *We say that $\langle \mathcal{X}_o, \eta_o, \mathcal{C}_o \rangle$* represents *a logical commutativity specification Φ of o iff for all $\varphi_{m_2}^{m_1} \in \Phi$ and all actions a of m_1 and b of m_2 we have:*

$$(\eta_o(a) \times \eta_o(b)) \cap \mathcal{C}_o = \emptyset \quad \textit{iff} \quad \varphi_{m_2}^{m_1}(a, b)$$

Therefore, instead of a given logical specification Φ, we can now work with an access point representation $\langle \mathcal{X}_o, \eta_o, \mathcal{C}_o \rangle$ equivalent to Φ. We utilize access point representations when detecting commutativity races.

5. Commutativity Race Detection

We next present our commutativity race detector, which is parameterized over an access point representation. We first motivate our approach, then we present our algorithm and discuss how its asymptotic complexity can be affected by choosing different commutativity specifications, and finally provide the formal guarantees.

5.1 Challenges

Recall that in Section 2, we discussed the benefits of working with an access point representation over working directly with a commutativity specification. Let us again consider the direct approach where we work with the commutativity specification directly. This approach works as follows. It records every action occurring in an execution and every time it encounters a new action, it checks whether there is a previously observed action on the same object that can happen concurrently with the current one and which does not commute with it.

Checking whether two actions a and b commute is done by simply checking whether the pair of actions provide a satisfying assignment to the corresponding commutativity formula. That is, whether $\varphi_{m_2}^{m_1}(a, b)$ evaluates to $true$ where m_1 is the method of action a and m_2 is the method of action b.

A key challenge with the direct approach is that it is oblivious to the type of commutativity specification being checked and ignores the fact that many actions share commonalities. For instance, as illustrated earlier in Fig. 4, several $\mathtt{put}$ actions share the common access point $o{:}\mathtt{resize}$. As a result, the algorithm records each action independently and requires $\Theta(|A|)$ commutativity checks for each encountered action, where A is the set of all actions occurring in the execution.

Our algorithm aims to remedy these deficiencies by exploiting the structure present in commutativity specifications. Instead of working directly with logical formulas, the algorithm leverages access point representations (described in Section 4.2). Access point representations allow us to expose some of the common structure present in commutativity predicates (we already illustrated this earlier in Fig. 4).

Further, in Section 6, we will introduce an expressive logical fragment (called ECL) where formulas in that fragment are translated to access point representations that require only $\Theta(1)$ checks for each encountered action as opposed to $\Theta(|A|)$.

	Event	Modifications of auxiliary state
Previous work		
	$e = \tau : fork(u)$	$T(u) \leftarrow inc_u(T(\tau))$
		$T(\tau) \leftarrow inc_\tau(T(\tau))$
	$e = \tau : join(u)$	$T(\tau) \leftarrow T(\tau) \sqcup T(u)$
	$e = \tau : acq(l)$	$T(\tau) \leftarrow T(\tau) \sqcup L(l)$
	$e = \tau : rel(l)$	$L(l) \leftarrow T(\tau)$
		$T(\tau) \leftarrow inc_\tau(T(\tau))$
Our work		
	$e = \tau : o.m(\vec{x})/\vec{y}$	$vc(e) \leftarrow T(\tau)$
		Execute Algorithm 1

Table 1. Handling synchronization and action events.

5.2 Common Setting

Our approach is applicable to any parallel language. However, to convey an end-to-end picture we describe how our algorithm fits with the fork-join constructs. The first four rows in Table 1 describe the standard way of updating vector clocks for the statements:

- $\tau : fork(u)$ – thread τ creates thread $u \in \mathit{Tid}$.
- $\tau : join(u)$ – thread τ waits until thread u terminates.
- $\tau : acq(l)$ – thread τ acquires a lock $l \in Lock$.
- $\tau : rel(l)$ – thread τ releases a lock $l \in Lock$.

To capture the happens-before relation, we maintain two auxiliary mappings $T : \mathit{Tid} \to VC$ and $L : Lock \to VC$. All vector clocks are initialized with $\bot_V$ and the auxiliary state is updated at every synchronization event in the trace according to Table 1. Such handling of vector clocks is standard (*e.g.* see [3]). The novel part of our approach, shown in the last row, is in handling of actions, specifically Algorithm 1 which we discuss next.

5.3 Our Approach

Next, we present our algorithm for detecting commutativity races given an access point representation. The algorithm works on-the-fly and reports races as the trace is being explored. The novel part in our dynamic analysis is handling of action events $e = \tau : o.m(\vec{x})/\vec{y}$ and is described in Algorithm 1 (in the algorithm we describe the handling of a single action event). We assume an access point representation $\langle \mathcal{X}_o, \eta_o, \mathcal{C}_o \rangle$ for each object $o \in Obj$ used by the program. Moreover, we assume that the sets $\mathcal{X}_o$ for different objects are disjoint. Let $\mathcal{X}$ be their union. Also, let $\mathcal{C}_o(pt)$ denote all the access points that conflict with pt, that is, $\mathcal{C}_o(pt) = \{pt' \mid (pt, pt') \in \mathcal{C}_o\}$. The algorithm maintains the following additional auxiliary state:

- $ptvc\colon \mathcal{X} \to VC$. Here, we keep a vector clock for each access point. For an access point pt, we use $pt.vc$ as a shortcut for $ptvc(pt)$. The entries $pt.vc$ are initialized by the algorithm on demand.
- $active\colon Obj \to \mathcal{P}(\mathcal{X})$. Here, for each object, we keep the set of access points touched by all object actions performed so far, so $active(o)$ is a finite subset of $\mathcal{X}_o$. Initially, $active(o)$ is the empty set for every object o.

In the first phase (Line 2 to Line 7), the algorithm checks for a commutativity race. The algorithm iterates over each access point pt touched by the action of the event e and seeks all active points $pt' \in active(o)$ which conflict with pt. This is done by computing the intersection of $active(o)$ and $\mathcal{C}_o(pt)$. A commutativity race is detected if a conflicting access point pt' can be touched concurrently to the current event e as determined by the vector clocks $vc(e)$ and $pt'.vc$.

In the second phase (Line 8 to Line 16), the vector clocks of the access points associated with the action of the event e are updated. The vector clock of every such access point pt accumulates the vector clocks of all actions that have touched pt so far. If an access point pt touched by the action e was not active, it is made active.

Algorithm 1: Commutativity Race Detector

Input: event $e = \tau : o.m(\vec{x})/\vec{y}$, vector clock $vc(e)$,
access point representation $\langle \mathcal{X}_o, \eta_o, \mathcal{C}_o \rangle$
Output: report a commutativity race or update auxiliary state

```
1  begin
2      // phase 1: check for commutativity races
3      for pt ∈ η_o(o.m(x⃗)/y⃗) do
4          for pt' ∈ active(o) ∩ C_o(pt) do
5              if pt'.vc ⋢ vc(e) then
6                  report "commutativity race";
7
8      // phase 2: update auxiliary state
9      for pt ∈ η_o(o.m(x⃗)/y⃗) do
10         if pt ∈ active(o) then
11             pt.vc ← pt.vc ⊔ vc(e)
12         else
13             // initialize
14             pt.vc ← vc(e);
15             active(o) ← active(o) ∪ {pt};
16
```

In the algorithm, the set $active(o)$ grows continuously. To reduce its size, several optimizations are possible. For example, if an object o is reclaimed (*i.e.* collected, released, etc.), all of its access points and their corresponding vector clocks can be reclaimed too, as no new races can be reported on a dead object. In our tool, we have implemented this optimization by attaching the auxiliary state directly to the object to which it belongs and then removing that state whenever the object dies. Another conceptually interesting optimization that we leave for future work is to remove unnecessary active access points by exploiting properties of the access point representation $\langle \mathcal{X}_o, \eta_o, \mathcal{C}_o \rangle$.

Example. Consider again the example trace from Fig. 3 discussed in Section 2 (repeated here for convenience):

event action	identifier	vector clock
τ_3 : `o.put('a.com', c1)` /*nil*	a_1	$\langle 3, 0, 1 \rangle$
τ_2 : `o.put('a.com', c2)` /c_1	a_2	$\langle 2, 1, 0 \rangle$
τ_m : `join` $\{\tau_2, \tau_3\}$		
τ_m : `o.size()`/1	a_3	$\langle 4, 1, 1 \rangle$

Let us explain how Algorithm 1 detects the commutativity race from Fig. 3. On execution of action a_1, the algorithm checks that access point $pt = o$:w:`'a.com'` does not conflict with any previous access point. This is true, as pt is the first encountered access point. Then, at Line 14, the algorithm records that pt was touched with vector clock $\langle 3, 0, 1 \rangle$. Next, action a_2 executes and touches the same access point pt. At Line 4, the algorithm encounters pt again which according to $\mathcal{C}_o$ conflicts with itself (the conflict matrix $\mathcal{C}_o$ for dictionary is shown in Fig. 7(c)). Further, the vector clock of the previous access is $\langle 3, 0, 1 \rangle$ which is incomparable with $\langle 2, 1, 0 \rangle$ – the vector clock of the current event. Therefore, a

commutativity race is reported. The race reveals that action a_2 can execute concurrently with some previous action and both actions touch access points that conflict. Continuing, the algorithm updates the vector clock of pt to the join $\langle 3,1,1\rangle$. In the remainder of the execution, no more commutativity races will be reported, because the vector clock of a_3 is $\langle 4,1,1\rangle$, which is greater than or equal than all previously recorded vector clocks.

5.4 Algorithm Complexity

The complexity of the algorithm is dominated by the number of iterations at Line 4. In particular, for an unrestricted access point representation, the set $\mathcal{C}_o(pt)$ can be infinite. For example, a naive access point representation for a dictionary object can be obtained by associating an access point with each possible action. Then, the access point for the `size` action would conflict with an *infinite* number of `put` actions that change the dictionary size. In such cases, enumerating the intersection $active(o) \cap \mathcal{C}_o(pt)$ at Line 4 must be accomplished by first enumerating the finite set $active(o)$ and then checking if every element belongs to $\mathcal{C}_o(pt)$. That is, the time needed to process each action is at least linear in $|active(o)|$ and the set $active(o)$ grows as the program executes.

However, for certain specifications, we can obtain a $\mathcal{C}_o(pt)$ that is finite, *e.g.*, the dictionary specification in Fig. 6 has $\mathcal{C}_o(pt)$ of size at most 2 (see Fig. 7(c)). In this case, the enumeration at Line 4 may be done the other way around: for every element of $\mathcal{C}_o(pt)$, perform the constant time check[3] whether it belongs to the set $active(o)$. Effectively, in this case, the number of iterations at Line 4 is bounded by $|\mathcal{C}_o(pt)|$. Later, in Section 6 we present a class of specifications which also includes the dictionary specification in Fig. 6. For every specification in this class there is an upper bound on $|\mathcal{C}_o(pt)|$ for any $pt \in \mathcal{X}_o$, thus enabling better complexity of the commutativity race detection algorithm.

5.5 Guarantees

We next state two important guarantees which hold in our setting. Proof of the first theorem can be found in the appendix, while the second theorem is standard and we do not provide further discussion. The theorems are valid under the assumption that the only source of nondeterminism in the underlying system is due to nondeterministic scheduling.

Theorem 5.1. *Algorithm 1 reports a commutativity race if and only if the observed trace π contains a commutativity race.*

The above theorem states that our analysis is precise in the sense that some commutativity race is reported if and only if some commutativity race exists according to the commutativity specification. In general, the first reported race is guaranteed to be a true race. The next theorem is independent of the algorithm and extrapolates commutativity race freedom to a class of traces.

Theorem 5.2. *If a trace π with a happens-before relation $\preceq$ has no commutativity races with respect to $\preceq$ and a* sound *commutativity specification Φ, then all traces which admit $\preceq$ and start in the same state as π:*

1. *end in the same state as π, and*
2. *have no commutativity races with respect to $\preceq$ and Φ*

The theorem states that if an observed trace is free of races, then each trace which admits the same happens-before relation as the observed trace is free of races and computes the same end result (guarantees determinism for the initial state of π). This is practically useful as it means that there is no need to explicitly enumerate and check those traces with our algorithm.

[3] Assuming $\Theta(1)$ average hash-table lookup

6. Commutativity Logic and its Translation

In this section we introduce a logical fragment for describing commutativity specifications, called ECL, as well as a translation procedure from that fragment to access point representations. The key benefit of ECL over arbitrary commutativity specifications is that we can translate ECL formulas to access point representations in which an access point always conflicts with a *bounded number* of other access points. This has complexity implications on the analysis (see Section 5.4).

Why* ECL*? Our approach to defining ECL is inspired by the work of Kulkarni *et al.* [10] which characterized the SIMPLE class of formulas and proved that under certain restrictions only such formulas can be checked with a mechanism called abstract locks. Unfortunately, SIMPLE does not precisely capture many practically useful specifications such as sets or the dictionary example in Fig. 6. We observe that in the setting of dynamic race detection (and concurrency analysis in general), the restriction considered in Kulkarni *et al.*'s work is unnecessary. Therefore, our objective is to develop a logical fragment which improves the asymptotic complexity of commutativity race detection yet is expressive enough to capture practical commutativity specifications.

6.1 ECL: extended commutativity logic

We next describe our logical fragment. We need to distinguish between arguments coming from the two actions referred in a specification formula. For this reason let V_1 and V_2 be two disjoint supplies of variables intended for arguments to the first method and for arguments to the second method respectively. We assume that the variables range over some domain $\mathcal{U}$.

Because ECL extends SIMPLE, for convenience, we first repeat the definition of SIMPLE from Kulkarni *et al.*:

Definition 6.1. L_S

$$S ::= V_1 \neq V_2 \mid S \wedge S \mid \text{true} \mid \text{false}$$

As mentioned, L_S does not allow one to capture many practical specifications. Therefore, in order to allow more expressive commutativity specifications, we combine L_S with an additional fragment L_B of more complex structure. Mainly, L_B allows one to express constraints other than $x \neq y$.

Let P_V stand for a set of atomic formulas interpreted in $\mathcal{U}$ with their variables restricted to be in a set V. In addition to equality, $x = y$, the set P_V may also contain formulas such as $x < y$, $x \cdot y = z$, etc. The sets P_{V_1}, P_{V_2} are a parameter to the following:

Definition 6.2. L_B

$$B ::= P_{V_1} \mid P_{V_2} \mid \neg B \mid B \wedge B \mid B \vee B \mid \text{true} \mid \text{false}$$

The key restriction on *atomic* L_B formulas is that their arguments must come from only one of V_1 or V_2. Therefore, the main characteristic property of L_B is that the truth value of any atomic L_B subformula depends on the valuation of variables in either V_1 or in V_2, but *not* in both.

Example. Suppose that $V_1 = \{x, y\}$ and $V_2 = \{z\}$. Then, L_B allows the atomic formulas $x < y$ and $0 < z$, but not the atomic formula $x < z$, because this atomic formula contains variables from both V_1 *and* V_2. L_B also allows us to capture the formula $x < y \wedge 0 < z$. Here, the truth value of the atomic sub-formula $x < y$ depends only on the variables in V_1. Similarly, the truth value of the atomic sub-formula $0 < z$ depends only on the variables in V_2.

The extended commutativity logic (ECL) combines the two fragments L_S and L_B. ECL combines these by permitting conjunctions of ECL formulas and disjunctions of ECL and L_B formulas.

Definition 6.3. ECL

$$X ::= S \mid B \mid X \wedge X \mid X \vee B$$

For an ECL method specification $\varphi^{m_1}_{m_2}(\vec{x};\vec{y})$ the variables $\vec{x}$ must be drawn from the set V_1, while the variables $\vec{y}$ must be drawn from the set V_2. A key property of ECL, easily proved by structural induction, is the following:

Lemma 6.4. *For any* ECL *formula, if a truth assignment to all the L_B atomic subformulas is given, then the whole formula simplifies to a L_S formula.*

Example. Here is an example commutativity specification expressible in ECL, also shown earlier in Fig. 6:

$$\varphi^{\texttt{put}}_{\texttt{put}}(k_1, v_1, p_1; k_2, v_2, p_2) := k_1 \neq k_2 \vee v_1 = p_1 \wedge v_2 = p_2$$

Here $k_1, v_1, p_1 \in V_1$ and $k_2, v_2, p_2 \in V_2$. This formula is not expressible in SIMPLE for two reasons: first, the formula contains a disjunction and second, the formula compares for equality, *i.e.* $v_1 = p_1$.

6.2 Translation from ECL to an access point representation

Recall that a logical commutativity specification Φ for an object $o \in \mathit{Obj}$ consists of a commutativity specification $\varphi^{m_1}_{m_2}$ for each pair of methods m_1, m_2. Next, we describe how to obtain an access point representation for such a specification Φ in ECL. This conversion of the ECL fragment builds upon, formalizes and extends the conversion of SIMPLE in [10].

Define $\mathcal{X}_o$**.** Access points are touched by methods during invocation. Each access point witnesses specific information about the invocation. The β component (to be defined below) collects the truth value of certain predicates over the action arguments and the return values. The translation defines two types of access points:

- $o.m{:}\beta{:}ds$ simply witnesses that a method m of the object o has been invoked
- $o.m{:}\beta{:}i{:}w_i$ witnesses the i-th argument or return value w_i of an invocation of m

The ds symbol is just a tag for the first type of access points. Each β vector maps an atomic L_B subformula from Φ to a truth value. Let $B(\Phi)$ be the set of all atomic formulas from the L_B fragment that occur in the specification Φ. Further, let each formula in $B(\Phi)$ be normalized by dropping the distinction between the two types of variables V_1 and V_2.

Example. Consider the dictionary specification in Fig. 6. Its L_B atomic subformulas are $v_1 = p_1$, $v_2 = p_2$, $v_1 = \mathit{nil}$, $p_1 = \mathit{nil}$. To normalize, we just erase the indices obtaining that $B(\Phi)$ is the set $\{v = p, v = \mathit{nil}, p = \mathit{nil}\}$.

To define $\mathcal{X}_o$, let M be the set of methods of the object o and let $\mathcal{U}$ be the domain of possible arguments and return values for these methods. The set of access points is the product:

$$\mathcal{X}_o = \{o\} \times M \times (B(\Phi) \rightharpoonup \{\mathit{true}, \mathit{false}\}) \times (\{ds\} \cup \mathbb{N} \times \mathcal{U})$$

Define β**.** Next, we define the β component associated with each action $a = o.m(\vec{u})/\vec{v}$. Let $B(\Phi, m)$ be the subset of $B(\Phi)$ consisting of those atomic formulas that are relevant to the method m, that is, normalized atomic subformulas of some $\varphi^{m_1}_{m_2} \in \Phi$ with $m \in \{m_1, m_2\}$. The vector $\beta : B(\Phi, m) \to \{\mathit{true}, \mathit{false}\}$ collects the truth values of $B(\Phi, m)$ evaluated on $a = o.m(\vec{u})/\vec{v}$, that is, for $q \in B(\Phi, m)$:

$$\beta(q) = q(\vec{u}\vec{v})$$

Example. Consider the action $a = o.\texttt{put}\,(5, 6)\,/\mathit{nil}$. The relevant specifications to `put` are $\varphi^{\texttt{put}}_{\texttt{put}}$, $\varphi^{\texttt{put}}_{\texttt{get}}$ and $\varphi^{\texttt{put}}_{\texttt{size}}$ from Fig. 6. Thus, the relevant atomic formulas $B(\Phi, \texttt{put})$ are:

$$v = p, v = \mathit{nil}, p = \mathit{nil}$$

and substituting the action parameters $k = 5$, $v = 6$ and $p = \mathit{nil}$ we obtain the vector:

$$\beta = \{(v = p) \mapsto \mathit{false}, (v = \mathit{nil}) \mapsto \mathit{false}, (p = \mathit{nil}) \mapsto \mathit{true}\}$$

Define η_o**.** Now we define which access points are touched by an action $a = o.m(\vec{u})/\vec{v}$. Let us number the arguments and return values of a, say by setting $w_1 \ldots w_n = \vec{u}\vec{v}$. Also, let us have the β vector associated with a. The action touches the access points:

$$\eta_o(a) = \{o.m{:}\beta{:}ds\} \cup \{o.m{:}\beta{:}i{:}w_i \mid i = 1..n\}$$

Example.

$$\eta_o(\texttt{put}\,(5, 6)\,/\mathit{nil}) = \{\, o.\texttt{put}{:}\beta{:}ds, o.\texttt{put}{:}\beta{:}1{:}5, \\ o.\texttt{put}{:}\beta{:}2{:}6, o.\texttt{put}{:}\beta{:}3{:}\mathit{nil}\,\}$$

Define $\varphi^{m_1}_{m_2}[\beta_1; \beta_2]$**.** We need the ability to plug back the values of two β vectors into a specification formula $\varphi^{m_1}_{m_2}$. Let m_1, m_2 be two methods with specification $\varphi^{m_1}_{m_2} \in \Phi$, and let β_1, β_2 be two beta vectors corresponding to some actions of m_1 and m_2. Denote the normalization of an atomic subformula $q \in \Phi$ by $\hat{q} \in B(\Phi)$. Define $\varphi^{m_1}_{m_2}[\beta_1; \beta_2]$ to be the result of substituting the atomic subformulas of $\varphi^{m_1}_{m_2}$ by:

$$q \mapsto \begin{cases} \beta_1(\hat{q}) & \text{if } var(q) \subseteq V_1 \\ \beta_2(\hat{q}) & \text{if } var(q) \subseteq V_2 \end{cases}$$

Each q gets replaced depending on whether it refers to parameters of m_1 or parameters of m_2.

Note that due to Lemma 6.4 and the definition of L_B, the resulting formula $\varphi^{m_1}_{m_2}[\beta_1; \beta_2]$ is equivalent to one in L_S.

Example. Let $m_1 = \texttt{put}$, β_1 map the atomic formula $k = v$ to *false* and $m_2 = \texttt{get}$. As $B(\Phi, \texttt{get}) = \varnothing$ we have that $\beta_2 = \{\}$. Then, by substitution, we obtain:

$$\varphi^{\texttt{get}}_{\texttt{put}}[\beta_1; \beta_2] \equiv k_1 \neq k_2 \vee \mathit{false}$$

Define $\mathcal{C}_o$**.** Finally, we obtain the conflict relation $\mathcal{C}_o$. Define $\mathcal{C}_o$ as the symmetric closure of the relation R described next. For every $\varphi^{m_1}_{m_2} \in \Phi$ we set:

1. $(o.m_1{:}\beta_1{:}ds, o.m_2{:}\beta_2{:}ds) \in R$ iff $\varphi^{m_1}_{m_2}[\beta_1; \beta_2] \equiv \mathit{false}$
2. $(o.m_1{:}\beta_1{:}i{:}u, o.m_2{:}\beta_2{:}j{:}u) \in R$ iff $\varphi^{m_1}_{m_2}[\beta_1; \beta_2] \not\equiv \mathit{false}$ and contains a conjunct $x_i \neq y_j$
3. $(pt_1, pt_2) \notin R$ for all other access points pt_1, pt_2

Example. For the methods `put` and `get` in Fig. 6 we obtain the conflicts:

$$(o.\texttt{put}{:}\beta_1{:}1{:}u, o.\texttt{get}{:}\varnothing{:}1{:}v) \in \mathcal{C}_o \text{ iff } u = v \text{ and } \neg\beta_1(k = v)$$

Optimizations. Several additional optimizations can be applied to reduce the number of access points and obtain the representation shown in Fig. 7. The steps are worked out as an example in the appendix. It is important to note that the translation above guarantees the mentioned $\Theta(1)$ bound while the optimization steps further reduce the number of access points.

6.3 Equivalence and Complexity

The described translation from ECL specifications to access point representations is correct in the sense of preserving equivalence (*cf.* Definition 4.5).

Application	Benchmark	Performance: qps or seconds			Races: total (distinct)	
		Uninstrumented	FASTTRACK	RD2	FASTTRACK	RD2
H2 database	`ComplexConcurrency`	2011 qps	685 qps	425 qps	1784 (26)	200 (2)
	`ComplexConcurrency` (alternate query distrib.)	1610 qps	601 qps	457 qps	1121 (24)	171 (2)
	`QueryCentricConcurrency`	1666 qps	599 qps	605 qps	209 (4)	0 (0)
	`InsertCentricConcurrency`	1912 qps	622 qps	622 qps	1551 (25)	22 (2)
	`Complex`	1874 qps	1143 qps	989 qps	9 (2)	0 (0)
	`NestedLists`	1893 qps	1086 qps	807 qps	202 (2)	0 (0)
Cassandra	`DynamicEndpointSnitch` test	2.907 s	12.226 s	13.527 s	24 (8)	81 (2)

Table 2. Evaluation of FASTTRACK and RD2 on two industrial applications. The number variables/objects with races are in brackets.

Theorem 6.5. *The access point representation obtained by translating an* ECL *formula* Φ *is equivalent to* Φ.

Now we consider the complexity of the obtained access point representation. For each method there are a finite number of possible β vectors because specifications are finite and therefore there are only a finite number of atomic L_B subformulas appearing. Also, if $o.m_1{:}\beta_1{:}i{:}u$ conflicts with any $o.m_2{:}\beta_2{:}j{:}v$ then v must equal u, that is, the number of such conflicts is a function of the number of conjuncts in the specification. We obtain:

Theorem 6.6. *In the resulting translation shown above, each access point conflicts with a bounded number of other access points.*

The bound depends on the size of the logical specification. This property is why we are able to obtain good asymptotic complexity of the race detection algorithm (see Section 5.4).

7. Evaluation

In this section, we present a preliminary evaluation of our approach. We focus on answering the following two questions: (i) is the overhead imposed by the commutativity race detector tolerable in practice as compared to state of the art low-level race detectors? (ii) is commutativity race detection effective for finding bugs?

To answer these questions, we implemented our commutativity analysis in a tool called RD2. RD2 is implemented in RoadRunner [4], an extensible dynamic analysis framework for Java. Then, we evaluated against two industrial Java applications and corresponding benchmarks, described below.

H2 database server. H2 is an open-source JDBC SQL database server. H2 recently added a Multi-Version Store (MVStore) which permits read operations to examine older versions of the data (*i.e.* Snapshot Isolation). The implementation is built upon several `ConcurrentHashMap`s, which we instrumented. We used H2 version 1.3.174 and applied the Pole Position open source benchmark against the database. Pole Position is a framework that compares how different SQL databases compare against a suite of benchmark scenarios, called "circuits". We ran five "circuits" of the benchmark, three of which are focused on concurrency: `ComplexConcurrency`, `QueryCentricConcurrency`, `InsertCentricConcurrency` and two of which do not involve concurrent queries: `Complex` and `NestedLists`. We ran each of the benchmarks for two minutes and calculated the number of queries per second (qps) that the H2 database handles.

Cassandra 2.0. The Apache Cassandra database is an open-source distributed database. Cassandra maintains a performance rank of the database nodes. Its component `DynamicEndpointSnitch` calculates this rank by continuously accumulating statistics from the various nodes using `ConcurrentHashMap`s. In this work, we ran our analysis against the `DynamicEndpointSnitch` test case which simulates dynamically changing node latencies.

Effectiveness. To measure effectiveness of the race detectors, we compared the following settings for each of the benchmarks: (i) uninstrumented code (ii) using the FASTTRACK [3] low-level race detector, and (iii) using RD2.

Then, we compared the runtime overheads and the reported races, summarized in Table 2. For FASTTRACK, we show the total number of races and distinct variables on which they occur, and for RD2 we show the total number of commutativity races and the distinct objects on which they occur. In terms of overhead, we found that RD2 imposes a performance penalty similar to FASTTRACK, which is expected because even in RD2, RoadRunner instruments reads and writes to all memory locations (plus the `ConcurrentHashMap`s for RD2). We note that if we only instrumented the `ConcurrentHashMap`s objects and *not* the basic memory locations, the overhead of RD2 would be lower.

In terms of precision, most races are highly redundant (meaning that they occur on the same memory locations or on the same concurrent hash map objects). RD2 discovered the following new and interesting commutativity races, which we believe are worth inspecting by the developers:

1. Concurrent accesses to the `freedPageSpace` map in the MVStore of H2 could lead to incorrect state of the server. The bug is already fixed in the latest (yet unreleased as of November 2013) version of the H2 database, but our tool was able to discover it automatically (in the currently released version).
2. Concurrent accesses to the `chunks` map in the MVStore of H2 could lead to the same result being computed multiple times, which might be a performance issue if the computation is expensive.
3. New entries to the `samples` map of the `DynamicEndpointSnitch` class could be added while its size is concurrently used as a performance hint during node rank recalculation, causing the performance hint to become obsolete.

Overall, we believe that commutativity races are a new and interesting indicator of concurrency problems that can be efficiently discovered with a commutativity race detector.

8. Related Work

In this section, we survey some of the more closely related work on commutativity and concurrency analysis. Most of this work falls in the space of optimistic concurrency control.

Commutativity. Commutativity has appeared in various contexts including lower bounds [1], operating system design [2], locking [9], synchronizing abstract data-types [15], concurrency control in transactions [18], thread-level speculation [11], transactional memory [7] and abstract dependence tracking [17].

Our work on the ECL logical fragment is inspired by Kulkarni *et al.*'s [10] treatment of commutativity specifications in the context of parallelization (which also builds on transactional boosting [7]). Their work defined the logical fragment SIMPLE and provided a translation from SIMPLE formulas into abstract locks with modes. However, SIMPLE does not capture many useful specifications such as sets and dictionaries. In their work, SIMPLE was motivated by the fact that a translator should be able to statically generate the abstract locks/modes to be acquired, while for our problem, this restriction is unnecessary. By lifting this restriction, we were able to introduce a more expressive fragment than SIMPLE, called ECL, which can capture practical specifications such as sets and dictionaries. Further, we have shown how to translate formulas in that fragment to access points and have also shown that formulas in ECL enable better complexity of the commutativity analysis. We believe that ECL is of interest in the context of thread-level speculation and transactional memory.

In addition, we note that the domains of optimistic parallelization and race detection are almost diametrically opposite. In optimistic concurrency, one would like to reduce the number of conflicts as much as possible, hence if two transactions do not overlap, even if they access the same abstract locks, none of the transactions would typically abort. In race detection however, even when threads do not overlap in an execution, we would like to find out whether they may have overlapped in another execution and report a conflict. This dictates the need to track a happens-before relation (*e.g.* via vector clocks). Further, we never acquire locks per-se (unlike [7]), meaning that we need not deal with deadlocks as is possible with transactions.

Further, the work of Kim *et al.* [8] provides several sound commutativity specifications (*e.g. AssociationList*) and proves their correctness. We believe that our work serves as a motivation to further provide such commutativity specifications. The work of Shacham et al. [16] found a number of incorrect usages of concurrent objects. They use a procedure which only generates inputs where the operations do not commute and then (bounded) model checks the space of interleavings for that input.

Race and atomicity detection. We see our work as generalizing classic happens-before dynamic race detectors [3] to deal with richer notions of conflict than basic reads and writes. Dynamic atomicity detectors such as Velodrome [5] are another line of work aiming to generalize race detection. However, it may not always be possible to write a natural atomicity specification. For instance, for our running example in Fig. 1, there is no natural atomicity specification (yet, there are commutativity conflicts). Further, we note that similarly to classic race detectors, state-of-the-art dynamic atomicity detectors also use a low-level notion of conflict based on reads and writes, see page 3, right column in Velodrome [5]. This low-level definition of conflict can be extended to handle much richer commutativity specifications (with the appropriate modifications of the atomicity algorithms to deal with access points). Therefore, we believe that the techniques presented in this work are applicable to generalizing atomicity detectors as well.

Checking concurrency patterns. Examples of tools that check properties on library methods are [6] and [13]. In [6], the authors dynamically check two quite restricted forms of single object concurrent typestate violations (not arbitrary typestate). In general, many objects where commutativity analysis is applicable (*e.g.* sets, maps) do not have an intuitive typestate and hence their work cannot handle analysis of such objects. Similarly to [16], the work of Lin *et al.* [13] statically checks for few library usage patterns that are likely to be harmful. Many of these patterns can be framed as instances of commutativity races.

9. Conclusion and Future Work

In this paper we introduced the concept of a *commutativity race* and also presented a dynamic commutativity race detector. Our analysis is a novel combination of access point representations with vector clocks. Further, we introduced a logical fragment (ECL) that captures useful commutativity specifications (*e.g.* maps) and a translation from that fragment to access point representations. A key benefit of ECL is that analysis of objects captured in this fragment require only a constant number of checks per operation as opposed to linear. We implemented a proof-of-concept of our approach and demonstrated its effectiveness in practice. Conceptually, our work can be seen as a generalization of traditional race detectors.

Interesting items for future work include extending ECL, generalizing other analyzers (*e.g.* atomicity) and providing optimizing compilers from ECL-like logics to access point representations.

References

[1] ATTIYA, H., GUERRAOUI, R., HENDLER, D., KUZNETSOV, P., MICHAEL, M., AND VECHEV, M. Laws of order: Expensive synchronization in concurrent algorithms cannot be eliminated. In *ACM POPL'11*.

[2] CLEMENTS, A. T., KAASHOEK, M. F., ZELDOVICH, N., MORRIS, R. T., AND KOHLER, E. The scalable commutativity rule: Designing scalable software for multicore processors. In *ACM SOSP'13*.

[3] FLANAGAN, C., AND FREUND, S. N. Fasttrack: Efficient and precise dynamic race detection. In *ACM PLDI'09*.

[4] FLANAGAN, C., AND FREUND, S. N. The roadrunner dynamic analysis framework for concurrent programs. In *ACM PASTE'10*.

[5] FLANAGAN, C., FREUND, S. N., AND YI, J. Velodrome: A sound and complete dynamic atomicity checker for multithreaded programs. In *ACM PLDI'08*.

[6] GAO, Q., ZHANG, W., CHEN, Z., ZHENG, M., AND QIN, F. 2ndstrike: Toward manifesting hidden concurrency typestate bugs. In *ACM ASPLOS'11*.

[7] HERLIHY, M., AND KOSKINEN, E. Transactional boosting: A methodology for highly-concurrent transactional objects. In *ACM PPoPP '08*.

[8] KIM, D., AND RINARD, M. C. Verification of semantic commutativity conditions and inverse operations on linked data structures. In *ACM PLDI'11*.

[9] KORTH, H. F. Locking primitives in a database system. *Journal of the ACM'83*.

[10] KULKARNI, M., NGUYEN, D., PROUNTZOS, D., SUI, X., AND PINGALI, K. Exploiting the commutativity lattice. In *ACM PLDI'11*.

[11] KULKARNI, M., PINGALI, K., WALTER, B., RAMANARAYANAN, G., BALA, K., AND CHEW, L. P. Optimistic parallelism requires abstractions. In *ACM PLDI'07*.

[12] LAMPORT, L. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM'78*.

[13] LIN, Y., AND DIG, D. Check-then-act misuse of java concurrent collections. In *IEEE ICST '13*.

[14] MATTERN, F. Virtual time and global states of distributed systems. In *Proc. Workshop on Parallel and Distributed Algorithms'88*.

[15] SCHWARZ, P. M., AND SPECTOR, A. Z. Synchronizing shared abstract types. *ACM Trans. Comput. Syst.'84*.

[16] SHACHAM, O., BRONSON, N., AIKEN, A., SAGIV, M., VECHEV, M., AND YAHAV, E. Testing atomicity of composed concurrent operations. In *ACM OOPSLA'11*.

[17] TRIPP, O., YORSH, G., FIELD, J., AND SAGIV, M. Hawkeye: Effective discovery of dataflow impediments to parallelization. In *ACM OOPSLA'11*.

[18] WEIHL, W. E. Commutativity-based concurrency control for abstract data types. In *Twenty-First Annual Hawaii International Conference on Software Track'88*.

A. Appendix

A.1 Proofs

Proof of Theorem 5.1. Let $e \in \pi$ be the current event that is analyzed. Let pt' be an access point and let $\{e_i\}_{i=1}^{n}$ be the set of all events in which pt' was previously touched by some action. The second phase of the algorithm maintains the invariant that $pt'.vc = vc(e_1) \sqcup \cdots \sqcup vc(e_n)$ and $pt' \in active(o)$ iff $n > 0$. Also, from the definitions of $\sqcup$ and $\sqsubseteq$, for any $c \in VC$ we have that $vc(e_1) \sqcup \cdots \sqcup vc(e_n) \not\sqsubseteq c$ iff $e_i \not\sqsubseteq c$ for some e_i.

It is enough to show that the first phase of Algorithm 1 will report a race iff there is a race between e and some other event $e' \leq_\pi e$. A race is reported iff there is pt' such that $pt'.vc \not\sqsubseteq vc(e)$ and $(pt, pt') \in \mathcal{C}_o$. Because $pt'.vc = vc(e_1) \sqcup \cdots \sqcup vc(e_n)$, this happens iff there exist some $e_i = e'$ such that $vc(e') \not\sqsubseteq vc(e)$ and $(pt, pt') \in \mathcal{C}_o$. The last is equivalent to $(pt, pt') \in \mathcal{C}_o$ and $e' \leq_\pi e$, but $e' \not\preceq e$ which is equivalent to a commutativity race because we work with a representation (Definition 4.5).

□

Proof of Theorem 6.5. Let $\varphi_{m_2}^{m_1}(x_1 \ldots x_k; y_1 \ldots y_l) \in \Phi$, and let a and b be some actions of m_1 and m_2 respectively. It is enough to show that that $\varphi_{m_2}^{m_1}(a, b) \equiv \mathit{false}$ iff there exist two access points $pt_1 \in \eta_o(a)$, $pt_2 \in \eta_o(b)$ such that $(pt_1, pt_2) \in \mathcal{C}_o$.

Let β_1 and β_2 be the corresponding beta vectors of a and b. By the definition of β_1 and β_2 we have $\varphi_{m_2}^{m_1}(a, b) \equiv \mathit{false}$ iff $\varphi_{m_2}^{m_1}[\beta_1; \beta_2](a, b) \equiv \mathit{false}$. The formula $\varphi_{m_2}^{m_1}[\beta_1; \beta_2]$ is equivalent to a L_S formula, that is, conjunction of literals of the form $x \neq y$. Therefore, $\varphi_{m_2}^{m_1}[\beta_1; \beta_2](a, b) \equiv \mathit{false}$ iff $\varphi_{m_2}^{m_1}[\beta_1; \beta_2] \equiv \mathit{false}$ or some of the mentioned conjuncts evaluates to false. By the definition of $\mathcal{C}_o$ this is the case iff there are two access points $pt_1 \in \eta_o(a)$, $pt_2 \in \eta_o(b)$ such that $(pt_1, pt_2) \in \mathcal{C}_o$.

□

A.2 Translating the dictionary specification

Consider the commutativity specification Φ of a dictionary object o from Fig. 6. The specification fits the ECL fragment and we will show how to obtain an access point representation from it. The result, however, will not match the one in Fig. 7 which can be seen as an optimized variant. Later, we will consider a few automatic transformations which can achieve this optimization.

We first have to find out the access points for each of the methods. To determine the shape of the β components, for each method we need to find out the L_B atomic subformulas occurring in the specification. Actually, only `put` has any:

$$B(\Phi, \texttt{put}) = \{v = p, v = \mathit{nil}, p = \mathit{nil}\}$$

Omitting access points that will not appear in the conflict relation, we have:

$$\begin{aligned} \eta_o(\texttt{put(}k\texttt{,}v\texttt{)/}p) &= \{o.\texttt{put}{:}\beta{:}ds, o.\texttt{put}{:}\beta{:}1{:}k\} \\ \eta_o(\texttt{get(}k\texttt{)/}v) &= \{o.\texttt{get}{:}\varnothing{:}1{:}k\} \\ \eta_o(\texttt{size()/}r) &= \{o.\texttt{size}{:}\varnothing{:}ds\} \end{aligned}$$

Where β stands for the beta vector of the corresponding action for which η_o is being defined.

Now we have to define the conflict relation $\mathcal{C}_o$. We will do this only for the `put` method, the rest being similar. Directly following the definition from Section 6.2 we obtain the conflicts:

$$\begin{aligned} &(o.\texttt{put}{:}\beta_1{:}1{:}u, o.\texttt{put}{:}\beta_2{:}1{:}v) \in \mathcal{C}_o \text{ iff} \\ &\qquad u = v \wedge (\neg\beta_1(k = v) \vee \neg\beta_2(k = v)) \\ &(o.\texttt{put}{:}\beta_1{:}1{:}u, o.\texttt{get}{:}\varnothing{:}1{:}v) \in \mathcal{C}_o \text{ iff} \\ &\qquad u = v \wedge \neg\beta_1(k = v) \\ &(o.\texttt{put}{:}\beta_1{:}ds, o.\texttt{size}{:}\varnothing{:}ds) \in \mathcal{C}_o \text{ iff} \\ &\qquad \neg(\beta_1(v = \mathit{nil}) \iff \beta_1(p = \mathit{nil})) \end{aligned}$$

However, the result differs from the access point representation given in Fig. 7. The two, however, are equivalent with respect to Definition 4.5, as can be shown by applying several transformations.

A.3 Simplifying the obtained access point representation

Let us continue the example and examine several rules that can be applied in order to simplify the access point representation of the specification Φ on Fig. 6.

Consolidation. The domain of `put`'s β vector is the set of atoms $B(\Phi, \texttt{put}) = \{v = p, v = \mathit{nil}, p = \mathit{nil}\}$. However, the two atoms $v = \mathit{nil}$ and $p = \mathit{nil}$ are only used in the L_B subformula $(v = \mathit{nil} \wedge p = \mathit{nil}) \vee (v \neq \mathit{nil} \wedge p \neq \mathit{nil})$. To shorten the notation we will use the equivalent $v = \mathit{nil} \iff p = \mathit{nil}$. This formula can replace the two atoms in the domain of β:

$$\beta : \{v = p, v = \mathit{nil} \iff p = \mathit{nil}\} \to \{\mathit{true}, \mathit{false}\}$$

We can consolidate such atoms because their separate valuation is irrelevant.

Dropping. Note that the two subformulas $v = p$ and $v = \mathit{nil} \iff p = \mathit{nil}$ never appear together in a single formula. That is why conflicts with $o.\texttt{put}{:}\beta{:}1{:}u$ do not depend on the truth value of $v = \mathit{nil} \iff p = \mathit{nil}$ and conflicts with $o.\texttt{put}{:}\beta{:}ds$ do not depend on the truth value of $v = p$. Therefore, in the corresponding access points we can drop the unneeded atoms from the domain of the definition of β. We obtain four possible β vectors:

$$\begin{aligned} r &= \{p = v \mapsto \mathit{true}\} \\ w &= \{p = v \mapsto \mathit{false}\} \\ \mathit{noresize} &= \{v = \mathit{nil} \iff p = \mathit{nil} \mapsto \mathit{true}\} \\ \mathit{resize} &= \{v = \mathit{nil} \iff p = \mathit{nil} \mapsto \mathit{false}\} \end{aligned}$$

Now, we make the replacement:

$$\begin{aligned} &o.\texttt{put}{:}\beta{:}1{:}u \to o{:}\texttt{r}{:}u \text{ iff } \beta(k = v) \\ &o.\texttt{put}{:}\beta{:}1{:}u \to o{:}\texttt{w}{:}u \text{ iff } \neg\beta(k = v) \\ &o.\texttt{put}{:}\beta{:}ds \to o{:}\texttt{noresize} \text{ iff } \beta(k = \mathit{nil} \iff p = \mathit{nil}) \\ &o.\texttt{put}{:}\beta{:}ds \to o{:}\texttt{resize} \text{ iff } \neg\beta(k = \mathit{nil} \iff p = \mathit{nil}) \end{aligned}$$

In general, parts of β can be dropped when they do not influence the conflict relation $\mathcal{C}_o$.

Cleanup. With the definitions above, the access point representation becomes:

$$\begin{aligned} (o{:}\beta_1{:}u, o{:}\beta_2{:}v) &\in \mathcal{C}_o \text{ iff } u = v \wedge (\beta_1 = \texttt{w} \vee \beta_2 = \texttt{w}) \\ (o{:}\beta_1{:}u, o.\texttt{get}{:}\varnothing{:}1{:}v) &\in \mathcal{C}_o \text{ iff } u = v \wedge (\beta_1 = \texttt{w}) \\ (o{:}\beta_1, o.\texttt{size()}{:}\varnothing{:}ds) &\in \mathcal{C}_o \text{ iff } \beta_1 = \texttt{resize} \end{aligned}$$

The access point $o{:}\texttt{noresize}$ is not referenced at all. Such access points are unnecessary and can be safely removed.

Replacement. A non-trivial transformation involves the fact that as far as the conflict relation $\mathcal{C}_o$ is concerned, the access points $o{:}r{:}v$ and $o.\texttt{get}{:}\varnothing{:}1{:}v$ carry the same information, that is:

$$(o{:}\beta_1{:}u, o{:}r{:}v) \in \mathcal{C}_o \text{ iff } (o{:}\beta_1{:}u, o.\texttt{get}{:}\varnothing{:}1{:}v) \in \mathcal{C}_o$$

Therefore, $o{:}r{:}v$ can safely be substituted for $o.\texttt{get}{:}\varnothing{:}1{:}v$, obtaining:

$$\eta_o(\texttt{get(}k\texttt{)/}v) = \{o{:}r{:}v\}$$

and of course, removing the now redundant access point.

In general, two access points pt_1 and pt_2 are congruent iff for any third one pt_3 we have $(pt_1, pt_3) \in \mathcal{C}_o$ iff $(pt_2, pt_3) \in \mathcal{C}_o$. Each access point in a congruence class can be replaced with a single representative from that class.