Checking Linearizability of Encapsulated Extended Operations*

Oren Zomer[1], Guy Golan-Gueta[1], G. Ramalingam[2], and Mooly Sagiv[1]

[1] Tel Aviv University, Tel Aviv, Israel
[2] Microsoft Research, Bangalore, India

Abstract. Linearizable objects (data-structures) provide operations that appear to execute atomically. Modern mainstream languages provide many linearizable data-structures, simplifying concurrent programming. In practice, however, programmers often find a need to execute a sequence of operations (on linearizable objects) that executes atomically and write *extended operations* for this purpose. Such extended operations are a common source of atomicity bugs.

This paper focuses on the problem of verifying that a set of extension operations (to a linearizable library) are themselves linearizable. We present several reduction theorems that simplify this verification problem enabling more efficient verification.

We first introduce the notion of an *encapsulated extension*: this is an extension that (a) does not introduce new shared state (beyond the shared state in the base linearizable library), and (b) accesses or modifies the shared state only through the base operations. We show that encapsulated extensions are widely prevalent in real applications.

We show that linearizability of encapsulated extended operations can be verified by considering only histories with one occurrence of an extended operation, interleaved with atomic occurrences of base and extended operations. As a consequence, this verification needs to consider only histories with two threads, whereas general linearizability verification requires considering histories with an unbounded number of threads.

We show that when the operations satisfy certain properties, each extended operation can be verified independently of the others, enabling further reductions.

We have implemented a simple static analysis algorithm that conservatively verifies linearizabilty of encapsulated extensions of Java concurrent maps. We present empirical results illustrating the benefits of the reduction theorems.

Keywords: concurrency, linearizability, atomicity, verification, composition, extension.

* Zomer, Gueta, and Sagiv were funded by the European Research Council under the European Unions Seventh Framework Program (FP7/2007-2013) / ERC grant agreement no. [321174-VSSC].

Z. Shao (Ed.): ESOP 2014, LNCS 8410, pp. 311–330, 2014.
© Springer-Verlag Berlin Heidelberg 2014

1 Introduction

Concurrent programs are challenging to write. To ease the programmer's burden, modern programming platforms provide libraries of efficient concurrent data structures. These libraries provide operations that are guaranteed to be *atomic*, while hiding the complexity of the implementation from clients.

Unfortunately, clients often need to atomically perform some computation that may invoke multiple library operations. Programmers end up extending a linearizable data type by defining new custom atomic operations, which we refer to as *extended operations*. Figure 1 and Figure 2 are real world examples of linearizable operations that extend the Java *ConcurrentMap* interface. As shown in [1], such extended operations are a common source of concurrency bugs. In this paper, we consider the problem of verifying the correctness of an extension of a linearizable data-structure. Specifically, we wish to verify that the extension of the data-structure is linearizable [2].

Encapsulated Extension. In this paper, we identify a restricted class of extensions of a data-structure, inspired by the examples in [1]. This class is realistic and includes many commonly found extensions. As we show, this class is also amenable to more efficient verification. An extension is said to be *encapsulated* if it satisfies the following two restrictions:

Encapsulation. The extension methods do not directly access or modify any global (shared) state. Instead, extension methods access shared state only via operations of the underlying data-structure that is being extended.

Open Environment. All of the operations of the underlying data-structure are exposed to the clients: i.e., none of the underlying operations are hidden by the extension.

A Simple Verification Approach. Informally, an execution in which multiple threads invoke a data-structure's operations concurrently is said to be *linearizable* if each invoked operation appears to execute instantaneously, with the result that the data-structure's operations appear to be executed sequentially (without any overlap). The data-structure is said to be *linearizable* if all possible concurrent executions involving the data-structure are linearizable.

Consider a data-structure with core methods $m_1, \ldots, m_n$ that has been extended by adding extension methods $em_1, \ldots, em_k$. We can verify that the extended ADT is linearizable by considering all executions of the following "driver" program, and verifying that each of these executions is linearizable (We write $s_1|s_2$ to indicate that either s_1 or s_2 may be executed non-deterministically). For simplicity we have omitted parameters and return-values in this code template.

```
Val computeIfAbsent(Key k) {
  Val temp1, temp2 ;
  temp1 = this.get(k) ;
  if (temp1 == null) {
    temp2 = hardLocalPureStateComputation(k) ;
    temp1 = this.putIfAbsent(k, temp2) ;
    if (temp1 == null) temp1 = temp2 ;
  }
  return temp1 ;
}
```

Fig. 1. A linearizable operation that extends Java *ConcurrentMap*. The pure computation can be, for example if `k` is an integer, evaluating the square of `k`. The `get` operation in *ConcurrentMap* returns the value mapped from the given key (initialized to `null`). The `putIfAbsent` operation in *ConcurrentMap* atomically checks whether the given key is mapped to `null`: if it is mapped to `null`, the operation immediately maps the key to the given value and returns `null`, otherwise the operation returns the non-null value that the key is mapped to without changing the map.

```
void inc(Class<?> key) {
  for (;;) {
    Integer i = this.get(key);
    if (i == null) {
      if (this.putIfAbsent(key, 1) == null) return;
    } else {
      if (this.replace(key, i, i + 1)) return;
    }
  }
}
```

Fig. 2. An extended encapsulated operation over Java *ConcurrentMap* from *OpenJDK 7*, class: *ThrowingTasks*. The `replace` operation atomically checks whether the given key is mapped to the first value: if it is mapped to that value, the operation immediately remaps the key to the second value and returns true, otherwise the operation returns false without changing the map.

```
while (∗) do {
  create new thread to execute {
    while (∗) do {
      m_1() | ··· | m_n() | em_1() | ··· | em_k();
    }
  }
}
```

Incremental Verification. Suppose the core ADT (consisting only of the core methods) is known to be linearizable. We can then exploit this to simplify the driver program as shown below, replacing each call to a core method m_i by "`atomic` s_i", where s_i is the sequential specification for m_i. (Note that this replacement is done within the code for any extension method em_j as well, even though that is not shown below.)

```
while (*) do {
  create new thread to execute {
    while (*) do {
      atomic {s_1()} | ··· | atomic{s_n()} | em_1() | ··· | em_k();
    }
  }
}
```

Note that this reduction is valid only because of the "encapsulation" assumption stated earlier. If the code for extended operations directly accesses or manipulates the shared state (of the underlying data-structure), this reduction is invalid. However, accessing this shared state via the core operations is fine.

Reduction to Two Threads. As we show in the paper, it is not necessary to consider all executions of the preceding driver program. Using induction, we show that it suffices to consider a single occurrence of any one extended operation and replace other occurrences of an extended operation em_i by an atomic execution of its sequential specification es_i. If the implementations and specifications do not depend on thread identifiers (such as Java *ThreadLocal* class), we can rewrite the driver program so that it contains only two threads (since all atomic executions of operations can be treated as executed by the same thread). This gives us the following simplified driver program:

```
// Thread 1 (Environment thread)
while (*) do {
 atomic { s_1() } | ··· | atomic { s_n() } | atomic { es_1() } | ··· | atomic { es_k() };
}
||
// Thread 2 (Nonatomic extension method)
{ em_1() | ··· | em_k(); }
```

Note that such a reduction is not possible for general linearizability verification. Consider the simple example shown in Figure 3, which is not linearizable. However, all histories of this example with less than `K` threads are linearizable. Hence, the verifier will find a counterexample only when it considers executions with K threads.

Proving linearizability is intractable even for finite systems, in general [3]. However, bounding the number of threads reduces the complexity of linearizability verification (see [3]).

Further Reductions. We then describe additional conditions (explained later) that, when satisfied, allow us to verify the linearizability of the extension operations em_1 to em_k independent of each other. These conditions, in fact, allow us

to verify the linearizability of the executions produced by the following driver program, for each i, independently.

```
// Environment thread
while (*) do {
  atomic { s_1() } | ··· | atomic { s_n() };
}
||
// One nonatomic extension method:
{ em_i() }
```

Such a reduction is not always valid, even when we have only one extension operation, as demonstrated by the example in Figure 4. This extension method sets a boolean register to true and returns the original value. This method is not linearizable. Assume that the initial value of the register is false and that there are two concurrent invocations of the extension method. It is possible for both invocations to return a value of false, which is not possible in any sequential execution. However, any execution that contains only one occurrence of the extension method (along with any number of occurrences of the core methods `read` and `write`) can be shown to be linearizable.

```
int s = 0;
// Specifications:
//   return value must be true.
// K is a constant value larger
// than 1.
boolean incReadAssertDec() {
  s++;
  boolean b = (s < K);
  s−−;
  return b;
}
```

Fig. 3. A simple example of a method which is linearizable for up to `K` threads. (we assume that the operations on s are atomic.)

```
boolean readAndWriteTrue() {
  boolean temp = this.read() ;
  if (!temp) {
    this.write(true) ;
  }
  return temp ;
}
```

Fig. 4. An extension of the interface of a boolean register. The base object has a boolean value and two atomic base-operations: **read()** that returns the boolean value and **write(x)** that overwrites it and returns nothing. This encapsulated extended operation is an incorrect implementation of a simple test-and-set operation.

Empirical Evaluation. Java's concurrent maps are widely used, not surprisingly, since they are a higher-level shared memory abstraction. Our empirical study shows that encapsulated extensions over maps are widely used, and that the reductions described above are applicable to many of these extensions, simplifying the verification. We have implemented a static checker for verifying linearizability of encapsulated extensions of the Java concurrent map.

However, we did not find encapsulated extensions over other interesting data structures, such as queues, stacks and deques, in which non-linearizability might

cause errors. The implementations of these data-structures, such as Java *ConcurrentLinkedQueue* and *ConcurrentLinkedDeque*, do not provide methods for "conditional modifications" like *ConcurrentMap*'s `putIfAbsent` and `replace`. For this reason, in most real-world scenarios, the programmer must call external synchronization mechanisms (such as locks and transactions) in order to implement linearizable extensions. This type of extensions contradicts our Encapsulation requirement and therefore it is not in the range of this paper. On the other hand, if those data structures had provided base operations for "conditional modifications", we could write interesting encapsulated extensions on top of them, as demonstrated in [4].

2 Concurrent Objects and Linearizability

In this section we review standard terminology relating to concurrent objects (without extended operations) and *linearizability* (as in [2]).

A concurrent execution of an object is modeled by a *history*, which is a finite sequence of method *invocation* and *response events*. We write a *method invocation* as $[_{t.m(arg)}$ where t is a thread name, m is a method name and *arg* denotes the values of actual argument values of the method. We write a *method response* as $]_{t.m}/b$ where t is a thread name, m is a method name and b is the return value. We sometimes write $t.m(arg)/b$ instead of writing the sequence of the two events $[_{t.m(arg)}$, $]_{t.m}/b$ (this is used as a short way to represent an invocation which is immediately followed by its corresponding response). For convenience, we assume that a unique identifier is attached to every event in a history.

A response *matches* an invocation if they have the same thread name and the same method name. A *method call* in a history h is a pair consisting of an invocation and the next matching response in h. An invocation is *pending* in h if no matching response follows the invocation. $complete(h)$ is the subsequence of h consisting of all non-pending invocations and all responses. A history h is *complete* if $h = complete(h)$.

A history h is *sequential* if the first event of h is an invocation, and each invocation, except possibly the last, is immediately followed by a matching response.

A *thread subhistory* $h|t$ of history h is the subsequence of all events in h whose thread names are t. Two histories h and h' are *equivalent* if for every thread t, $h|t = h'|t$.

Definition 1 (well formed history). *A history h is* well formed *if each thread subhistory of h is sequential.*

We assume that all histories that represent object executions are well formed — because, given a concurrent object x, well formed histories represent all reasonable behaviors of x (see [2]).

Definition 2 (Linearization of a history). *We say that a sequential history s is a linearization of a history h, if there exists a history h' such that the following conditions are satisfied:*

- *h' is constructed by appending zero or more responses to h.*
- *$complete(h')$ is equivalent to s.*
- *If a response event e precedes an invocation event e' in h, then the same is true in s.*

Definition 3 (Sequential Specification). *A* sequential specification *of a concurrent object is a set of sequential histories.*

A sequential specification is used to describe the legal behaviors of an object in the absence of concurrency.

Definition 4 (Linearizable Object). *We say that an object x is linearizable with respect to a sequential specification S, if for any feasible history h of x there exists $s \in S$, such that s is a linearization of h.*

Note that, from the above definition, if x is linearizable with respect to S then any feasible sequential history of x is in S. Furthermore, intuitively, any feasible history of x can be seen as a history in which each method call is *atomic*.

3 Linearizability of Encapsulated Extensions

In this section we generalize the model from Section 2 for encapsulated extensions of a linearizable object and present our reduction theorem for proving linearizablity of encapsulated extensions.

3.1 The Problem

Let Base be a specification describing a (base) linearizable object. An encapsulated extension of Base consists of a set of extension methods (including their implementation). The only global (shared) state accessed by the extension methods is the state of Base, which can be accessed only via the methods of Base.

Extended Histories. Consider an execution of an arbitrary concurrent client program that uses the extended object. For our purposes, it suffices to focus on the invocation and response events of the (base and extended) operations of the object. Hence, we model an execution of the object by an *extended-history*, defined to be a finite sequence of method *invocation* and *response events* in which the events can be divided into two types:
(i) *basic events*: represent invocations and responses of base methods of the given object ;
(ii) *extension events*: represent invocations and responses of the extended methods.
Each event in an extended-history is either a basic event or an extension event (and not both). As in Section 2, we assume that a unique identifier is attached to each event in an extended-history.

Figure 5 shows an example for 3 extended histories. In this figure, the events that refer to the *inc* method are extension events and the other events are basic events.

Internal Events. Let h be an extended-history that contains an extension invocation event e_{inv}. We say that a basic event e is executed by e_{inv} in h, if e and e_{inv} have the same thread name, and one of the following conditions is satisfied: (1) e appears between e_{inv} and the next matching response of e_{inv}; (2) e_{inv} is pending in h, and e appears after e_{inv}. We write $h|e_{inv}$ to denote the subsequence of h of all events that are executed by e_{inv}. We say that a basic event e is internal in h if e is executed by an extension invocation event in h.

For example, in the extended-history h_1 from Figure 5, the events that are marked with an underline are executed by the extension event $[_{t_1.inc(c)}$ and therefore they are internal events in h_1. Together they form the subsequence $h_1|[_{t_1.inc(c)}$.

Two Perspectives. An *object perspective* of an extended-history h, denoted by $obj(h)$, is the maximal subsequence of h such that $obj(h)$ does not contain extension events. A *client perspective* of an extended-history h, denoted by $client(h)$, is the maximal subsequence of h such that $client(h)$ does not contain internal events. Figure 5 shows the two perspectives of an extended history.

Definition 5 (well formed extended-history). *We say that an extended-history h is* well formed *if: (1) both $obj(h)$ and $client(h)$ are well formed histories, (2) for every extension invocation e_{inv} in h that is non-pending, $h|e_{inv}$ is complete.*

In the sequel, we consider only well-formed extended-histories.

Definition 6 (Sequential Specification). *A* sequential specification *of an object with extended operations is a set of extended-histories S such that every $s \in S$ is sequential and does not contain internal events.*

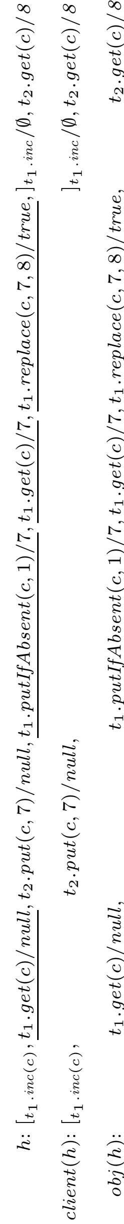


Fig. 5. Example for 3 extended histories of a Map with the extended operation from Figure 2. The histories are executed by threads t_1 and t_2.

Semantics of an Encapsulated Extension. An implementation of a linearizable object x with extended operations defines a set of possible extended histories H_x, defined as follows.

Define an *operation history* (for an extended method m) to be an extended history consisting of an invocation event e of m, followed by a sequence of internal events executed by e, optionally followed by a matching response of e. The semantics of the implementation of m, denoted $[\![m]\!]$ can be formally represented as a set of operation histories (denoting possible executions of a single invocation of m).

Given an extended history h and an extended invocation event e in h, define $h[e]$ to be the sequence $e(h|e)$ if e is pending in h and the sequence $e(h|e)e'$, if e' is the next matching response of e in h. Thus, $h[e]$ represents the operation history corresponding to e.

An extension x of a sequential specification Base consists of a set of extension operations $m_1, \cdots, m_k$. The set of extended histories H_x is defined to be the set of all well-formed extended histories h such that (a) $obj(h) \in$ Base, and (b) For any invocation event e, of an extension operation m_i, in h, we have $h[e] \in [\![m_i]\!]$.

The above definition captures the possible behaviors of x when used with any linearizable implementation of Base. The following definition thus captures the intuition that x should work correctly when used with any correct implementation of Base.

Definition 7 (Linearizable Encapsulated Extension). *We say that the encapsulated extension x is linearizable with respect to a sequential specification S, if for every $h \in H_x$ there exists $s \in S$ such that s is a linearization of $client(h)$.*

3.2 The Reduction Theorem

Properties of Extended-Objects. It can be checked that the set H_x satisfies the following properties, for any extended history h:

(1) if $h \in H_x$ and h' is a well-formed subsequence of h such that $obj(h) = obj(h')$, every internal event in h' is executed by the same extension invocation in h' as in h, and every extension response in h' matches the same extension invocation in h' as in h, then $h' \in H_x$.
(2) if $obj(h) \in H_x$ and for every extension invocation event e_{inv} there is $h' \in H_x$ in which $h[e_{inv}] = h'[e_{inv}]$, then $h \in H_x$.

Condition (1) means that we can create a history in H_x by omitting some of the extension invocation events with their next matching responses (or without them if they are pending). This ensures that the behaviour of the concurrent object is not affected by the extended events. This condition is satisfied because the concurrent object's state is only accessed by its client API.

Condition (2) means that the behavior of an extended method only depends on its arguments and its interaction with the base object. This condition is satisfied because the only shared state (between threads) is the state of the concurrent object.

Reduction Theorem. Let e_{inv} be an invocation event in an extended-history h. We say that e_{inv} is *interrupted* if there exists an event e in h such that: (i) e_{inv}

and e have a different thread name; (ii) e appears after e_{inv}; (iii) e does not appear after the matching response of e_{inv}. For example, in the extended-history h_1 from Figure 5, the event $e_{inc} = [_{t_1.inc(c)}$ is interrupted because the events of $t_2.put(c, 7)/null$ appear between e_{inc} and its matching response.

We write $\#(h)$ to denote the number of interrupted invocation events in h. We write H_x^k to denote $\{h \in H_x \mid \#(h) \leq k\}$, and H_x^0 to denote the subset of sequential histories. $client\left[H_x^0\right] = \left\{client(h) \mid h \in H_x^0\right\}$ is the client perspective of the sequential histories.

We present our reduction theorem below, treating the implementation of the extension itself as its sequential specification. Specifically, we consider the case where $client\left[H_x^0\right]$ is the sequential specification for the extension. In [4] we present a generalization of this theorem which handles general specifications.

Theorem 1 (Reduction Theorem). *If the set H_x^1 is linearizable with respect to $client\left[H_x^0\right]$, then the set H_x is linearizable with respect to $client\left[H_x^0\right]$.*

Proof (Sketch). We use induction to show that for any $n \geq 1$, H_x^n is linearizable with respect to $client\left[H_x^0\right]$. Let's assume that for some $k \geq 1$, H_x^k is linearizable, and prove that H_x^{k+1} is linearizable. Let $h \in H_x^{k+1}$ be a history with $\#(h) = k+1$ which contains an interrupted extension invocation e_{inv}. Let's assume that e_{inv} is not pending and has a matching response e_{res} (the case in which e_{inv} is pending can be shown in a similar way).

Using condition (1), we can remove e_{inv} and e_{res} from h, and get a new history $h' \in H_x$ with $\#(h') = k$. Notice that all internal method calls $h|e_{inv}$ are not internal in h', and appear in $client(h')$. By the induction hypothesis, $client(h')$ is linearizable — let s' be its linearization. s' also contains the subsequence $h|e_{inv}$.

$s' \in client\left[H_x^0\right]$, so let $h'' \in H_x^0$ be a history such that $client(h'') = s'$. We know that $\#(h'') = 0$, and we also know that $h|e_{inv}$ is a subsequence of $client(h'')$.

Let's add e_{inv} to h'' right before the beginning of the subsequence, and e_{res} right after the end of the subsequence, and denote the new history by $\hat{h}$.

$obj(\hat{h}) = obj(h'') \in H_x$, and for any invocation $e'_{inv} \neq e_{inv}$ in $\hat{h}$ we know that $\hat{h}[e'_{inv}] = h''[e'_{inv}]$. Furthermore, for e_{inv} in $\hat{h}$ we know that $\hat{h}[e_{inv}] = h[e_{inv}]$. Together, we can apply condition (2), so $\hat{h} \in H_x$.

$\#(\hat{h}) \leq \#(h'')+1 = 1$, so by the induction hypothesis $client(\hat{h})$ is linearizable. $client(\hat{h})$ can be created from $client(h)$ by omitting some pending invocations, appending matching responses to other pending invocations, and moving e_{inv} and e_{res} closer to each other. The order of operations in $client(\hat{h})$ preserves the order of operations in $client(h)$, and therefore the linearization of $client(\hat{h})$ is also a linearization of $client(h)$, which means that $client(h)$ is linearizable.

The complete proof is presented in [4].

4 Non-interfering Linearizable Extensions

In this section, we consider conditions under which different linearizable extensions of a concurrent object do not interfere with each other. Specifically, let

$em_1, \cdots, em_k$ be encapsulated extended operations of a concurrent object *Base*. Suppose that, for each i, $\{em_i\} \cup Base$ is linearizable. We present sufficient conditions under which $\{em_1, \cdots, em_k\} \cup Base$ is guaranteed to be linearizable. When these conditions hold, verifying linearizability of an encapsulated extension is further simplified as each extended operation can be independently verified. Many extended operations in the programs in our empirical studies satisfy these conditions.

Recall that a method call is a pair of events of the form $[_{t.m(a)}]_{t.m}/b$ which we also refer as $t.m(a)/b$. In the sequel, we may refer to $m(a)/b$ when the thread name is irrelevant or can be understood from the context.

Given a sequence $\alpha = c_1 \cdots c_m$, where each c_i is a method call of the form $m_i(a_i)/b_i$, we define $t.\alpha$ to be the sequential history $t.c_1 \cdots t.c_m$. We denote $\mathcal{M}$ to the set of base method calls, and $\mathcal{M}_E$ to denote the set of both base method calls and extension method calls.

4.1 Replaceability

We first introduce a notion of *replaceability*.

Let $c \in \mathcal{M}_E$ be some method call and let $M \subseteq \mathcal{M}_E$ be a set of method calls. We say that $c \propto M$ if for every concurrent history $\alpha(t.c)\beta$ in *client* $[H_x]$ there is some $c' \in M$ such that $\alpha(t.c')\beta$ is in *client* $[H_x]$. For example:

$$\texttt{readAndWriteTrue}()/\texttt{false} \propto \{\texttt{write}(\texttt{true})/\texttt{ok}\}$$

$$\texttt{computeIfAbsent}(3)/4 \propto \{\texttt{get}(3)/4\}$$

$$\texttt{computeIfAbsent}(3)/9 \propto \{\texttt{get}(3)/9, \texttt{put}(3,9)/\texttt{null}\}$$ [1]

We say that a method call c is *replacement equivalent to* M if $c \propto M$ and for every $c' \in M$ we have $c' \propto \{c\}$. We say that c is *replaceable by* M if c is replacement equivalent to some subset of M. We say that a method is *replaceable by* M if each of its method calls is replaceable by M.

Recall that H_x^1 denotes the set of extended histories of x containing at most one occurrence of an interrupted invocation event. For any set of method calls $M \subseteq \mathcal{M}_E$, let H_M^1 denote the subset of histories from H_x^1 in which all the uninterrupted operations are in M.

Lemma 1. *If c is replaceable by a set of method calls M, and all the histories in H_M^1 are linearizable, then all the histories in $H_{M\cup\{c\}}^1$ are linearizable.*

Proof. Assume that all histories in H_M^1 are linearizable. Consider any history $h \in H_{M\cup\{c\}}^1$. Replace all (uninterrupted) appearances of c with other calls from M to get a history h' in H_M^1. Let s' be a linearization of h'. Replace the replacement calls back by c to get a sequential history s, which will be a linearization of h.

Corollary 1. *If every method call in $\mathcal{M}_E \setminus \mathcal{M}$ (of an extended operation) is replaceable by $\mathcal{M}$, then H_x is linearizable iff $H_{\mathcal{M}}^1$ is linearizable.*

[1] The pure computation in `computeIfAbsent` calculates the square of the given key.

Discussion. Consider the examples `computeIfAbsent` (Figure 1) and `inc` (Figure 2). Each of these extension operations are replaceable (by the base map method calls). For example, the method call `computeIfAbsent`(3)/9 is replaceable because in any history we can replace such uninterrupted call with a call to `put` or `get`, as appropriate:

$$\texttt{computeIfAbsent}(3)/9 \propto \{\texttt{get}(3)/9, \texttt{put}(3,9)/\texttt{null}\}$$

and in any other history with these calls, we can replace them back:

$$\texttt{get}(3)/9 \propto \{\texttt{computeIfAbsent}(3)/9\}$$

$$\texttt{put}(3,9)/\texttt{null} \propto \{\texttt{computeIfAbsent}(3)/9\}$$

It follows from the above corollary that these two extension operations are non-interfering. To verify that an extension consisting of this pair of operations is linearizable it suffices to verify that the two extensions consisting of each of these operations separately is linearizable.

Not only does the corollary help decouple the verification of multiple extension operations, it also helps simplify the verification of an extension consisting of a single operation. This is because the set of histories $H^1_{\mathcal{M}}$ we need to check is smaller than the set H_x even when the extension consists of a single operation.

Just as we expect, the above corollary does not apply to the example in Fig. 4. As explained in Section 1, $H^1_{\mathcal{M}}$ (i.e., the set of histories with at most one invocation of `readAndWriteTrue`) is linearizable for this example, but the extension is not linearizable. Corollary 1 does not apply because `readAndWriteTrue`()/`false` is not replaceable — we can take any history in *client* $[H_x]$ and replace a method call `readAndWriteTrue`()/`false` with the method call `write`(`true`)/`ok` to get a new history in *client* $[H_x]$, however, in some histories in *client* $[H_x]$ we cannot replace a method call `write`(`true`)/`ok` back with `readAndWriteTrue`()/`false` and get a history in *client* $[H_x]$ (consider histories where the state of the register before the call is `true`).

4.2 Composition Closure

A sequence of method calls β is said to be *atomically equivalent* to a method call c if for all α, γ, we have $\alpha(t.\beta)\gamma \in obj\,[H_x]$ iff $\alpha(t.c)\gamma \in obj\,[H_x]$. We say that a set of method calls M is *composition-closed* if every sequence of calls from M is atomically equivalent to a single call in M.

Example. A *Generic Register* is a register with three linearizable base operations: `read()` that returns the register's value, `write(x)` that changes the register's value to `x` and returns the register's old value (an unconditional "*swap*"), and a unique operation `compareAndSwap(expect,new)` that changes the register's value to `new` if it equals `expect`, and returns the register's old value (whether it changed or not).

In this example, the sequence write(7)/0 write(8)/7 is atomically equivalent to the call write(8)/0. Furthermore, any sequence of read and write over a generic register (with at least one write):

$$m_1(args_1)/b_1 \dots m_k(args_k)/b_k$$

is atomically equivalent to write($args_j$)/b_1, where j is the index of the last write — this single operation makes the change of the whole sequence atomically, and returns the value of the register before the sequence. Hence, the set of all method calls to {read, write} is composition-closed. This composition-closure property holds even if we include the compareAndSwap operation.

Lemma 2. *The set of all base method calls of a generic register is composition-closed.*

Now let's look at encapsulated operations over any $\mathcal{M}$ that is composition-closed:

Lemma 3. *If $\mathcal{M}$ is composition-closed, then every encapsulated extended operation of the object is replaceable by $\mathcal{M}$.*

Proof. In any extended history, an uninterrupted call of the encapsulated extended operation can be replaced by its internal base operation calls. This sequence of base calls is atomically equivalent to a single base call which can replace it, and vice versa.

By combining Lemma 3 with Corollary 1, we get:

Corollary 2. *Let $\mathcal{M}$, the set of all base method calls of a concurrent object, be composition-closed. For any encapsulated extension x of this object, if $H^1_{\mathcal{M}}$ is linearizable, then H_x (and x) is linearizable.*

The immediate implication of Corollary 2 on the generic register example is that checking the linearizability of $H^1_{\{read()/b, write(a)/b, compareAndSwap(a,b)/c\}}$ guarantees the linearizability of H_x. We can further reduce the set of histories that need to be checked by noticing that every compareAndSwap$(a,b)/c$ is replaceable by the set of read and write operations:

Corollary 3. *For encapsulated extended operations over a generic register, verifying the linearizability of the histories in $H^1_{\{read()/b, write(a)/b\}}$ guarantees the linearizability of every history in H_x.*

In Section 5.2 we show that verifying the linearizability of many real world methods can be reduced to verifying linearizability of encapsulated extended operations over a generic register.

4.3 Further Reductions

Let's look at some history $h \in H^1_{\mathcal{M}}$, assuming $\mathcal{M}$ is composition-closed. Let's assume that there is a sequence $s \in \mathcal{M}^*$ of successive base operations in h, and

the interrupted encapsulated operation does not call any internal operations during that sequence, i.e., its thread is idle. The sequence of base operations is atomically equivalent to some single base operation $m(a)/b \in \mathcal{M}$ that can replace s and give a new history $h' \in H^1_{\mathcal{M}}$. Verifying the linearizability of h' guarantees that h is linearizable, because we can take the linearization of h' and replace $m(a)/b$ back with s.

Let $\overline{H^1_{\mathcal{M}}} \subseteq H^1_{\mathcal{M}}$ be the subset of histories where between every two uninterrupted base operations, the interrupted encapsulated operation must have called an internal operation. By induction, we can conclude:

Lemma 4. *If $\mathcal{M}$ is composition-closed, then verifying the linearizability of $\overline{H^1_{\mathcal{M}}}$ guarantees the linearizability of $H^1_{\mathcal{M}}$, and therefore guarantees the linearizability of H_x.*

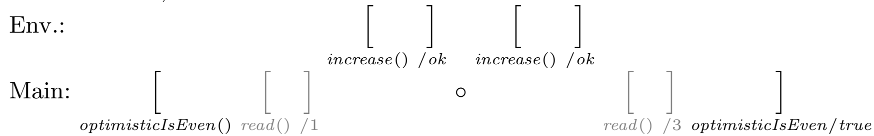
Corollary 4. *For encapsulated extended operations over a generic register, verifying the linearizability of the histories in $\overline{H^1_{\{read()/b, write(a)/b\}}}$ guarantees the linearizability of every history in H_x.*

Corollary 4 subsumes the results of Corollary 3.

In every history $\overline{h} \in \overline{H^1_{\mathcal{M}}}$, between every two uninterrupted base operations, there must be an internal operation of the interrupted call. This means that if $\overline{h}$ is linearizable, the linearization point of the interrupted call may be seen as if it happened in one of its internal operations (or in its invocation/response) — we look at the uninterrupted calls with the linearization points that precede and succeed the one of the interrupted call, find an internal operation (of the interrupted call) that reside between them, and move the linearization point of the interrupted call inside it, without breaking the total order of the linearization points.

Lemma 5. *If $\mathcal{M}$ is composition-closed and $\overline{H^1_{\mathcal{M}}}$ is linearizable, then we can linearize every history in $\overline{H^1_{\mathcal{M}}}$ using linearization points that reside in the same thread.*

Notice the necessity of $\mathcal{M}$ being composition-closed. Figure 6 is an artificial example for a replaceable encapsulated operation over a base data-structure that is not composition-closed — the sequence `increase`$()/ok$ `increase`$()/ok$ is not atomically equivalent to any single operation. In this linearizable example, some histories can be linearized only by picking a linearization point that resides in a different thread, such as:



```
boolean optimisticIsEven() {
  int temp1, temp2;
  temp1 = read();
  if (temp1 % 2 == 0) {
    return true;
  }
  temp2 = read();
  if (temp1 == temp2) {
    return false;
  }
  else {
    return true;
  }
}
```

$\texttt{optimisticIsEven}()/\texttt{true} \propto \{\texttt{read}()/x \mid x \text{ is even}\}$
$\forall x \text{ even} : \texttt{read}()/x \propto \{\texttt{optimisticIsEven}()/\texttt{true}\}$
$\texttt{optimisticIsEven}()/\texttt{false} \propto \{\texttt{read}()/x \mid x \text{ is odd}\}$
$\forall x \text{ odd} : \texttt{read}()/x \propto \{\texttt{optimisticIsEven}()/\texttt{false}\}$

Fig. 6. An encapsulated extended operation over an integer register with two base operations: `read()` that returns the register's value and `increase()` that increases its value by one. `optimisticIsEven()` is replaceable by the base methods.

5 On the Applicability of the Reduction

5.1 Checking Encapsulation of Extended Operations

Checking that a method is an encapsulated extension can be done conservatively by checking that: (1) The method does not access global mutable variables, and (2) all external methods invoked by an encapsulated method are either base-methods or pure.

Out of 109 methods used [5], 55 methods were identified as encapsulated operations, using the technique described in [6]. The base data-structure in all of the 55 methods was the linearizable Java `ConcurrentMap` interface.

5.2 Checking Composition Closure

In general, checking that an encapsulated operation is replaceable (as defined in Section 4) can be hard. It requires verifying all sequential executions, which is undecidable. In contrast, checking composition closure can be done once and for all for a given base data structure. Unfortunately, the Java `ConcurrentMap`, which is heavily used, does not satisfy this closure property since a sequence of operations on different keys is not necessarily equivalent to any single `ConcurrentMap` operation.

We observed 52 out of the 55 operations employ maps in a limited fashion: any single invocation of the operation is guaranteed to invoke map operations on only one key (Note that different execution paths may, however, operate on different keys — Figure 7 is an interesting example which illustrates this). Such operations are guaranteed to be replaceable by `ConcurrentMap`'s base operations. In fact,

the code can be syntactically replaced by an equivalent extended encapsulated operation over a generic register as follows:

$$\begin{aligned}
\texttt{map.get(k)} &\Rightarrow \texttt{reg.read()}\\
\texttt{map.put(k,v)} &\Rightarrow \texttt{reg.write(v)}\\
\texttt{map.remove(k)} &\Rightarrow \texttt{reg.write(null)}\\
\texttt{map.putIfAbsent(k,v)} &\Rightarrow \texttt{reg.compareAndSwap(null,v)}\\
\texttt{map.replace(k,v1,v2)} &\Rightarrow (\texttt{reg.compareAndSwap(v1,v2)} == \texttt{v1})
\end{aligned}$$

Intuitively, checking the linearizability of the new method is equivalent to checking the linearizability of the original method.

The other 3 out of the 55 operations employ either use `size` and `clean` or more than one key and therefore are not considered.

```
final FxLanguage DEFAULT = ...
final ConcurrentMap<FxLanguage, FxValueRenderer> renderers = ...

FxValueRenderer getInstance(FxLanguage language) {
  if (language == null) {
    // default renderer always exists
    return renderers.get(DEFAULT);
  }
  if (!renderers.containsKey(language)) {
    renderers.putIfAbsent(language, new FxValueRendererImpl(language));
  }
  return renderers.get(language);
}
```

Fig. 7. An extended encapsulated operation from *Flexive*, class: *FxValueRendererFactory*. In every execution path, only a single key is used. This example is **not** linearizable — consider the following history: $[_{A.getInstance(L)}, A.containsKey(L)/false, A.putIfAbsent(L, Fx_A)/null, B.put(L, Fx_B)/Fx_A, A.get(L)/Fx_B,]_{A.getInstance}/Fx_B$.

5.3 Checking Linearizaibility via Abstract Interpretation

We implemented a conservative tool to check the linearizability of the 52 examples using the abstract interpreter described in [4]. The tool employs our theoretical results by checking only histories with 2 threads in which the extended operation run once. We verified 24 examples as linearizable (Table 1) and detected 27 linearizabilty violations (Table 2). Our implementation failed to verify one linearizable example due to the abstraction, and issued a false alarm (see Figure 8) — The reason for the failure is an over approximation that did not store the correlations between objects and constants, such as `osFamily` and `"windows"/"unix"`.

Table 1. Encapsulated operations verified as linearizable by the static analysis method presented in [4]

Application Name	Class Name	Code Lines	Verification Time	Abstract States
Apache ServiceMix	SimpleLockManager	11	2375 (ms)	49
Clojure	Namespace	8	2329 (ms)	49
Cometdim	ChatService	7	2169 (ms)	49
DWR	AbstractMapContextScope	14	2357 (ms)	49
ehcache-spring-annotation	CacheAttributeSourceImpl	11	2466 (ms)	49
FindBugs	Profiler	8	2451 (ms)	54
Granite	ExternalizerFactory	14	2559 (ms)	49
GWTEventService	DefaultUserManager	11	1702 (ms)	14
Hazelcast	Log4jFactory	12	2450 (ms)	49
ifw2	PropertyNavigator	14	2794 (ms)	91
ifw2	ReflectiveClone	11	2310 (ms)	49
ifw2	ClassInfo	10	2326 (ms)	49
Jboss	AOPLogger	14	2404 (ms)	49
Jetty	AbstractBayeux	12	2450 (ms)	49
Jetty	OortChatService	7	2185 (ms)	47
Jetty	OortChatService	7	2341 (ms)	47
Jexin	ActiveTemplateMap	8	2341 (ms)	47
Jsefa	InitialConfiguration	14	2388 (ms)	49
Keyczar	StreamCache	12	2502 (ms)	49
OpenJDK	ThrowingTasks	12	3776 (ms)	98
Tammi	StaticPersisterFactory	18	3308 (ms)	122
ProjectTrack	MethodCallRecorder	8	2326 (ms)	48
ProjectTrack	MethodCallRecorder	8	2357 (ms)	49
Yasca	Profiler	8	2356 (ms)	49

Table 2. Encapsulated operations with non-linearizability reports issued by the static analysis. The bold row (autoandroid) is the benchmark from Figure 8 that raised a false alarm.

Application Name	Class Name	Code Lines	Verification Time	Abstract States
Adobe BlazeDS	FIFOMessageQueue	10	1498 (ms)	17
Adobe BlazeDS	FIFOMessageQueue	10	1452 (ms)	17
Annsor	Annsor	11	1405 (ms)	12
Apache Cassandra	SuperColumn	15	1529 (ms)	12
Apache Cassandra	ColumnFamily	21	1498 (ms)	12
Apache MyFaces Trinidad	SessionChangeManager	5	1297 (ms)	8
Apache Tomcat	ApplicationContext	9	1343 (ms)	9
Apache Tomcat	ApplicationContext	8	1374 (ms)	12
Apache Tomcat	ReplicatedContext	6	1436 (ms)	9
Apache CXF	ClassResourceInfo	12	1483 (ms)	24
autoandroid	**AndroidTools**	**16**	**1342 (ms)**	**17**
dyuproject	StandardConvertorCache	10	1438 (ms)	17
dyuproject	StandardConvertorCache	11	1437 (ms)	20
Flexive	MessageBean	7	1390 (ms)	8
Flexive	FxValueRendererFactory	10	1655 (ms)	19
GlassFish	BeanManager	13	1531 (ms)	13
Gridkit	ReflectionPofSerializer	14	1455 (ms)	12
GWTEventService	AutoIncrementFactory	4	1266 (ms)	7
Hazelcast	ClientEndpoint	8	1655 (ms)	12
Hudson	Hudson	10	1307 (ms)	12
JRipples	HessianSkeletonProviderImpl	21	1483 (ms)	19
memcache-client	SockIOpool	10	1327 (ms)	8
Tammi	StaticVariableRegistry	14	1389 (ms)	11
RestEasy	XmlJAXBContextFinder	8	1395 (ms)	13
RestEasy	JsonJAXBContextFinder	5	1732 (ms)	12
RestEasy	JsonJAXBContextFinder	5	1436 (ms)	12
Torque-spring	PersistenceManagerFactory	8	1529 (ms)	12
Webmill	ContextNavigator	9	1545 (ms)	12

```
public static AndroidTools forOsFamily(String osFamily) {
  AndroidTools instance = androidTools.get(osFamily);
  if (instance == null) {
    AndroidTools newInstance = null;
    if (osFamily.equals("windows")) {
      newInstance = new WindowsAndroidTools();
    } else if (osFamily.equals("unix")) {
      newInstance = new UnixAndroidTools();
    } else {
      throw new UnsupportedOperationException(
        "Don't know how to start android tools on " + osFamily);
    }
    instance = androidTools.putIfAbsent(osFamily,newInstance);
    if (instance == null) instance = newInstance;
  }
  return instance;
}
```

Fig. 8. Application Name: *autoandroid*, Class Name: *AndroidTools*. A linearizable encapsulated operation that the verification failed to verify, due to impreciseness of our abstraction.

6 Related Work

Linearizability checking tools can be very effective in identifying bugs and a substantial body of work exists in this space, as discussed below. A distinguishing aspect of our work is that we focus on a special case, namely verifying linearizability of encapsulated extensions of a linearizable object. This problem was motivated by [1] which shows that extended operations of linearizable collections are widespread and are a source of concurrency bugs. While [1] presents a dynamic tool for checking linearizability of extended operations, we focus on static verification of the same.

Modular Reasoning. The basic techniques we utilize have a long history in the literature on modular reasoning techniques for concurrent systems. The idea of using a general client over-approximating the thread environment is common in modular verification. Previous work represented the environment as invariants [7] or relations [8] on the shared state. This idea has also been used early on for automatic compositional verification [9]. In addition, this approach has led to the notion of thread-modular verification for model checking systems with finitely-many threads [10], and has also been applied to the domain of heap-manipulating programs with coarse-grained concurrency [11]. The main ideas in these works is to approximate the thread environment.

Exploiting Atomicity of Methods. [12] shows that schedules where linearizable operations are executed with interruptions need not be generated. This can reduce the number of interleavings that need to be explored. This insight has also been discussed and made use of in the preemption sealing work of [13]. [14,15] use the atomicity proof to simplify the correctness proofs of multithreaded programs. [16] presents a proof calculus for reasoning about concurrent programs with atomic sections. It would be interesting to see if such a proof calculus can be used to simplify the proofs of our reduction theorems.

Dynamic Tools for Finding Linearizabilty Violations. Vyrd [17] is a dynamic checking tool that checks a property similar to linearizability. Line-Up [18] is a dynamic linearizability checker that enumerates schedules.

Static Linearizability Verification. [19] manually proves correctness of several interesting concurrent data structure implementations using rely-guarantee reasoning. The PVS system has been successfully used to semi-automatically verify linearizability [20,21,22] of several interesting programs.

[23] pioneered the idea of using abstract interpretation [24] to develop an automatic over-approximation for checking linearizability. Thus, the algorithm can prove linearizability in certain programs but may fail due to overly conservative abstraction. [25,26] generalize [23] using a thread-centric approach to programs with unbounded number of threads. [27] combines the idea of bounded difference with rely guarantee reasoning and shape abstractions in order to perform fast linearizability checks.

Composing Linearizable Operations. Recently several interesting techniques for enforcing atomicity of sequences of linearizable operations were developed. [28] employs a variation of the join-calculus to compose operations via `DCAS`. [29,6] synthesize locks to enforce atomicity and deadlock freedom. In contrast to these approaches, we focus on understanding the complexity of verifying the linearizability of a special useful class of composed operations.

References

1. Shacham, O., Bronson, N.G., Aiken, A., Sagiv, M., Vechev, M.T., Yahav, E.: Testing atomicity of composed concurrent operations. In: OOPSLA, pp. 51–64 (2011)
2. Herlihy, M.P., Wing, J.M.: Linearizability: a correctness condition for concurrent objects. TOPLAS 12(3) (1990)
3. Alur, R., McMillan, K.L., Peled, D.: Model-checking of correctness conditions for concurrent objects. Inf. Comput. 160(1-2), 167–188 (2000)
4. Zomer, O., Golan-Gueta, G., Ramalingam, G., Sagiv, M.: Checking linearizability of encapsulated extended operations. Technical report, Tel Aviv University (2013), `http://www.cs.tau.ac.il/~ggolan/papers/ESOP14TechRep.pdf`
5. Shacham, O.: Verifying Atomicity of Composed Concurrent Operations. PhD thesis, Tel Aviv University (2012)
6. Golan-Gueta, G., Ramalingam, G., Sagiv, M., Yahav, E.: Concurrent libraries with foresight. In: PLDI, pp. 263–274 (2013)
7. Hoare, C.A.R.: Towards a theory of parallel programming. Operating System Techniques (1972)
8. Jones, C.B.: Specification and design of (parallel) programs. In: IFIP Congress (1983)

9. Clarke Jr., E.: Synthesis of resource invariants for concurrent programs. TOPLAS 2(3), 338–358 (1980)
10. Flanagan, C., Qadeer, S.: Thread-modular model checking. In: Ball, T., Rajamani, S.K. (eds.) SPIN 2003. LNCS, vol. 2648, pp. 213–224. Springer, Heidelberg (2003)
11. Gotsman, A., Berdine, J., Cook, B., Sagiv, M.: Thread-modular shape analysis. In: Proceedings of the 2007 ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2007, pp. 266–277. ACM, New York (2007)
12. Filipovic, I., O'Hearn, P., Rinetzky, N., Yang, H.: Abstraction for concurrent objects. Theoretical Computer Science 411(51-52), 4379–4398 (2010)
13. Ball, T., Burckhardt, S., Coons, K.E., Musuvathi, M., Qadeer, S.: Preemption sealing for efficient concurrency testing. In: Esparza, J., Majumdar, R. (eds.) TACAS 2010. LNCS, vol. 6015, pp. 420–434. Springer, Heidelberg (2010)
14. Flanagan, C., Qadeer, S.: A type and effect system for atomicity. In: PLDI, pp. 338–349 (2003)
15. Musuvathi, M., Qadeer, S.: Iterative context bounding for systematic testing of multithreaded programs. In: PLDI, pp. 446–455 (2007)
16. Elmas, T., Qadeer, S., Tasiran, S.: A calculus of atomic actions. In: POPL, pp. 2–15 (2009)
17. Elmas, T., Tasiran, S., Qadeer, S.: Vyrd: verifying concurrent programs by runtime refinement-violation detection. In: PLDI, pp. 27–37 (2005)
18. Burckhardt, S., Dern, C., Musuvathi, M., Tan, R.: Line-up: a complete and automatic linearizability checker. In: PLDI, pp. 330–340 (2010)
19. Vafeiadis, V., Herlihy, M., Hoare, T., Shapiro, M.: Proving correctness of highly-concurrent linearisable objects. In: PPoPP (2006)
20. Doherty, S., Groves, L., Luchangco, V., Moir, M.: Formal verification of a practical lock-free queue algorithm. In: de Frutos-Escrig, D., Núñez, M. (eds.) FORTE 2004. LNCS, vol. 3235, pp. 97–114. Springer, Heidelberg (2004)
21. Colvin, R., Groves, L., Luchangco, V., Moir, M.: Formal verification of a lazy concurrent list-based set algorithm. In: Ball, T., Jones, R.B. (eds.) CAV 2006. LNCS, vol. 4144, pp. 475–488. Springer, Heidelberg (2006)
22. Gao, H., Hesselink, W.H.: A formal reduction for lock-free parallel algorithms. In: Alur, R., Peled, D.A. (eds.) CAV 2004. LNCS, vol. 3114, pp. 44–56. Springer, Heidelberg (2004)
23. Amit, D., Rinetzky, N., Reps, T.W., Sagiv, M., Yahav, E.: Comparison under abstraction for verifying linearizability. In: Damm, W., Hermanns, H. (eds.) CAV 2007. LNCS, vol. 4590, pp. 477–490. Springer, Heidelberg (2007)
24. Cousot, P., Cousot, R.: Abstract interpretation: A unified lattice model for static analysis of programs by construction of approximation of fixed points. In: POPL, pp. 238–252 (1977)
25. Berdine, J., Lev-Ami, T., Manevich, R., Ramalingam, G., Sagiv, M.: Thread quantification for concurrent shape analysis. In: Gupta, A., Malik, S. (eds.) CAV 2008. LNCS, vol. 5123, pp. 399–413. Springer, Heidelberg (2008)
26. Manevich, R., Lev-Ami, T., Sagiv, M., Ramalingam, G., Berdine, J.: Heap decomposition for concurrent shape analysis. In: Alpuente, M., Vidal, G. (eds.) SAS 2008. LNCS, vol. 5079, pp. 363–377. Springer, Heidelberg (2008)
27. Vafeiadis, V.: Automatically proving linearizability. In: Touili, T., Cook, B., Jackson, P. (eds.) CAV 2010. LNCS, vol. 6174, pp. 450–464. Springer, Heidelberg (2010)
28. Turon, A.: Reagents: expressing and composing fine-grained concurrency. In: PLDI, pp. 157–168 (2012)
29. Hawkins, P., Aiken, A., Fisher, K., Rinard, M.C., Sagiv, M.: Concurrent data representation synthesis. In: PLDI, pp. 417–428 (2012)