Using Program Closures to Make an Application Programming Interface (API) Implementation Thread Safe

Eitan Farchi Itai Segall
IBM Research Labs at Haifa
Israel
farchi@il.ibm.com
itais@il.ibm.com

João M. Lourenço Diogo Sousa
Departamento de Informática and CITI
Universidade Nova de Lisboa, Portugal
joao.lourenco@fct.unl.pt
dm.sousa@campus.fct.unl.pt

ABSTRACT

Consider a set of methods implementing an Application Programming Interface (API) of a given library or program module that is to be used in a multithreaded setting. If those methods were not originally designed to be thread safe, races and deadlocks are expected to happen. This work introduces the novel concept of program closure and describes how it can be applied in a methodology used to make the library or module implementation thread safe, by identifying the high level data races introduced by interleaving the parallel execution of methods from the API. High-level data races result from the misspecification of the scope of an atomic block, by wrongly splitting it into two or more atomic blocks sharing a data dependency.

Roughly speaking, the *closure of a program P, $clos(P)$*, is obtained by incrementally adding new threads to P in such a way that enables the identification of the potential high level data races that may result from running P in parallel with other programs.

Our model considers the methods implementing the API of a library of program module as concurrent programs and computes and analyses their closure in order to identify high level data races. These high level data races are inspected and removed to make the interface thread safe. We illustrate the application of this methodology with a simple use case.

Categories and Subject Descriptors

D.2.4 [**Software Engineering**]: Software/Program Verification—*Validation*; D.1.3 [**Programming Techniques**]: Concurrent Programming—*Parallel Programming*

General Terms

Languages, Reliability, Verification

Keywords

Atomicity Violation, High-Level Data Race, Debugging, Static Analysis, Concurrency

PADTAD'12, July 16, 2012, Minneapolis, MN, USA

1. INTRODUCTION

With the introduction of multicore platforms, concurrent programming is becoming mainstream. Libraries are thus commonly required to make their API (Application Programming Interface) thread safe. The implementation of the thread-safe version of an API is expected to be efficient and correct. Given an efficient thread-safe API implementation, the applications using the library can take advantage of the multicore setting to improve performance[1]. To make the library API thread safe, potential data races, deadlocks, and high level data races [1] need to be analysed. Intuitively, a high level data race is a non-atomic access to sets of variables, for which the relations between them actually call for atomic access. Note that the access to each set may be protected and atomic, but accesses to the different sets are not atomic.

An API is composed of a set of methods exposed to the API users. Each method may create new threads during its execution. As a result, concurrent anomalies, such as data races, high level data races and deadlocks, are possible regardless of the requirement for thread-safety between methods. Our analysis will address the detection of high-level data races in such settings, as well as those introduced by the parallel execution of the API's methods.

Adapting an existing implementation of an API to become thread safe involves three important stages: i) eliminate all data races caused by concurrent accesses to shared variables using a fine grained locking strategy, i.e. protect small blocks of code by locks (as opposed to rough protection of entire methods); ii) eliminate any deadlocks originated from the locking policy implemented in the first stage; and iii) eliminate the high-level data races existing in the implementation after stage ii) is concluded. Classical deadlock and data race identification tools can be applied to an existing sequential API implementation to help in the first two stages of the process of making the API thread safe. This work focuses on the third stage, i.e., on the analysis of high level data races in order to make the API thread safe.

To facilitate the discussion, we make some simplifying assumptions. In the conclusion we discuss how to remove them. The simplifying assumptions are: i) the data races are eliminated using a fine grain locking strategy over a single global lock; ii) we know *a priori* the set of shared variables; iii) we know *a priori* the order by which the atomic blocks (code blocks protected by a lock/unlock pair) are executed; iv) we know all subsets of shared variables accessed under

[1] Another problem that is addressed in [5] is the correctness of the application implementation.

the scope of a lock during the execution of one of the methods in the API; and v) all API methods have a fixed number of internal threads.

We are given a concurrent program P. The program is assumed to use a fine grained single lock strategy. The program region delimited by the lock/unlock pair is called an *atomic region*. Each program thread t_i in P is assumed to access a set of shared variables that may also be accessed by other threads. A *view* v of some atomic region of thread t_i is the set of shared variables accessed by t_i in that atomic region. A *maximal view* of some thread t_i is a view of t_i that is not contained in any other view of t_i. There is a *high level data race* between two threads t_i and t_j in P, if the sets resulting from the intersection of a maximal view v in t_i with the views of t_j, $V(t_j)$, do not form a chain. They form a chain if $\forall u, w \in V(t_j), v \cap u \subseteq v \cap w$ or *vice versa*. This concept of high level data races regarding the program's runtime was introduced in [1].

In this paper, we also take into account the control flow graph and dependencies between views when considering high level data races. In other words, we consider only views that may be executed one after the other in some execution of the program and that have shared variables in common. This increases the chance that a high level data race actually represents an intended transaction and subsequently a real concurrent anomaly, hence reducing the so called *false positives*. Other view dependencies could also be considered, such as an indirect data dependency between views through an auxiliary variable that does not appear in any of them. However, in this paper, we focus on the non-empty view intersection for simplicity, and discuss further options in Section 6.

An API is defined by a set of methods. We model a method as a concurrent program with a single point of entry. Given a set of methods $m_1, \ldots, m_n$ modeled as the set of concurrent programs $P_1, \ldots, P_n$, the high level data races that may result by executing any subset of methods concurrently are analysed to tackle the thread-safety challenge. We refer to the new program that may execute any subset of $P_1, \ldots, P_n$ concurrently as P. Instead of analysing the high level data races of P, the closure operation is introduced and applied to each P_i, thus obtaining a new program $clos(P_i)$. We then analyse the high level data races of $clos(P_i)$, in order to identify anomalies associated (in a manner that will be defined in the paper) with all of the high level data races of P.

For each of the programs P_i, the analysis of $clos(P_i)$ is local to P_i and also points to potential problems that are currently not occurring in P but may occur in future versions of P. For example, if P_i has the following views, $\{x, y\}$, $\{y, z\}$, belonging to the same thread and executing consecutively, but in P there is no other view with $\{x, y, z\}$, then no high level data races are reported in P concerning the consecutive access of $\{x, y\}$ and $\{y, z\}$ in P_i. Alas, in a future versions of P, the programmer might introduce a new thread with the view $\{x, y, z\}$, resulting in a high level data race. This potential future anomaly is highlighted by the high level data race analysis of $clos(P_i)$, but not by the high level data race analysis of P.

The rest of this paper is organised as follows. First, in Section 2, the concept of a program closure, $clos(P)$, for a concurrent program P is motivated and developed. In Section 3 we discuss how program closures can be used to solve the API thread safety problem. A detailed use case illustrating the process is covered in Section 4. Related works are discussed in Section 5, and conclusion and future work are covered in 6, including some considerations on how to remove the simplifying assumptions introduced in this Section.

2. PROGRAM CLOSURE

Consider the concepts of a *view* and a *maximal view* as defined in [1] and in the introduction above. Further consider a concurrent program P that has m threads $\{t_1, \ldots, t_m\}$[2]. The views associated with thread t_i $(1 \leq i \leq m)$ are denoted by $V_1^i, \ldots, V_{n_i}^i$, where $V_j^i, 1 \leq j \leq n_i$ is a view in thread t_i. We say that two views associated with t_i, V_j^i and V_k^i, are dependent if there is some execution of P for which V_k^i is executed after V_j^i, and $V_j^i \cap V_k^i \neq \emptyset$. For t_i we denote the directed graph obtained by that relation as D_i.

If the intersection of some maximal view in t_j, V_k^j, with the views of a maximal length path in D_i, $p = (V_1^i, \ldots, V_l^i)$, does not form a chain, then a high level data race exists between thread t_j and thread t_i. They form a chain if, $\forall u, w \in p, V_k^j \cap u \subseteq V_k^j \cap w$ or *vice versa*.

While the definitions discussed in this paper hold for maximal length paths in general, for practical usage the paths must be finite. Therefore we refer only to maximal length paths that do not include repeating views.

For each maximal length path $p = (V_1^i, \ldots, V_l^i)$ in D_i that does not include repeating views, we create a new thread t_p that is associated with a single view $C_p = V_1^i \cup V_2^i \cup \ldots \cup V_l^i$. C_p is also referred to as the closure set associated with the maximal path p and the thread t_i. Intuitively, the maximal path reflects a set of variable accesses that should be atomic, due to the dependencies between each pair of consecutive views.

Note that each thread we are adding to the program has only one view. As a result, the interaction of any two newly added threads will never add new high level data races to the program.

We define the closure of P, $clos(P)$, to be the program obtained from P by adding such threads, t_p, for each maximal length path p in D_i, for each thread i in P. The set of high level data races in P is contained in the set of high level data races in $clos(P)$. If the set of high level data races in P and $clos(P)$ are equal we say that P is a *closed program*.

To motivate the above definition a sequence of several examples is given.

2.1 Examples

In the following examples we use the keyword **atomic** and the associated block to denote the scope of the lock.

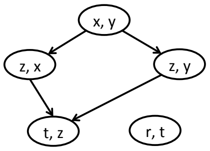
[2] For simplicity we assume that the number of program threads is fixed and known.

2.1.1 Example 1

T_1

```
atomic { x = y; }
if (cond_1)
    atomic { z = x; }
else
    atomic { z = y; }
if (cond_2)
    atomic { t = z; }
else
    atomic { r = t; }
```

Dependency graph for T_1



$C_2 = \{x, y, z, t\}$
$C_3 = \{x, y, z, t\}$

Consider a program P_1 with **a single thread** executing the code T_1 above. The closure sets associated with this thread are $C_2 = C_3 = \{x, y, z, t\}$. So, to close the program we must add a new thread T_2 accessing the variables in C_2.

T_1	T_2
$V_1^1 = \{x, y\}$	$C_2 = C_3 = \{x, y, z, t\}$
$V_2^1 = \{z, x\}$	
$V_3^1 = \{t, z\}$	
$V_4^1 = \{z, y\}$	
$V_5^1 = \{r, t\}$	

Note that according to our definition of dependency between views, as the intersection of $\{r, t\}$ with $\{z, x\}$ and with $\{z, y\}$ is empty, then there is no dependency between $\{z, x\}$ and $\{r, t\}$, neither between $\{z, y\}$ and $\{r, t\}$, although they have a control flow relation.

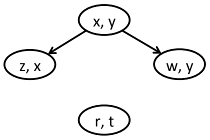
This example shows how the program closure captures potential high level data races that do not currently exist, but that may exist if further threads are to be introduced to the system. It also illustrates that multiple equal closure sets will be associates with a single new thread in the closed program.

2.1.2 Example 2

T_1

```
atomic { x = y; }
if (cond_1)
    atomic { z = x; }
else
    atomic { w = y; }
atomic { r = t; }
```

Dependency graph for T_1



$C_2 = \{x, y, z\}$
$C_3 = \{x, y, w\}$

Consider a program P_2 with **a single thread** executing the code T_1 above. $clos(T_1)$ has two new threads T_2 and T_3, with views C_2 and C_3 respectively.

T_1	T_2	T_3
$V_1^1 = \{x, y\}$	$C_2 = \{x, y, z\}$	$C_3 = \{x, y, w\}$
$V_2^1 = \{z, x\}$		
$V_3^1 = \{w, y\}$		
$V_4^1 = \{r, t\}$		

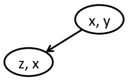
The new program $clos(T_1)$ has now two high level data races: one is obtained from $V_1^1 + V_2^1$ and C_2, and the other is obtained from $V_1^1 + V_3^1$ and C_3. Each newly added thread in the program closure represents a set of threads that in the future may be introduced in the system, hence potentially triggering the identified high level data race.

2.1.3 Example 3

T_1

```
atomic { x = y; }
if (cond_1)
    atomic { z = x; }
```

Dependency graph for T_1



$C_2 = \{x, y, z\}$

Consider a program P_3 with multiple threads executing the code above. This program has no high level data races. As all the threads in P_3 execute the same code, they all share the same views $V_1^1 = \{x, y\}$ and $V_2^1 = \{z, x\}$, and have the same maximal path in their control flow (V_1^1, V_2^1), so to close the program we will add only one single thread with the view $C_2 = \{x, y, z\}$. The new program $clos(P_3)$ has a high level data race, as $V_1^1 \cap C_2 = V_1^1$ and $V_2^1 \cap C_2 = V_2^1$ and neither $V_1^1 \subseteq V_2^1$ nor $V_2^1 \subseteq V_1^1$, i.e., V_1^1 and V_2^1 do not form a chain.

2.1.4 Example 4

T_1

```
atomic { x = y; }
atomic { z = x; }
```

T_2

```
atomic { x = y + z; }
```

Consider a program P_4 with two threads T_1 and T_2. In this example, $clos(P_4)$ would include one extra thread with the view $C_3 = \{x, y, z\}$, which is already in P_4 (in T_2). Hence P_4 is already closed and P_4 and $clos(P_4)$ have the same (possibly empty) set of high level data races.

2.1.5 Example 5

T_1

```
atomic { x = y; }
if (cond_1)
    atomic { z = x; }
else
    atomic { z = y; }
if cond_2)
    atomic { t = z; }
else
    atomic { r = t; }
```

T_2

```
atomic {
    x = y + z + t;
}
```

T_3

```
atomic {
    x = y + z + t;
}
```

Consider the program P_5 above obtained from the example in Section 2.1.1 by the closure process, i.e., $P_5 = clos(P_1)$. All the closure sets introduced in $clos(P_5)$ would be equal to an already existing view in P_5, hence no new views nor threads are introduced for $clos(P_5)$. This is an example of Lemma 2.1 below ("*$clos(P)$ is always closed*").

2.2 Properties of Program Closure

LEMMA 2.1. *For any program P, $clos(clos(P)) = clos(P)$.*

COROLLARY 2.2. *The closure of a program P is always closed.*

PROOF. Clearly, $clos(P) \subseteq clos(clos(P))$. Every thread in $clos(clos(P))$ that was not in P is either a result of applying closure to a thread in P or to a thread in $clos(P) - P$. When we apply the closure to a thread in P we clearly remain in $clos(P)$. On the other hand, when we apply the closure to a thread t in $clos(P) - P$ it does not result in a new thread as threads in $clos(P) - P$ have a single view. □

3. MAKING AN API IMPLEMENTATION THREAD SAFE

Given an implementation of an Application Programming Interface (API), we would like to make it thread safe, i.e., let each method in the API execute in parallel to the others correctly. We expect that the API methods will need to be modified, and appropriate protection added, in order for them to execute correctly in parallel. In this section we demonstrate how the closure of a concurrent program can aid in identifying the modifications required in order to make the API thread safe.

We consider each API's method call to be a spawn of a new thread that will execute this method. Note, however, that an API's method can create more threads as it executes, thus it does not necessarily have just one thread.

Let V be the set of views belonging to any of the API methods. Each API method can therefore be viewed as a concurrent program P_i that is a set of set of views, hence $P_i \in \mathcal{P}(\mathcal{P}(V))$. As a result, for our purpose, the set of API methods is a set of concurrent programs $API = \{P_1, \ldots, P_k\}$, and executing them in parallel creates a new concurrent program $PAPI = P_1 \cup \ldots \cup P_k$. For example, if

$$P_1 = \{\{\{x,y\},\{y,z\}\},\{\{x\},\{x,y\},\{y\}\}\}$$

has two threads, and

$$P_2 = \{\{\{x\},\{y,z\}\},\{\{x\},\{y\}\}\}$$

has two threads as well, then

$$\begin{aligned} PAPI &= \{\{\{x,y\},\{y,z\}\},\{\{x\},\{x,y\},\{y\}\}\} \\ &\quad \cup \{\{\{x\},\{y,z\}\},\{\{x\},\{y\}\}\} \\ &= \{\{\{x,y\},\{y,z\}\},\{\{x\},\{x,y\},\{y\}\}, \\ &\quad \{\{x\},\{y,z\}\},\{\{x\},\{y\}\}\} \end{aligned}$$

and has four threads.

The set of high level data races that arise from the execution of $PAPI$ is denoted by $HLDR(PAPI)$. We are interested in analysing the relation between this set and the sets of high level data races of the closure of the API methods, i.e., the sets $HLDR(CLOS(P_i)), i = 1, \ldots, k$. We will show that focusing on the high level data races of the closure of each API method, $HLDR(CLOS(P_i)), i = 1, \ldots, k$, is sufficient for the purpose of making $PAPI$ a thread safe program.

Lemma 3.1. *Any high level data race in $PAPI$ between threads t_i and t_j is either a false alarm, or there is a corresponding high level data race in $HLDR(clos(P_i))$ or in $HLDR(clos(P_j))$.*

Proof. Consider a high level data race in $PAPI$ between threads t_i and t_j. Assume w.l.o.g. that the high level data race is between a maximal view V^j of thread t_j and a set of views $V_1^i, \ldots, V_r^i$ of thread t_i. By definition, $\{V_1^i \cap V^j, \ldots, V_r^i \cap V^j\}$ do not form a chain, i.e., there exists a, b s.t. $V_a^i \cap V^j \nsubseteq V_b^i \cap V^j$ and $V_b^i \cap V^j \nsubseteq V_a^i \cap V^j$, which implies that $V_a^i \nsubseteq V_b^i$ and $V_b^i \nsubseteq V_a^i$.

Consider views V_a^i, V_b^i as above. If V_a^i and V_b^i are contained in a maximal path in D_i, then by construction $clos(P_i)$ contains a maximal view V' s.t. $V_a^i \cup V_b^i \subseteq V'$, therefore $V_a^i \cap V' \nsubseteq V_b^i \cap V'$ and $V_b^i \cap V' \nsubseteq V_a^i \cap V'$, therefore a corresponding high level data race exists in $HLDR(clos(P_i))$.

Otherwise, V_a^i and V_b^i are not contained in a maximal path in D_i. This can be either because no control flow exists between them (i.e., there is no run in which V_a^i runs after V_b^i or vice versa), or because they are disjoint. In either case, we consider the high level data race a false alarm. □

To illustrate the case when V_a^i and V_b^i are not contained in a maximal path in D_i, consider the following example:

T_0

```
If(cond1)
atomic { x = y; }
else
atomic { z = x; }
```

T_1

```
atomic { x = y + z; }
```

Intersecting the maximal view of T_1, $\{x,y,z\}$, with the views of T_0, $\{\{x,y\},\{z,x\}\}$ results in a non chain $\{\{x,y\}, \{z,x\}\}$ and thus in a high level data race. But this is a false alarm, as such high level data race can never actually occur at runtime as only $\{x,y\}$ or $\{z,x\}$ may occur in a run of T_0, but never both.

Note that Lemma 3.1 makes no assumptions about the threads being different from each other. Therefore the Lemma covers also the case where the same API's method is called twice, i.e., when it runs in parallel to itself.

4. A DETAILED USE CASE

In this section we introduce a step-by-step description of the proposed methodology to make an API implementation thread-safe. To clearly highlight the approach, the example is intentionally kept simple, and we will assume the original implementation of the API was single-threaded.

As described before, for the first two steps the programmer applies standard race detection and deadlocks techniques to synchronize the API methods. The added synchronization can be fine grained and may lead to high level data races. Next, the third step, applying the closure analysis techniques proposed in this paper, is applied to find the potential high level data races introduced in the first steps.

4.1 Making the Account API Thread Safe

This example, inspired by the example in [4], describes a possible implementation of a bank account API. At any given time, the account state includes how much the customer has in USD and EUR, and a transaction serial number. The API methods are getBalanceUSD(), getBalanceEUR(), setBalanceUSD(), setBalanceEUR() and transferUSDtoEUR(), implemented as depicted in Figure 1.

In the first step of the proposed methodology to make an API implementation thread-safe, a trivial adaptation of the API, is to enforce a single global lock on program statements accessing the shared variables, namely balanceUSD, balanceEUR, and serialNumber, as depicted in Figure 2. In a more complex program this adaptation could be hinted by race detection tools. As we are using fine grained locking on a single lock, no deadlocks are introduced by this transformation. Again, in a more complex program, the detection of deadlocks could be hinted by deadlock detection tools. Hence, after applying the first two steps, the new version of the API is both deadlock and data race free.

Next, in the third step, the closure analysis is used to alert the programmer of existing or possible high level data races. Each API method, m_i, is modeled as a concurrent program P_i. The analysis of the closure of P_i, $clos(P_i)$ is used to identify the potential high level data races in the methods

```
public class Account {

  protected int balanceUSD;
  protected int balanceEUR;
  protected int serialNumber;

  public Account(){
    this.balanceUSD = 0;
    this.balanceEUR = 0;
    this.serialNumber = 0;
  }

  // operation on USD balance
  public int getBalanceUSD() {
    serialNumber++;
    return balanceUSD;
  }
  public int setBalanceUSD(int valUSD) {
    serialNumber++;
    balanceUSD = valUSD;
  }

  // operation on EUR balance
  public int getBalanceEUR() {
    serialNumber++;
    return balanceEUR;
  }
  public int setBalanceEUR(int valEUR) {
    serialNumber++;
    balanceEUR = valEUR;
  }

  // transfer USD to EUR
  public void transferUSDtoEUR(int valUSD) {
    // withdraw the money
    int tmp = getBalanceUSD();
    tmp = tmp - valUSD;
    setBalanceUSD(tmp);
    // convert currency
    int valEUR = cambioUSDtoEUR(valUSD);
    // deposit in the new currency
    tmp = getBalanceEUR();
    tmp = tmp + valEUR;
    setBalanceEUR(valEUR);
    // until now we executed 4 sub-transactions
    // transferUSDtoEUR() is only one trsaction
    // so we need to subtract 3 from
    // serialNumber
    serialNumber = serialNumber - 3;
  }
}
```

Figure 1: The unprotected implementation of the Account API

```
public class Account {

  // No need to protect the constructor
  public Account() { ... }

  public int getBalanceUSD() {
    synchronize (this) {
      serialNumber++;
      return balanceUSD;
    }
  }
  public void setBalanceUSD(int valUSD) {
    synchronize (this) {
      serialNumber++;
      balanceUSD = valUSD;
    }
  }

  public int getBalanceEUR() {
    synchronize (this) {
      serialNumber++;
      return balanceEUR;
    }
  }
  public void setBalanceEUR(int valEUR) {
    synchronize (this) {
      serialNumber++;
      balanceEUR = valEUR;
    }
  }

  // "transferUSDtoEUR" calls getBalance*()
  // and setBalance*(), and both are already
  // protected, so no need to protect again.
  // However, we need to proteect serialNumber
  public void transferUSDtoEUR (int valUSD) {
    ...
    // until now we executed 4 sub-transactions
    // transferUSDtoEUR() is only one trsaction
    // so we need to subtract 3 from
    // serialNumber
    synchronize (this) {
      serialNumber = serialNumber - 3;
    }
  }
}
```

Figure 2: The fine grain locking-based protected implementation of the AccountTest API

m_i of the API.

For this API we shall consider five programs P_1 to P_5, corresponding to methods getBalanceUSD(), setBalanceUSD(), getBalanceEUR(), setBalanceEUR(), and transferUSDtoEUR() respectively. Table 4.1 summarises the views for each program P_i (corresponding to each method m_i) of this API.

Each program P_k, $(1 \leq k \leq 4)$ either reads or writes one of the balance fields (for USD and EUR currency), and updates the transaction serial number. Hence, each contains a single view with its corresponding balance field and serial number. As P_5 resorts to the usage of the getBalanceUSD/EUR() and setBalanceUSD/EUR() methods, it has two views, each containing one of the balance fields and the serial number. P_5 has a third view resulting from the access to serialNumber in an atomic region.

Each program $P_k(1 \leq k \leq 4)$ has a single maximal path that includes their corresponding unique view. P_5 also has a single maximal path that include two views $\{V_1^5, V_2^5, V_3^5\}$. As the intersection of these views is not empty, the closure of P_5 will contain an additional thread with a single view $C_4^5 = V_1^5 \cup V_2^5 \cup V_3^5 = \{balanceUSD, balanceEUR, serialNumber\}$ that includes all the variables accessed in the maximal path.

We now proceed by detecting the high level data races in each $clos(P_i)$ $(1 \leq i \leq 5)$, separately. As no views were added to $clos(P_k)$ w.r.t. P_k $(1 \leq k \leq 4)$, we have that $clos(P_k) = P_k$ contains a single view, hence the parallel execution of the m_k methods cannot generate high level data races.

Next, we analyze $clos(P_5)$ for high level data races. The set of views in $clos(P_5)$ is $V^5 = \{V_1^5, V_2^5, V_3^5, C_4^5\}$, where C_4^5 is a *maximal view* of one of the threads. Calculating, we get $I_1^5 = V_1^5 \cap C_4^5 = \{balanceUSD, serialNumber\}$, $I_2^5 = V_2^5 \cap C_4^5 = \{balanceEUR, serialNumber\}$, and $I_3^5 = V_3^5 \cap C_4^5 = \{serialNumber\}$. As $I_3^5 \subseteq I_1^5$ and $I_3^5 \subseteq I_2^5$, we know I_3^5 will not be part of a high level data race. However, as neither $I_1^5 \nsubseteq I_2^5$ nor $I_2^5 \nsubseteq I_1^5$, we may still conclude that the execution of method transferUSDtoEUR() in a multithreaded environment is not thread-safe and may lead to high level data races.

The programmer may fix this anomaly by having two versions of each get and set balance methods, one that acquires the lock and another that doesn't. The transfer method can then use the versions of methods that do not acquires the lock and introduce a lock that covers the entire scope of the transfer operation.

5. RELATED WORK

To our best knowledge, this work is the first to address specifically the detection of high-level data races in sequential libraries transformed to be used in a multithreading setup. However, several past works addressed the detection of high-level data races in concurrent programs.

There are many previous works addressing the problem of detection of atomicity violations, using static or dynamic program analysis technique [2, 3, 4, 6]. Some even propose strategies to mask or even eliminate the detected atomicity violations, but this objective is out of the scope of this paper. In this section we will only address some works that are closely related to our approach.

Artho et al. [1] introduces the concepts of *view* of an atomic block, that is a set containing all the shared variables accessed within that block, the *maximal view* of a thread, which are those views that are not a subset of any other view of that thread, and the concept of *view consistency*, which when violated indicates a potential high-level data races. Our work builds on these concepts, but introduces some additional concepts, like the dependency relation between views and the closure of programs, and addresses the specific problem of high-level data races in APIs that are to become thread safe.

Shacham et al. [5] propose a methodology for testing atomicity of composed concurrent operations. They propose a technique that is based on modular testing of client code in the presence of an adversarial environment, and use commutativity specifications to reduce the number of executions explored to detect a bug. Our approach is able to analyse a program module without the client code, and reduces drastically the space of states to be checked by applying the new concept of program closure and analysing each method of the API separately.

6. CONCLUSIONS

This paper addresses the problem of making an API implementation thread safe. Data races and deadlocks can be handled using standard tools, and in this paper we tackle the problem of analyzing and avoiding potential high level data races. We propose the new concept of *program closure*, $clos(P)$, and demonstrate that it is sufficient to separately analyse the closure of each API method in order to identify the high level data races of the API implementation. Using this approach, we avoid the analysis of the very large space of combinations of API methods. Following this suggested methodology, the high level data race anlaysis has to be applied to n concurrent programs, $clos(P_i)$. On the other hand, if the high level data race would have been applied to the API as a whole, it would have been applied to n choose 2 pairs of API methods occurring concurrently. The approach is further elucidated with a simple use case.

Several simplifying assumptions were made, which can be mitigated or even eliminated. The closure analysis requires the identification of the set of concurrent threads (Introduction assumption v), the set of shared variables that are accessed in atomic regions (Introduction assumption ii), the order by which the atomic regions are executed within a thread (Introduction assumption iii), and the variables in each view (Introduction assumption iv). All this knowledge can be approximated using static analysis. Alternatively, the concurrent program execution can be monitored to identify the threads of a given run, and the shared variables that were used under the scope of locks. This is the approach taken in [1]. If the runtime approach is taken, then a test suite may be developed for the API and the high level data races analysis is applied to each test in the test suite.

The limitation of using a single lock (Introduction assumption i) is a technical limitation aimed at simplification of the presentation. It can be removed by analyzing the dependencies as discussed in the paper for each of the program's locks.

We have developed a static analysis tool that implements the closure concept, assuming a pre-determined number of threads in the concurrent program and a single thread per API method. There is ongoing work to enhance this implementation and test it against larger real life programs.

The closure operation has merit in itself. It is probably useful for program maintenance in general, as it identifies potential high level data races that may be introduced in future releases of a concurrent program, pointing out to the devel-

Table 1: The Views for the Account example.

Program	Method	Views
P_1	getBalanceUSD()	$V_1^1 = \{balanceUSD, serialNumber\}$
P_2	setBalanceUSD()	$V_1^2 = \{balanceUSD, serialNumber\}$
P_3	getBalanceEUR()	$V_1^3 = \{balanceEUR, serialNumber\}$
P_4	setBalanceEUR()	$V_1^4 = \{balanceEUR, serialNumber\}$
P_5	transferUSDtoEUR()	$V_1^5 = \{balanceUSD, serialNumber\}$, $V_2^5 = \{balanceEUR, serialNumber\}$, $V_3^5 = \{serialNumber\}$
$clos(P_5)$	transferUSDtoEUR()	$V_1^5 = \{\ldots\}$, $V_2^5 = \{\ldots\}$, $V_3^5 = \{\ldots\}$, $C_4^5 = \{balanceUSD, balanceEUR, serialNumber\}$

oper that they may revise their code before deploying a new version. Further work is needed to determine the effectiveness of the approach in this general program maintenance setting. This approach of relating the closure operation to program maintenance operations can take interesting developments. Can a programmer get online indications that her latest changes in a program module did not introduce any high-level data races, but it introduced the potential for high-level data races if the module API is used in accordance to a specific criteria? The program closure might be able to support such an IDE (Integrated Development Environment) feature.

The dependency graph, D_i, can be defined in other ways. Instead of the intersection between two views, one might think of other criteria, e.g., a data dependency between the two views through an auxiliary variable. A unifying theory of high level data races that captures the different possible dependencies and high lights the trade-off between them is called for.

7. ACKNOWLEDGEMENTS

This research was partially funded by the European Community's Seventh Framework Programme [FP7/2007-2013] under grant agreement 257574 (FITTEST) and the Euro-TM EU COST Action IC1001, and by the Portuguese National Science Foundation in the research projects RepComp (PTDC/EIA-EIA/108963/2008), Synergy-VM (PTDC/EIA-EIA/113613/2009), and the research grant SFRH/BD/41765/2007.

8. REFERENCES

[1] Artho, C., Havelund, K., Biere, A.: High-level data races. Software Testing, Verification and Reliability 13(4), 207–227 (Dec 2003)

[2] Lourenço, J., Sousa, D., Teixeira, B.C., Dias, R.J.: Detecting concurrency anomalies in transactional memory programs. Comput. Sci. Inf. Syst. 8(2), 533–548 (2011)

[3] Pessanha, V., Dias, R.J., Lourenço, J.M., Farchi, E., Sousa, D.: Practical verification of high-level dataraces in transactional memory programs. In: Proceedings of the Workshop on Parallel and Distributed Systems: Testing, Analysis, and Debugging. pp. 26–34. PADTAD'11, ACM, New York, NY, USA (2011), http://doi.acm.org/10.1145/2002962.2002968

[4] von Praun, C., Gross, T.R.: Static detection of atomicity violations in object-oriented programs. Journal of Object Technology 3(6), 103–122 (Jun 2004), http://www.jot.fm/contents/issue_2004_06/article5.html, workshop on Formal Techniques for Java-like Programs (FTfJP), ECOOP 2003

[5] Shacham, O., Bronson, N., Aiken, A., Sagiv, M., Vechev, M., Yahav, E.: Testing atomicity of composed concurrent operations. In: Proceedings of the 2011 ACM international conference on Object oriented programming systems languages and applications. pp. 51–64. OOPSLA '11, ACM, New York, NY, USA (2011), http://doi.acm.org/10.1145/2048066.2048073

[6] Wang, L., Stoller, S.: Run-Time Analysis for Atomicity. Electronic Notes in Theoretical Computer Science 89(2), 191–209 (Oct 2003)