Correctness of Request Executions in Online Updates of Concurrent Programs

Technical Report

Yogesh Murarka  Umesh Bellur
Department of Computer Science and Engineering
Indian Institute of Technology Bombay, Mumbai, India
{yogeshm, umesh}@cse.iitb.ac.in

June, 2008

Abstract

Online update is a technique which reduces the disruption caused during software update by applying a patch to a running process. It is a challenge to update a process while ensuring that it continues to operate correctly during and after the update. Most of the continuously running processes concurrently execute the requests arriving in a stream. Online update should guarantee a correct outcome of the requests which execute during and after an update. In this report we provide a solution for the same.

We first define a request execution criteria to ensure the correct outcome of the requests which execute concurrently with an update. The report then describes an online update approach that fulfills this criteria. The approach avoids deadlocks during update by analyzing interthread dependencies and guarantees that the process remains in a consistent state after the update. Thus, the update procedure is guaranteed to terminate and the requests which execute during and after an update are ensured the correct execution. Our literature survey reveals that this is the first solution to update concurrent programs while requests are executing and ensure correctness.

1. Introduction

Correctness and continuity are the two major concerns while updating a system. Correctness means an update should not make a system behave in an erroneous manner, while continuity means a system should continue to provide the service during an update with minimum disruption. A common approach to ensure correctness is to shut down the system. Although, this guarantees the correctness it causes service downtime. An update can be scheduled in advance, however the disruption caused remains undesirable for high availability applications. An alternate approach to apply a patch is the online update where a running process is updated [13, 14, 15]. Guaranteeing the correctness of online update is more complex than that of offline update because online update has to ensure the correct outcome of ongoing executions. This report presents an online update solution for the concurrent object oriented programs.

Most of the high availability applications concurrently process continuously arriving requests. Existing solutions for online update either abort some of the executing requests and restart them or hold the new requests till in-process requests are completed and the process is updated. The first approach cannot be used for applications where aborting an ongoing execution is unacceptable. In
the second approach, the application will be virtually unavailable till executing requests are completed. The advantage of such an update will be lost if some requests take a long time to finish. This report presents a solution to update a process while requests are in the middle of execution without aborting them.

Updating a process while a request is executing can produce an incorrect outcome. Because a part of the request might get executed on the old program and the rest of it on the updated program. The outcome of the request will neither be as per the old program nor as per the updated program. For example, in a code fragment illustrated in Figure 1(a), a request is processed using an Order-Handler thread which spawns an Allocator thread and prepares a response. During the response preparation a bill is generated for the order. Allocator thread computes the estimation for the delivery time, etc. In the patch illustrated in Figure 1(b) the request execution is modified to generate the bill during the estimation than the response generation. Updating the process before OrderHandler thread has prepared the response and after Allocator thread has generated the estimation will not bill the order.

```
void OrderHandler::run(Request req, Response res)
  
Order ord = req.getOrder();
Thread t = new Allocator(ord); t.start();
ord.genResponse();
...
void Allocator::run(Order ord){
  if(ord.getAvailability())
    ord.getEstimation();
  ...
  Order::genResponse(){genBill(); ...}
  Order::getEstimation(){deliveryTime();... }

  (a) Old Program

  Order::genResponse(){...}
  Order::getEstimation(){.. genBill(){..}

  (b) Patch

  Figure 1: An Example Scenario
```

Our earlier solution for online update of concurrent programs [24] ensures a type safe update and the solution can be configured to ensure that a request is executed entirely either using the old class definitions or using the updated class definitions. However, this is not sufficient to ensure correctness. The update may leave the process in an inconsistent state and may induce a deadlock. During the update some request might be executing on the old program and some requests might be executing on the updated program. A shared object might go in inconsistent state when two requests executing on the different programs simultaneously modify it. To ensure correctness concurrently running requests need to be synchronized during the update. Interthread synchronization in the program along with update synchronization can lead to a deadlock.

The solution presented in this report guarantees correct outcome of requests during and after the update and ensures that the update does not lead to a deadlock. The correct outcome is guaranteed by enforcing that: (a) a request is executed completely either on the old program or on the updated program, and (b) the process state accessed by the request is a valid state as per the corresponding program. Deadlock is prevented by analyzing interthread dependencies and by synchronizing the process threads accordingly.

The report is organized as follows. In the next section we specify a request execution criteria to ensure the correct outcome of requests during and after the update. In Section 3 we present an online update procedure to enforce this execution criteria. This is achieved by scheduling the update
at specific points in the process execution. In Section 4 we derive a safety condition to be satisfied by an update schedule to ensure the request execution criteria and to avoid deadlock. Section 5 presents a procedure to compute an update schedule that satisfies the safety condition. The schedule is computed by analyzing the old program and the patch. Section 6 discuss the related work and Section 7 concludes.

2. Semantics of Online Update

This section describes our update model and our criteria for an online update.

2.1. Process Model

Many applications requiring high availability, such as web based enterprise applications, have to process continuously arriving requests. The thread in the process accepting these requests executes an infinite loop. We consider a simple process model where within an iteration of the loop a request is accepted, a new thread gets instantiated, and the request is delegated to it. We assume that the thread running the main method in the program is an infinitely running thread which keeps accepting the requests. Threads which are spawned by the main thread to process the requests, terminate in finite time.

We assume an object oriented programming model since object oriented languages are dominant in enterprise applications which are our target. We assume that interthread communication is performed using shared objects. We present the concurrency model in detail in Section 4.

2.2. Patch Model

Considering a class as a unit of change, a patch can modify existing classes, add new classes, and remove existing classes from a program. The program before applying a patch is termed old program and the program which would result after the patching is termed new program. While updating the program, a patch may also update the data used by the program. Data has to be updated so that the new program can work correctly with it. For an offline update, data consist of persistent state. In the case of an online update, data also include the state of the process. For an object oriented program a process state contains set of objects which refer to each other. To have a unified state representation, we model the persistent state as objects in the process.

We classify changes in a program definition as follows.

- **Deleted class**: A class in the old program which does not exist in the new program.
- **Added class**: A class in the new program which did not exist in the old program.
- **Replaced class**: A class in the old program which is modified in the new program.
- **Replacing class**: A class in the new program which is modified version of a class in the old program.
- **Unmodified class**: A class which exists in both the old program and the new program without any change.

A class is recognized by its name. If the name of a class is changed then the class with the original name is considered as a deleted class and the class with the new name is considered as an
added class. A class in the old program is considered as modified in the new program if the new program contains a class with the same name but with a different set of members or different method implementation. We call a deleted or replaced class as an old class and an added or replacing class as a new class.

Modifying a class definition might not result in having to update the state of its objects. For example, a replacing class can be an extension of a replaced class with some new methods. In such case, objects of the old class might be used by the new program threads without having to update their state. On the contrary, an object state may need to be updated even though the class definition remains same. For example, while updating the names from ASCII to Unicode the class remains same as String. Classes are classified according to compatibility of their objects’ state as follows.

- **Backward state compatible class**: Let, \( o_b \) be an object of class \( B \) created and/or modified by a thread executing an old program. Class \( B \) is backward state compatible if a thread executing the new program can use \( o_b \) without having to update its state.

- **Forward state compatible class**: Let, \( o_f \) be an object of class \( F \) created and/or modified by a thread executing a new program. Class \( F \) is forward state compatible if a thread executing the old program can use \( o_f \) as it is. For example, storage objects containing the data encrypted by the new program may or may not be used by the old program as it is.

In the rest of the report we use the term **compatibility** to mean state compatibility. States of non backward compatible class objects are updated using State Transfer Functions (STF) [6, 15] so that the new program can use them. We briefly explain the STF in the following.

**State Transfer Function**

A common approach to specify the state update is to define a State Transfer Function (STF) [6, 15, 22] for each non backward compatible class. Since an STF has to accesses the state of an object, STF for a class is defined as an additional method in the class. Two example STFs are presented in the Figure 2. Example in Figure 2.(a) defines an STF for the class \( B \) which is modified in the patch.

In the example the new class is referred as \( B' \) and the old class is referred as \( B \). STF instantiates a new object of the modified class, initialize it from the state of old object, and return the new object. During update the object returned by the STF takes over the identity of the object on which the STF was invoked. Example in Figure 2.(b) defines an STF for the class \( C \) though the class is not modified in the patch. The STF updates the name from ASCII to Unicode.

```
B' B::STF(){
    B' bnew = new B'();
    bnew.set(this.a);
    return bnew;
}
void B'::set(int a){
    this.a = (float) a;
}

void C::STF(){
    this.name = toUni(this.name);
    String C::toUni(String n){
        String uni = new String();
        ...
        return(uni);
    }
```

(a) (b)

Figure 2: Example STFs

STFs are executed by the update procedure. An STF is applied on an object after all requests executing the old program have completed their executions on the object, and before any request executing on the new program has accessed it.
Because a deleted class is not backward compatible, an STF should be defined to make its objects compatible with the new program. A patch for online update contains a list of non forward compatible classes, definitions of the new classes, and STFs for non backward compatible classes.

For simplicity, we restrict our attention to modification of classes that are not used by the *main thread*. The aim is to explain how the classes used by request processing threads are updated.

### 2.3. Criteria for the Online Update

As discussed in the introduction we ensure that a request is executed entirely either on the old program or on the new program. We term a request executed on the old program as an *old request* and a request executed on the new program as a *new request*. We define the execution criteria for them in the following.

**Request Execution Criteria:** Let, $u$ be an update and $r_i$ be an application request which executes concurrently with $u$. The execution of $r_i$ should satisfy at least one of the following conditions.

**OPE Old Program Execution:** Request $r_i$ has been executed using the old program. $r_i$ has not accessed a non backward compatible class object after $u$ has updated it. $r_i$ has not accessed a non forward compatible class object after a new request has accessed it.

**NPE New Program Execution:** Request $r_i$ is executed using the new program. $r_i$ has not accessed a non backward compatible class object before $u$ has updated it.

Next section explains our procedure to apply a patch to a process.

### 3. Applying the Online Patch

A patch may or may not affect execution of all threads in a process. We call a thread as *affected thread* if interleaving of the update with it can affect the correctness of the update. Points in the execution of an affected thread where an update can be safely interleaved are called the *update points*. Every affected thread has a region of contiguous points which are unsafe for the update. Such a region is called as *unsafe region*. Figure 3 conceptually illustrates the unsafe region. The rational for existence of a single unsafe region in each affected thread is given in the next section. All the points before and after the unsafe region are the update points. To prevent a thread from being in the unsafe region, during the update, we insert the *update calls* at the start and the exit of the unsafe region.

An update call can be either blocking or non-blocking. Non-blocking update calls simply notify that a process thread has crossed an update point. Non-blocking update calls are used to postpone the update till a thread exits the unsafe region. Blocking update calls are used to block a thread from entering its unsafe region till the update is complete. Set of update points of all affected threads where update calls are inserted, is called the Process Update Cut (PUC). The threads which are
blocked before the unsafe region will execute as the new program threads and the threads which have crossed the unsafe region before the update will execute as the old program threads.

Figure 4: Update Procedure

Figure 4 illustrates important steps in our update procedure. On receiving a patch, the process is suspended (event e1) and a PUC is selected depending on the values of instruction pointers (ips) of all threads in the process. The Update calls are inserted at the PUC and the process is resumed (event e2). After all the non blocking update calls are executed, the patching starts (event e3). After the process is patched, the update calls are removed, threads waiting on the blocking update calls are notified, and the update procedure ends (event e4). We refer event e3 as the start of the update and event e4 as the end of the update. After an update, some threads in the process might still be executing old requests using unmodified classes. The process will start behaving exactly like an updated process after all threads executing the old requests terminate (event e5).

While reaching non blocking update calls, i.e. between event e2 and e3, new instances of the affected threads can be started in the process. Update calls are also inserted in such threads.

In the rest of this report we restrict our discussion to ensuring correctness, i.e. computing a PUC in our update procedure. Techniques to patch a process have been developed by many researchers [9, 11, 13, 18, 21]. Next section presents the conditions to be satisfied by the update points.

4. Ensuring the Correctness

We begin this section with a concurrency model and an interprocedural flow graph representation of the computation defined by a program. An update point is an execution point in the interprocedural flow graph of a thread. We then define the condition over execution points that can be selected as update points.

4.1. Program Model

We use the term thread to refer to a process thread and the term thread class to refer to a class that is marked runnable, by virtue of which a thread is associated with every object of it. A main()
method is present in every program and a start() method is present in every thread class. Execution of a program starts with the execution of the main() method in a main thread. Main thread and other live threads in a process start the new threads by invoking start() method on them.

Threads communicate with each other through shared objects. A locking mechanism in order to provide exclusive access to a shared object is implemented through synchronized blocks. Every object has a lock associated with it. The lock is acquired when a thread enters a synchronized block of the object, and it is released on the exit of the thread from the block.

Threads can interrupt their execution and become inactive by calling a wait() method on an object inside the synchronized block. The thread releases the lock over the object before becoming inactive. Another thread wakes up such inactive thread by invoking notify() method from a synchronized block over the same object.

**Flow Graph**

Computation defined by a method is represented by a flow graph. Flow graph $G_m$ of a method $m$ is represented as triple $(N_m, E_m, s_m)$. Where, $N_m$ is a set of basic blocks in method $m$. A method invocation statement is represented by two basic blocks, a call block and a return block. Entrance and exit points of a synchronized block are represented by blocks $lock(o)$ and $unlock(o)$, where $o$ is the lock object of the synchronized block. $E_m = E^0_m \cup E^1_m$ is a set of edges. An edge $e(n_1, n_2) \in E^0_m$ iff there is direct transfer of control from block $n_1$ to $n_2$, and an edge $e(n_1, n_2) \in E^1_m$ iff $n_1$ is a call block and $n_2$ is the corresponding return block. Every method $m$ has a unique entry block $s_m$ and a unique exit block $e_m$.

**Interprocedural Flow Graph**

We use the Interprocedural Flow Graph (IFG) representation in [28] to represent computation defined by a program. IFG of a sequential program is defined as $(N^*, E^*, s_{main})$, where $N^* = \bigcup_m N_m$, and $E^* = E^0 \cup E^1$. Where, $E^0 = \bigcup_m E^0_m$, and an edge $(n_1, n_2) \in E^1$ iff (a) $n_1$ is a call block of a method invocation statement and $n_2$ is the entry node of the invoked method, or (b) $n_1$ is the exit node of a method $m$ and $n_2$ is the return block of an invocation statement of method $m$. In the case of polymorphic call, edges are added in $E^1$ for all possible methods that can be invoked. $s_{main}$ is the entry node of the main method.

In an IFG two program points exist for each node $n$. A program point immediately before $n$ and a program point immediately after $n$. A program point can be reached by multiple paths during the execution. We use the notion of execution point to differentiate between the different execution contexts in which a program point can be reached. A path in an IFG from $s_{main}$ to a program point represents an execution point. However, all paths in an IFG are not valid paths. Valid paths are formally defined in [28]. An execution point $ep_2$ is reachable from an execution point $ep_1$ iff there exists a valid path from $ep_1$ to $ep_2$.

In a concurrent program we define an IFG for each thread class. $IFG_C = (N^*_C, E^*_C, s_{start})$ of a thread class $C$ is generated by considering start method as the main method. While computing the set $E^1_C$, a direct edge from the call block to the corresponding return block is added for method invocation statements for start(), wait(), and notify() methods.
4.2. Ensuring Request Execution Criteria

To fulfill the request execution criteria specified in Section 2.3 a request has to be executed either completely on the old program or completely on the new program. Since class is the unit of change, objects accessed during the execution of a request should be either from classes in the old program or classes in the new program. A request can access an object either by instantiating it or by invoking a method on it.

Let, \( O \) be the set of all objects used in the process. We define the following subsets of \( O \).

- \( O_{oc} \) : Set of old class objects.
- \( O_{nc} \) : Set of new class objects.
- \( O_{nbc} \) : Set of non backward compatible class objects.
- \( O_{nfc} \) : Set of non forward compatible class objects.

An ongoing request is either migrated on the new program or completed on the old program. A request \( r_n \) can be migrated to execute on the new program if \( r_n \) has not accessed an object in \( O_{oc} \cup O_{nbc} \). Execution of \( r_n \) will be as if it were executed on the updated process from the beginning.

Request \( r_n \) can accesses a non forward compatible class object \( o_f \in O_{nfc} \) before migration, and it will still satisfy the condition NRE. However, after \( r_n \) has modified \( o_f \), an old request \( r_o \) might use it. Hence, if \( r_n \) is migrated after it has accessed \( o_f \) then \( r_o \) can violate the condition ORE. Therefore, \( r_n \) can be migrated iff it has not accessed an object in the set \( O_{mp} = O_{oc} \cup O_{nfc} \cup O_{nbc} \). We call the set \( O_{mp} \) as migrate preventing set.

A request \( r_o \) which cannot be migrated has to be executed on the old program. However, the updation need not be postponed till the completion of \( r_o \). The program can be replaced during the execution of \( r_o \) at a point beyond which \( r_o \) will not access objects in \( O_{oc} \). Similarly, objects in \( O_{nbc} \) can be updated at a point beyond which \( r_o \) will not access them. Here it has been assumed that the new object \( o_n \), returned by an STF invoked on an object \( o_o \in O_{nbc} \), transparently takes over the identity of \( o_o \). Thus, the client object \( o_o \) holding a reference to \( o_o \) need not be updated.

As discussed in Section 3, after the completion of the update new requests can execute concurrently with the old requests. Hence, \( r_o \) can access an object \( o_n \in O_{nc} \) generated by a new request if \( o_n \) is a shared object of a type that is present in the old program. Therefore, we postpone the update till the point beyond which \( r_o \) will not access an object in \( O_{up} = O_{oc} \cup O_{nfc} \cup O_{nbc} \cup O_{nc} \). We call the set \( O_{up} \) as update preventing set.

Summarizing the above discussion, an update is safe if each request satisfies at least one of the following conditions.

\( C_{migrate} \) An object in the migrate preventing set \( O_{mp} \) has not been accessed before the update is completed.

\( C_{postpone} \) An object in the update preventing set \( O_{up} \) is not accessed after the update is started.

Execution points in the threads where the process can be updated without violating both the above conditions are safe points for the update. We now formally define the condition for safe update points. First we will consider that requests are executed using single threads and then we will generalize for the multi-threaded requests.
4.2.1 Single Threaded Request

Assume a request \( r_i \) is executed using a single thread \( \tau_i \) of the thread class \( C \). Let \( P_C \) be the set of all execution points in \( ICG \). We define two functions over execution points in \( P_C \) as follows:

- \( C.accessed(ep) \): Set of objects that can be accessed by a thread \( \tau_i \) of class \( C \) from execution point \( s_{start} \in P_C \) till execution point \( ep \) while \( \tau_i \) executes concurrently with the old request threads.
- \( C.canaccess(ep) \): Set of objects that can be accessed by a thread \( \tau_i \) of class \( C \) from \( ep \) till \( e_{start} \in P_C \) while \( \tau_i \) executes concurrently with the old request threads and the new request threads.

Let migrate set \( P^m_C \subseteq P_C \) be the set of points such that \( r_i \) will satisfy condition \( C_{migrate} \) if its thread \( \tau_i \) is at a point in \( P^m_C \) when the process gets updated.

\[
P^m_C = \{ ep | ep \in P_C \land C.accessed(ep) \cap O_{mp} = \emptyset \} \quad (1)
\]

Similarly, let postpone set \( P^p_C \subseteq P_C \) be the set of points for condition \( C_{postpone} \).

\[
P^p_C = \{ ep | ep \in P_C \land C.canaccess(ep) \cap O_{up} = \emptyset \} \quad (2)
\]

Execution points in migrate set \( P^m_C \) and postpone set \( P^p_C \) are safe update points. When \( P_C = P^m_C = P^p_C \) the thread class \( C \) remains unaffected by the update.

4.2.2 Multi Threaded Request

Many requests execute using multiple threads. For correct outcome of a request either all threads executing a request have to satisfy the condition \( C_{migrate} \) or all of them should satisfy the condition \( C_{postpone} \). However, all threads of a request may not interleave with the update. Some threads might have terminated before the update starts and some threads might start after the update is completed.

We use the following notations for threads participating in the execution of a request \( r_i \).

- \( T_i \): Set of thread classes used to execute a request \( r_i \).
- \( T^\text{terminated}_i \): Set of thread classes \( C \in T_i \) such that a thread of type \( C \) executing \( r_i \) has terminated before the start of the update.
- \( C.gen(ep) \): Set of thread classes whose instance can be started, directly or transitively, by the thread \( \tau_i \) of thread class \( C \) beyond point \( ep \in P_C \) while \( \tau_i \) executes concurrently with the old and the new request threads.

A thread which can access an object in the update preventing set \( O_{up} \), if started after the update, can violate the condition \( C_{postpone} \). We define a set of thread classes in \( T_i \) whose instance can access an object in the update preventing set \( O_{up} \) as \( T^u_i \).

\[
T^u_i = \{ C | C \in T_i \land C.canaccess(s_{start}) \cap O_{up} \neq \emptyset \}
\]

If request \( r_i \) cannot be migrated then any thread in \( T^u_i \) should not be started after the update. We redefine the postpone set \( P^p_C \) to ensure the same as follows:

\[
P^p_C = \{ ep | ep \in P_C \land C.canaccess(ep) \cap O_{up} = \emptyset \land C.gen(ep) \cap T^u_i = \emptyset \} \quad (3)
\]
A thread which is terminated before an update can violate the condition \( C_{\text{migrate}} \). A request cannot be migrated if any of its terminated threads might have accessed an object from the migrate preventing set \( O_{mp} \). Set of thread classes in \( T_i \) whose instance can access an object in the migrate preventing set \( O_{mp} \) is represented by \( T_i^{mp} \).

\[
T_i^{mp} = \{ C | C \in T_i \land C.\text{accessed}(e_{start}) \cap O_{mp} \neq \emptyset \}
\]

We now define the safety condition for update points.

**Safety Condition (SC)** For a request \( r_i \), if \( T_i^{mp} \cap T_i^{\text{terminated}} \neq \emptyset \) then the update points for all threads which are executing \( r_i \) should be in the corresponding postpone set \( P_p \). Otherwise, all its executing threads should remain, collectively, either in the corresponding migrate set \( P_m \) or in the corresponding postpone set \( P_p \).

For the safety condition, we define the following lemmas, and based on these lemmas we claim the sufficiency of safety condition.

**Lemma 1** If condition SC is met then an old class object cannot be accessed after an update has been started.

**Intuition:** Requests executing on the old program do not access any old class after update is started. Requests migrated on the new program can not instantiate any old class object. All old class objects created by the old requests are replaced by the STFs.

**Proof:** Assume that an object of an old class \( R \) has been accessed by a thread after the update has been started. The old class object could have been accessed either (i) during the update or (ii) after the update.

- **Case i:** As per the condition SC ips of all threads remain either in the migrate set \( P_m \) or in the postpone set \( P_p \) during the update. A thread can not access an old class object before leaving the migrate set \( P_m \). Similarly, a thread can not access an old class object after entering the postpone set \( P_p \).
- **Case ii:** After the update is completed constructor call to \( R \) will instantiate class \( R \) in the new program, and hence object of the old class \( R \) can not be instantiated after the update. All old class objects instantiated before start of the update are replaced by the STFs. Hence, after the update no old class object exists in the process.

**Lemma 2** If condition SC is met then a request can access either an old class object or a new class object but not the both.

**Intuition:** A new class object can not be accessed before the update is completed and an old class object can not be accessed after the update has been started. If a requests’ threads are at points in the postpone set \( P_p \) during the update then the request can not access the new class object after the update is completed. Otherwise the requests’ threads are at points in the migrate set \( P_m \) during the update and hence the request has not accessed an old class object before start of the update.

**Proof:** We present a proof by contradiction. Let, an old class object \( o_d \) and a new class object \( o_a \) are accessed by a request \( r_i \).
As per Lemma 1 an old class object can not be accessed after the update has been started. Therefore, a thread $\tau_1$ of the request $r_i$ must have accessed the object $o_f$ before start of the update. Hence, as per definition of the migrate set $P^{m}_m$ of the thread $\tau_1$ is outside the migrate set $P^{m}$ at the start of update. As per the condition SC the only update points outside the migrate set $P^{m}$ are points in the set $P^{p} - P^{m}$. Since the update points of thread $\tau_1$ can not be in the migrate set $P^{m}$, as per the condition SC the update points of all thread of the request $r_i$ must be in the corresponding postpone set $P^{p}$. However, a new class object can not be accessed after reaching a point in the postpone set $P^{p}$. □

Claim 1  Condition SC is sufficient to fulfill the request execution criteria.

Proof: We present a proof by contradiction. Let, both OPE and NPE be violated by a request $r_i$.

$r_i$ is not satisfying OPE means either (i) $r_i$ has accessed a new class object, or (ii) $r_i$ has accessed a non backward compatible class object after an STF has modified it, or (iii) $r_i$ has accessed a non forward compatible class object after it is accessed by a new request.

- Case i: As per Lemma 2 $r_i$ has not accessed an old class object. Hence, $r_i$ has been executed completely on the new program. Let, $\tau_1$ be a thread of request $r_i$ executing during the update such that either $\tau_1$ or a descendant of $\tau_1$ has accessed a new class object. As per definition of the postpone set $P^{p}$, during the update ip of the thread $\tau_1$ can not be in the postpone set $P^{p}$. As per condition SC only safe update points outside $P^{p}$ are the points in $P^{m} - P^{p}$. Since, update points of the thread $\tau_1$ are not in the postpone set $P^{p}$ as per the condition SC update points of all thread of the request $r_i$ must be in the corresponding migrate set $P^{m}$. As per the definition of set $P^{m}$ a non backward compatible class object is not accessed by any $r_i$ thread before update. Request $r_i$ satisfy condition NPE.

- Case ii: Let, $\tau_1$ be a thread of the request $r_i$ executing during the update such that either $\tau_1$ or a descendant of $\tau_1$ has accessed a new class object. As per the definition of postpone set $P^{p}$ ip of the thread $\tau_1$ can not be in the postpone set $P^{p}$ during update. On the similar lines of argument as in the Case i, request $r_i$ satisfy condition NPE.

- Case iii: Let, $\tau_1$ be a thread of the request $r_i$ which has accessed an object $o_f$ in the set $O_{nbc}$ after a thread $\tau_2$ of a new request has accessed it. Thread $\tau_2$ can not access object $o_f$ before update is completed and thread $\tau_1$ can not access object $o_f$ after update has been started. Hence, contradiction.

□

A request $r_i$ can be migrated only if the instruction pointers (ips) of all its live threads are in the corresponding migrate set $P^{m}$. However, the threads should not leave migrate set $P^{m}$ before the updation is completed. If migration is not possible then the updation has to be postponed till ips of all threads reach the points in corresponding postpone set $P^{p}$. We define the start and the exit boundary of a set of points $P$ in an IFG as follows:

**start**($P$) : Let $ep_1 \in P$ be the first execution point on a path $p$ from $s_{start}$ to $e_{start}$. $start(P)$ is a set of all such execution points on all paths in the IFG from $s_{start}$ to $e_{start}$.

**exit**($P$) : Let $ep_2 \in P$ be the last execution point on a path $p$ from $s_{start}$ to $e_{start}$. $exit(P)$ is a set of all such execution points on all paths in the IFG from $s_{start}$ to $e_{start}$.
We call the set of points \( \text{exit}(P^m_C) \) as the migrate cut and \( \text{enter}(P^p_C) \) as the postpone cut. These cuts are represented as \( \text{cut}^m_C \) and \( \text{cut}^p_C \), respectively. A cut postdominates an execution point \( ep \) if all paths from \( ep \) to the exit of the IFG contain a point in the cut. If \( ip \) of a thread is postdominated by the migrate cut then the \( ip \) is in the migrate set \( P^m \). If the \( ip \) is not postdominated by either the migrate cut or the postpone cut then the \( ip \) is in postpone set \( P^p \). Otherwise, the \( ip \) is in the unsafe region.

To migrate a thread \( \tau_i \) that is of type \( C \), blocking update calls are inserted at the corresponding migrate cut \( \text{cut}^m_C \). Blocking update calls prevent the thread to leave the migrate set \( P^m \) before the update is completed. Similarly to postpone the update, non blocking update calls are inserted at the corresponding postpone cut \( \text{cut}^p_C \). However, inserting a blocking update call can lead to a deadlock. Now we present an approach to avoid this.

### 4.3. Ensuring a Deadlock Free Update

An update can lead to a deadlock because of two reasons: (a) a new request and an old request may execute concurrently, and (b) blocking Update Calls (UCs) inserted for the update. In his section we present a solution to avoid the deadlock because of blocking UCs. Deadlock arising due to the former reason can be avoided by using known deadlock detection algorithms. We do not address the former cause in this report.

Figure 5 illustrates deadlock scenarios arising from blocking UCs. Deadlock may occur due to circular dependencies between thread \( \tau_1, \tau_2 \) and the update thread. In Figure 5(a) thread \( \tau_2 \) will wait at blocking UC for the update, update thread will wait for thread \( \tau_1 \) to execute the non blocking UC, and thread \( \tau_1 \) will wait at \( \text{wait} \) statement for \( \tau_2 \) to notify it. Thus the process will be in deadlock. Similarly, a blocking UC inside a synchronization block, as illustrated in Figure 5(b), can lead to a deadlock.

In general we see that dependencies between UCs along with the interthread dependencies in the program cause a deadlock. In order to model the interthread dependencies, we use a Parallel Execution Graph (PEG). In a PEG interthread dependencies in a program are represented as \( \text{notify} \) edges and \( \text{synchronization} \) edges. Now we briefly describe the PEG.
Parallel Execution Graph

PEG of a concurrent program is defined as \((N, E)\), where \(N = \bigcup C N^∗_C\) and \(E = E^0 \cup E^1 \cup E^p \cup E^n \cup E^s\). Here, \(E^0 = \bigcup C E^0_C\) and \(E^1 = \bigcup C E^1_C\).

Sets \(E^p\), \(E^n\), and \(E^s\) contain the parent-child edges, notify edges, and synchronization edges respectively.

1. parent-child edge: An edge \(e(n_1, n_2) \in E^p\) iff \(n_1\) invokes a \texttt{start} method and \(n_2\) is the entry node of the invoked \texttt{start} method.

2. Notify edge: An edge \((n_1, n_2) \in E^n\) iff \(n_1\) is a notify statement, \(n_2\) is a wait statement, and a thread waiting at the node \(n_2\) can be activated by the other thread on reaching the node \(n_1\).

3. Synchronization edge: An edge \((n_1, n_2) \in E^s\) iff \(n_1\) is \texttt{unlock}(o_1), \(n_2\) is \texttt{lock}(o_2), and \(o_1\) and \(o_2\) are aliases.

Note that an interthread dependency edge can be present between two nodes from the same IFG. A notify edge from node \(o.notify\) to node \(o.wait\) of an IFG represents that two thread instances of class \(C\) can have \texttt{wait notify} dependency between them.

Avoiding Deadlock

A thread which waits at a blocking UC gets released only after all threads with non blocking UC have executed their UCs and the process is updated. Therefore, similar to a wait notify dependency, threads with blocking UCs are dependent on all threads with non blocking UCs.

To avoid a dependency cycle involving a synchronization edge we ensure that a thread does not hold a lock while waiting for the update. For this we move the migrate cut (blocking UCs) before the start of the synchronized block. We formalize the same in the following:

Let \(N^{lock}_C\) be the set of lock nodes in \(N^*_C\), let \(n^u\) be an unlock node corresponding to a lock node \(n^l\), and let \(Path_C(b_1, b_2)\) be a predicate which is true if there exists a path in \(IFG_C\) from \(b_1\) to \(b_2\). Set of points in \(P^m_C\) which are inside a synchronized block that contains a point in \(cut^m_C\) is defined as \(P^a_C\).

\[
P^a_C = \{ ep | ep \in P^m_C \land \exists n^l, ep_m (n^l \in N^{lock}_C \land ep_m \in cut^m_C \land Path_C(n^l, ep_m) \land Path_C(ep_m, n^u) \land Path_C(n^l, ep) \land Path_C(ep, n^u)) \}
\]

\(P^a_C\) will contain all points in the migrate set \(P^m_C\) which are inside the synchronization block from \(n^l\) to \(n^u\) that contains a point \(ep_m \in cut^m_C\). To move the migrate cut before such synchronization block, we redefine the migrate set \(P^m_C\) as follows.

\[
P^m_C = P^m^* - P^a
\]

where, \(P^m^* \) stands for \(P^m_C\) in Equation 1

As illustrated in Figure 5(a) a dependency cycle can involve a \texttt{wait notify} edge. The dependency cycle is formed whenever a non blocking UC is reachable from a blocking UC through a notify edge in the PEG. To avoid the dependency cycle involving a \texttt{wait notify} edge we remove the blocking UCs. We redefine the migrate set \(P^m_C\) in the following for the same.
Let, $T$ be the set of all thread classes in the old program and $edges(ep_1, ep_2)$ be the edges on all paths in the PEG from $ep_1$ to $ep_2$. To remove the blocking UCs, wherever the corresponding migrate cut can lead to a deadlock, we redefine the migrate set $P^m$ as an empty-set as follows:

$$P^m_C = \begin{cases} \emptyset & \text{iff } \exists D, ep_1, ep_2 \ (D \in T \land ep_1 \in P^m_C \land ep_2 \in P^m_C \land (edges(ep_1, ep_2 \cap E^C) \neq \emptyset) ) \\ P^m_C \ast & \text{otherwise} \end{cases}$$

where, $P^m_C \ast$ stands for $P^m_C$ in Equation 4

Based on this new definition of migrate set $P^m_C$ we make the following claim.

**Claim 2** Condition SC is sufficient to avoid the deadlock when the migrate set $P^m$ and postpone set $P^p$ used in SC are as defined in Equation 5 and Equation 3 respectively.

**Proof:** Consider the sequence of events illustrated in Figure 4. A deadlock can not occur before the event e2 and after the event e4. Event e4 will be reached from the event e3 since beyond e3 the update thread does not depend on any process thread. Below we prove by contradiction that the event e3 will be reached from the event e2.

Let a deadlock happens before the process reaches e3 and $S$ be the set of threads involved in the deadlock. Since e3 has not been reached, a thread $\tau_a \in S$ has not reached the non blocking UC. Since old program threads can run concurrently without deadlock, a thread in $S$ must be waiting at a blocking UC. Let $S_b \subset S$ be the set of threads waiting at the blocking UCs.

Since the program is deadlock free, threads in $S - S_b$ can not be circularly waiting on each other. Hence, each thread in $S - S_b$ is either directly or transitively waiting for a thread in $S_b$. Let, $\tau_a, \tau_1, \ldots, \tau_n, \tau_b$ be a chain, where $\tau_a$ is waiting for $\tau_1$, $\tau_1$ is waiting for $\tau_2$ and so on, and $\tau_b \in S_b$. Let, thread $\tau_i \in S$ is waiting at $ip_i$. Since a thread $\tau_i$ is waiting for thread $\tau_{i+1}$ a path from $ip_{i+1}$ to $ip_i$ exists in PEG containing an edge from $E^n \cup E^s$. Therefore, there exists a path from $ip_b$ to $ip_a$ in the PEG.

As per definition of the migrate set $P^m$ thread $\tau_b$ is not holding any lock. Therefore, $\tau_n$ must be waiting for $\tau_b$ to notify it. Hence, the path from $ip_b$ to $ip_n$ must have an edge from $E^n$. This means the path from $ip_b$ to $ip_a$ contains an edge from $E^n$ which contradicts with the definition of the migrate set $P^m$.

Our update procedure discussed in Section 3 adds the update calls at a PUC, which is a set of update cuts of all affected threads in the process. In the next section we present a synopsis of the procedure to compute the same.

## 5 Scheduling Update

We compute a PUC in two steps. First step computes the cuts for all affected thread classes in the program. Second step selects an update cut for live threads in the process in order to form the PUC. Second step is executed at run time, as mentioned in Section 3. To minimize the run time overhead we have kept our second step simple by statically gathering the required information in the first step. The steps are as follows.

### 5.1 Static Analysis: Migrate Cut and Postpone Cut

Resolving virtual method calls to build an IFG is a crucial step in analyzing a program. Accuracy of migrate cut $cut^m_C$ and postpone cut $cut^p_C$ depend on the accuracy of the call edges $E^C$ in $IFG_C$. 

14
For example, let \( o : m() \) be an statement where \( o \) is of the type \( A \). \( A1 \) and \( A2 \) are the subtypes of \( A \), where \( A1 \) is a replaced class and \( A2 \) is an unmodified class. If an edge is present from \( o : m() \) to the entry of \( A1 : m() \) then the migrate cut should postdominate \( o : m() \), otherwise the statement \( o.m() \) will not affect the migrate cut. Therefore, accuracy of cuts depend on the accuracy of call edges in the IFG.

Various methods have been developed to compute IFGs of an object oriented program in the presence of dynamic binding. A simple approach is Class Hierarchy Analysis (CHA). Let \( S \) be the set of all classes having type same as the declared type of \( o \) or any of it subtype. In CHA a statement \( o.m() \) can invoke method \( m() \) from any class in the set \( S \). CHA is used to generate an IFG, which is then used to do more detailed program analysis and generate more accurate IFGs.

Migration cuts are computed on the IFGs in the old program, and postpone cuts are computed on the IFGs of a program which is combination of the old and the new program. In the combined program all replaced, replacing, deleted, added, and unmodified classes are present. A replacing class is named as \( C' \) and replaced class is named as \( C \). While computing the edges \( E1 \) for the combined program using CHA, it is assumed that a variable of type \( C \) can point to both \( C \) and \( C' \) and all their subtypes. Our implementation uses Variable Type Analysis (VTA) [31] which computes more more accurate IFG than CHA. While performing VTA of the combined program a new statement \( \text{new}C() \) is treated as to return an object of either an the old class \( C \) or a the new class \( C' \). We are implementing the analysis for Java using Soot [1].

5.2 Dynamic Analysis: Process Update Cut

During update depending on the execution status of the process either of the migrate cut and the postpone cut is selected for each affected thread. The scheduler first checks the execution status of each request and then decides to migrate the corresponding threads or to postpone postpone the update as per the condition SC.

In order to check the execution status of the requests, threads executing same request are grouped together. Let, \( \tau_0 \) be the main thread and \( \text{parent}(\tau_i) \) be the parent thread of the thread \( \tau_i \). Every thread started by the main loop executes a separate request. Therefore, a separate request group \( T_{\tau_i} \) is computed for each thread \( \tau_i \) having \( \text{parent}(\tau_i) = \tau_0 \). \( \tau_i \) and all descendant threads of \( \tau_i \) are added in the set \( T_{\tau_i} \).

We assume that the ip of the completed threads are at the end of their IFGs. First we check if a request \( \tau_i \) can be migrated on the new program. That is, for all threads in \( T_{\tau_i} \) their ips should be postdominated by the corresponding migrate cuts. If true then the migrate cuts of all threads in \( T_{\tau_i} \) are added in the PUC. Otherwise, the request can not be migrated and the update has to be postponed till all threads in \( T_{\tau_i} \) reaches a point in the corresponding postpone sets \( P_p \). Therefore, if the postpone cut of a thread postdominates its ip then the postpone cut is added in the PUC. Otherwise, the threads’ ip is already in the postpone set and hence no cut is selected for the thread.

Complexity

Here we discuss the time complexity of our dynamic analysis. Let, \( r \) be the number of ongoing requests and \( z \) be the number of statements in the old program. Complexity of categorizing the threads as per requests is linear to the number of executing requests. Postdominate check can be performed in \( O(z) \). Since, a cut has to be computed for all the requests the time complexity is \( O(r \times z) \).
6 Related Work

Online update of software systems was first discussed by Fabry [13]. The author classified the problem of online update in two subproblems: (i) “constructing the system in such a way that the program and the data structure which they manage can be changed without stopping the system”, and (ii) “how one convinces oneself that a change will operate correctly if it is installed”. The author only addressed the first problem and considered solution to the second problem to be nontrivial. Since then many researchers have addressed these problems, but most of the research is towards first problem, that is providing mechanisms for modifying a process. Comparatively less research is focused on the correctness of the update, which our work addresses. This section first briefly describe different solutions for the process modification and then discuss approaches to correctness.

6.1 Implementation Techniques for Online Update

Many researchers have extended existing dynamic linkers or defined their own dynamic re-linking libraries for managing the online update [18, 9, 3, 29, 13]. Information required by their re-linkers are generated during the program compilation. Additionally the compiler may insert hooks in the program. The hooks are used during relinking. Some researchers took the run time environment support for the online update. Extended virtual machines [8, 11] and distributed system middlewares [4, 19] can be used for the same. Another notable approach is purely program transformation based where program is transformed and supporting functionalists are added in the program [26]. Our solution to ensure the correctness can be used with most of these implementations to perform an online update.

6.2 Correctness in Online Update

Existing work on online update either do not provide any guarantee about the ongoing executions [2, 5, 12, 16, 19, 30] or depend on developers reasoning and input [14, 21, 24, 25] for the same. Following we briefly explain and compare them with our solution.

Stoyle et. al. [30] and Duggan [12] have presented two different approaches for a type safe update. Stoyle et. al. ensure that no code manipulates an instance of the new type assuming it to be of the old type. They use type mutation approach, where all instances of an old type are updated to the new type at once. Where as, Duggan [12] uses type versioning approach in which an instance can be converted from the old type to the new type and vice-versa. Expected version of the type of an instance is determined from the calling context and the version adapter is applied, if required. Only ensuring type safety during an update might not be sufficient, but it is necessary to ensure the correctness. Our request execution criteria guaranties a type safe update.

Gupta et. al. [16, 15] have formally proved that for an arbitrary old program and its new version it is undecidable to find whether a given update point is safe or not. Their condition for a safe update is, the process should reach a reachable state of the new program after the update. If a state can be reached by executing a program from the beginning then the state is a reachable state. They have proposed a sufficient condition for a safe update. We observed that the requirement of reachable state is to stringent. Therefore, our solution allow developers to specify STFs which may not lead the process to a reachable state of the new program but ensures that the new program can work correctly with the updated state.

Lee [21] has developed an update system for sequential programs written in StarMod language. A patch is divided into multiple units and applied in an order which guarantees type and semantic consistency of intermediate versions of the application. Developers can specify the set of procedures
which should not to be executing while updating a given procedure or variable. This way correct execution of semantically dependent procedure can be ensured. Frieder and Segal [14] guarantee stronger safety conditions than Lee [21]. Their mechanism ensures that a new procedure does not invoke an old procedure. However, an old procedure can invoke a new procedure via a mapper procedure which maps a call to an old method into an equivalent call on new method. To avoid any inconsistent behavior, they also allow developers to specify the set of procedures which should be inactive while updating a procedure. In our earlier work [24] we have defined a criteria to isolate execution of some methods from the update. We ensured the methods annotated by the developers as encapsulated methods are execute either on the old program or on the new program. The similar guarantee is provided by Neamtiu et. al. [25]. Though the authors present problem in the context of concurrent process they solve it only for sequential processes.

The challenge of synchronizing multiple threads in a concurrent process is similar to synchronizing update at multiple nodes in a distributed system. Bloom and Day [5], Sameer et al [2], and Kramer and Magee [19] presents the online update solutions for distributed systems. Bloom and Day avoid the need of synchronizing update among nodes by restricting the changes applied to a node. Sameer et. al. [2] also update the nodes independently and rely on the application to handle resulting errors. Kramer and Magee [19] depends on the application assistance to synchronize the update. Solution proposed in this report analysis the interthread dependencies in a program and accordingly synchronizes the threads during an update.

Other set of solutions for online update are application specific. For example, in databases [6, 22] and workflows [7, 27], considerable research addresses the problem of dynamic schema update. Practitioners have designed Telecommunication systems which can be dynamically updated [17]. Operating systems being impotent and critical, researchers are extending them to support the online patching [3, 23].

7. Conclusion

To minimize the disruption caused during software updates, this report proposes an online update solution that updates a running program while requests are in the middle of execution. The challenge of ensuring correct execution of requests during and after such an update is addressed in the report. Correct execution is ensured by updating the process at specific execution points called as update points. The update points are selected by analyzing the changes made in the patch and the interthread dependencies in the program. We have proved the correctness of the update point selection criteria.

Solution presented in this report has to be extended for prevalent process models like inversion of control and distributed execution. We believe that the proposed approach when implemented in practice will be helpful to reduce maintenance downtime of applications with long running requests.

References


