Correctness of Request Executions in Online Updates of Concurrent Object Oriented Programs

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

Abstract

Online update is a technique that reduces the disruption caused by a software update. It does so by applying a patch to a running process as opposed to shutting down the process and restarting it. The challenge here lies in ensuring correct operation during and after the update. In this paper, we present the correctness criteria involved in such situations and a solution to performing updates safely based on these correctness criteria. The approach we use 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 that execute during and after an update are ensured correct execution. Our literature survey reveals that this is amongst the first solutions 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 correctness it causes service downtime. An update can be scheduled in advance, however the disruption caused remains undesirable for high availability applications. An alternative is online update, where a running process is updated [8, 9, 10]. Guaranteeing the correctness of online update is difficult because update process has to ensure the correct outcome of ongoing executions. In this paper we present an online update solution for concurrent object oriented programs.

Most high availability applications process continuously arriving requests by spawning a new thread (or using a thread from a pool) to handle the request. These requests execute concurrently. Existing solutions for online update either abort some of the executing requests and restart them, or hold the new requests till the in-process requests complete and the process is updated. The former approach cannot be used where aborting an ongoing request execution is unacceptable. In the second approach, the application will be virtually unavailable till all executing requests complete. The requests can perform operation like file upload, ETL, and complex database query. The advantage of such an update will be lost if some requests are long running. In our research we look 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 part of the request might get executed on the old program and part of it on the updated program. The outcome of the request will neither be conformance with the old program nor with the updated program. For example, in a code fragment illustrated in Figure 1(a), a request is processed using an OrderHandler thread that 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 rather 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
We specify a request execution criteria to ensure the 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.

Following are the key contributions of this paper.

- We specify a request execution criteria to ensure the correct outcome of requests during and after the update (Section 2). We present an online update procedure to enforce this execution criteria (Section 3).
- We derive a safety condition to be satisfied by an update schedule to ensure the request execution criteria and to avoid deadlock. (Section 4)
- We present 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 5)

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 (main thread) 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. 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 after the patch 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 consists of persistent state. In the case of an online update, data also includes the state of the process. For an object oriented program a process state comprises of the set of objects that 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 that does not exist in the new program.
- **Added class**: A class in the new program that did not exist in the old program.
- **Replaced class**: A class in the old program that is modified in the new program.
- **Replacing class**: A class in the new program that is the modified version of a class in the old program.
- **Unmodified class**: A class that 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 changes then the class with the original name is considered to be deleted and the class with the new name is considered to be added. A class is considered modified in the new program if the new program contains a class with the same name as the old but with a different set of members or different method definitions. 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 a 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 String class remains unchanged. 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 \( \alpha_f \) 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 \( \alpha_f \) 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 paper we use the term compatibility to mean state compatibility. States of non backward compatible class objects are updated using State Transfer Functions (STF) \([4, 10]\) so that the new program can use them. 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 earlier 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.

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.

3. Applying the Online Patch

A patch may or may not affect execution of all threads in a process. We term a thread affected if interleaving the update with it can affect the correctness of the program execution. 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 an unsafe region. Figure 2 conceptually illustrates the unsafe region. The rationale 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 entry and the exit of the unsafe region.

![Figure 2. Safe and unsafe region](image)

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 3. Update procedure](image)

Figure 3 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 af-
ter 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.

From here on, we restrict our discussion to ensuring correctness, i.e. computing a PUC in our update procedure. Many techniques to patch a process exist [5, 6, 8, 11, 13] and we will not describe this here.

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. $E_m = E_m^0 \cup E_m^1$ is a set of edges. An edge $e(n_1, n_2) \in E_m$ iff there is direct transfer of control from block $n_1$ to $n_2$, and an edge $e(n_1, n_2) \in E_m^1$ 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 [17] 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, $E^0 = \bigcup_m E_m^0$, 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 a 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 [17]. 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 $E^*_C$, 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 \( O_{nc} \cup O_{nfc} \). Execution of \( r_n \) will be as if it were executed on the updated process from the beginning.

Request \( r_n \) can access 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 if it has not accessed an object in the set \( O_{mp} = O_{nc} \cup O_{nfc} \cup O_{nbc} \). 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 \( O_{nc} \). Similarly, objects in \( O_{nfc} \) 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 \( O_{up} = O_{nc} \cup O_{nfc} \cup O_{nbc} \cup O_{nc} \) even while it executes concurrently with the new requests. We call the set \( O_{up} \) as update preventing set.

In summary, an update is safe if each request satisfies at least one of the following conditions:

- ** occasion**: An object in the migrate preventing set \( O_{mp} \) has not been accessed before the update is completed.
- ** 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 \( IFG_C \). We define two functions over execution points in \( P_C \) as follows:

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

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

\[
P_{m} = \{ ep \in P_C \cap C.accessed(ep) \cap O_{mp} = \emptyset \} \tag{1}
\]

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

\[
P_{p} = \{ ep \in P_C \cap C.canaccess(ep) \cap O_{up} = \emptyset \} \tag{2}
\]

Execution points in migrate set \( P_{m} \) and postpone set \( P_{p} \) are safe update points. When \( P_C = P_{m} = P_{p} \) 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 \( r_i \).
- ** \( T_{i terminated} \)**: 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 \( T_{i up} \).

\[
T_{i up} = \{ C \in T_i \cap C.canaccess(s_{start}) \cap O_{up} \neq \emptyset \}
\]

If request \( r_i \) cannot be migrated then any thread in \( T_{i up} \) should not be started after the update. We redefine the postpone set \( P_{p} \) to ensure the same as follows:

\[
P_{p} = \{ ep \in P_C \cap C.canaccess(ep) \cap O_{up} = \emptyset \}
\]

A thread which is terminated before an update can violate the condition \( C_{migrate} \). A request cannot be migrated if any of its terminated threads might have accessed an object
from the migrate preventing set \( \mathcal{O}_{mp} \). Set of thread classes in \( T_i \) whose instance can access an object in the migrate preventing set \( \mathcal{O}_{mp} \) is represented by \( T_i^{mp} \).

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

We now define the safety condition for update points.

Safety Condition (SC) For a request \( r_i \), if \( T_i^{mp} \cap T_i^{terminated} \neq \emptyset \) 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.

**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.

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

See the technical report [14] for the proofs.

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:

\[
\begin{align*}
\text{start}(P) & : \text{Let } ep_1 \in P \text{ be the first execution point on a path } p \text{ from } s_{start} \text{ to } e_{start}. \text{ start}(P) \text{ is a set of all such execution points on all paths in the IFG from } s_{start} \text{ to } e_{start}. \\
\text{exit}(P) & : \text{Let } ep_2 \in P \text{ be the last execution point on a path } p \text{ from } s_{start} \text{ to } e_{start}. \text{ exit}(P) \text{ is a set of all such execution points on all paths in the IFG from } s_{start} \text{ to } e_{start}.
\end{align*}
\]

We call the set of points \( \text{exit}(P^p_{iC}) \) as the migrate cut and \( \text{enter}(P^p_{iC}) \) as the postpone cut. These cuts are represented as \( \text{cut}^m_{iC} \) and \( \text{cut}^p_{iC} \) 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 (UCs) are inserted at the corresponding migrate cut \( \text{cut}^m_{iC} \). Blocking UCs prevent the thread to leave the migrate set \( P^m \) before the update is completed. Similarly to postpone the update, non blocking UCs are inserted at the corresponding postpone cut \( \text{cut}^p_{iC} \). However, inserting a blocking UC 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 UCs inserted for the update. In this section we present a solution to avoid the deadlock because of blocking UCs. Deadlock arising due to the former reason can be avoided by locating the cause of deadlock using known deadlock detection algorithms and appropriately expanding the unsafe regions to avoid the same. We do not address the former cause in this paper.

Figure 4 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 4(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} \text{ statement for } \tau_2 \text{ to notify it. Thus the process will be in deadlock. Similarly, a blocking UC inside a synchronization block, as illustrated in Figure 4(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 notify edges and synchronization edges. Now we briefly describe the PEG.

### Parallel Execution Graph

PEG of a concurrent program is defined as \((\mathcal{N}, \mathcal{E})\), where \( \mathcal{N} = \bigcup \mathcal{N}_C^* \) and \( \mathcal{E} = \mathcal{E}^0 \cup \mathcal{E}^1 \cup \mathcal{E}^p \cup \mathcal{E}^n \cup \mathcal{E}^s \). Here, \( \mathcal{E}^0 = \bigcup \mathcal{E}_C^0 \) and \( \mathcal{E}^1 = \bigcup \mathcal{E}_C^1 \). Sets \( \mathcal{E}^p \), \( \mathcal{E}^n \), and \( \mathcal{E}^s \) contain the parent-child edges, notify edges, and synchronization edges respectively.

- **parent-child edge**: An edge \( e(n_1, n_2) \in \mathcal{E}^p \) iff \( n_1 \) invokes a start method and \( n_2 \) is the entry node of the invoked start method.
- **Notify edge**: An edge \( (n_1, n_2) \in \mathcal{E}^n \) 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 \( n_1 \).
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 $n^m$ 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 $P^m_C$.

$$P^m_C = \{ ep | ep \in P^m_C \land \exists n^l, ep_m(n^l) \in N_{lock}^C \land ep_m \in cut^m_C \}$$

$P^m_C$ will contain all points in the migrate set $P^m_C$ which are inside the synchronization block from $n^l$ to $n^m$ 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.

$$P^m_C = P^m_C - P^m_C$$

where, $P^m_C$ stands for $P^m_C$ in Equation 1

As illustrated in Figure 4(a) a dependency cycle can involve a 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 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 $P^m_C$ as an empty-set as follows:

$$P^m_C = \{ 0 \land \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^R = 0 \) \}$$

$$P^m_C$$ 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 $P^m_C$ and postpone set $P^m_C$ used in SC are as defined in Equation 5 and Equation 3 respectively.

See the technical report [14] for the proof.

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 the 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. Time complexity of the second step is $O(r \times z)$ where, $r$ is the number of in-process requests during the update and $z$ is the number of statements in the old program.

First step statically computes the cuts by analyzing the old and the new programs. Cuts of the thread classes are
computed by iterative data flow analysis. Migrate 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 programs. In the combined program all the replaced, replacing, added, deleted, and unmodified classes are present. We extend the Variable Type Analysis [19] for computing the IFGs of the combined program. We are implementing the analysis for Java using Soot [1]. We use the existing algorithms for the different operations, like computing PEG, which are performed in the first step. In [14] both the static and the dynamic analysis are presented in detail.

6. Related Work
Existing works on online update either do not provide any guarantee about ongoing execution [2, 3, 7, 10, 12, 18] or depend on developers reasoning and input [9, 13, 15, 16] for the same. We present them briefly as follows:

Lee [13] has developed an update system for sequential programs. They allow developers to specify a set of procedures which should not execute while updating a given procedure. Hence, the correct execution of semantically dependent procedures can be ensured. Frieder and Segal [9] guarantee stronger safety conditions than Lee [13]. 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. To avoid inconsistent behavior, they also allow developers to specify a set of procedures which should be inactive while updating a procedure. Murarka et al. [15] have defined a criteria to isolate execution of a method from an update. They ensure that the methods annotated by the developers are executed either on the old program or on the new program. A similar guarantee is provided by Neamtiu et al. [16].

Challenge of synchronizing multiple threads in a concurrent process is similar to synchronizing update at multiple nodes in a distributed system. Bloom and Day [3], Sameer et al. [2], and Kramer and Magee [12] present dynamic update solutions for distributed systems. Bloom and Day avoid the need of synchronizing updates among nodes by restricting the changes. Sameer et al. [2] also update the nodes independently and rely on the application to handle resulting errors. Kramer and Magee [12] depend on the application assistance to synchronize the update.

7. Conclusions & Future Work
To minimize the disruption caused during software updates, we proposed a solution that updates a running program while requests are in execution. Correct execution of requests during and after such an update is ensured by updating the process at specific execution points termed update points. Update points are selected by analyzing changes made in the patch and the interthread dependencies in the program. We have proved the correctness of the update point selection criteria.

The solution presented in this paper could be extended for prevalent process models like thread pipelining and distributed execution. We believe that the proposed approach when implemented in practice will be helpful to significantly reduce maintenance downtime.

Acknowledgment
We would like to thank Ravindra Guravannavar and Purushottam Kulkarni for their useful comments.

References