Reachability of Safe State for Online Update of Concurrent Programs

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

Abstract

Online update helps in reducing the maintenance downtime by applying a patch to a running process. To avoid the errors that can arise due to an online update, existing online update solutions interleave the update with process execution at specific states called safe states. This approach works well for sequential processes but for multi-threaded processes it presents the challenge of reaching a safe state across all threads. It turns out that in concurrent processes, even if individual threads can reach a safe state in bounded time a process may take unbounded time to reach a safe state or may never reach it. Therefore, with existing update solutions a user is uncertain whether a patch can be applied or not till the user tries to apply the patch.

In this report we address the problem of safe state reachability for multithreaded processes. We prove that identifying whether a process can reach a safe state is undecidable. Hence, we derive a sufficient (conservative) condition for checking the reachability of a safe state. Our novel thread scheduling technique can force a process to reach a safe state in bounded time if the sufficient condition is met. The proposed approach eliminates the uncertainty of online updates. Complexity analysis of our algorithms shows that the practical implementation of our technique will work well with most real-world applications.

1 Introduction

Application downtime, planned or unplanned, is an important concern in many domains. Software updates are a major reason for planned downtimes, and are inevitable due to the pressing need for adding features or fixing bugs and security vulnerabilities [10]. Online update techniques aim to reduce planned downtime by updating an application while it is running.

Online update techniques can be broadly categorized as versioning [3] and mutation [4, 5, 9, 6, 11, 12] depending on whether multiple versions of an entity (type/class/function) can co-exist in a process or not. Mutation technique is relatively simple and hence it is used by most online update solutions. To avoid any error that can arise due to updating a running process, mutation based techniques interleave the update with process execution at specific process states called safe states. Though this approach works well for sequential processes, for multithreaded processes it presents the challenge of ensuring that all threads of the process reaches a safe state together, before the update starts.
Majority of the existing update solutions for concurrent processes [9, 12, 15] use a time-out approach, where if the process reaches a safe state before the time-out period then the process is suspended at that safe state and the patch is applied. Otherwise, the update is aborted. With the time-out approach the user is uncertain whether a patch can be applied or not till the user attempts to apply the patch. Murarka et. al. [11] propose an approach that removes this uncertainty but their approach restricts the changes allowed in a patch. Most long running applications execute as multithreaded processes. Because of these limitations of online update solutions for multithreaded processes, online update is rarely used in practice. This paper proposes an update procedure which removes the uncertainty about the applicability of a patch without restricting the changes allowed in the patches.

Instead of waiting for the process to reach a safe state our update procedure forces the process to reach a safe state by scheduling the process threads in a particular order. We block the threads using update calls. Update call is similar to the breakpoint. On execution of an update call the thread that executes it gets suspended. The thread suspended at an update call can be resumed by the update thread which executes the update procedure. Our update call can be perceived as a special implementation of return barrier that is supported in Jvolve [15], a Java VM that allows online software update. We term a state where every thread in the process is blocked either at an update call or at an interthread synchronization statement like wait and lock statements a blocking state. The objective of the update procedure is to force a process to reach a blocking state which is safe for online update. We use the following steps to achieve this.

Step 1 : Suspend the process, add update calls in the threads at some specific points in execution, and resume the process.

Step 2 : Wait till the process reaches a blocking state.

Step 3 : If the blocking state is safe, stop. Otherwise, selectively release some threads that are suspended at update calls, and goto Step 2.

For example, consider a process illustrated in Figure 1.(a). Let, the safe state for an
online update is when both the threads $\tau_1$ and $\tau_2$ are at the last program points in their
loops. Hence, we add the update calls at these points as illustrated in the Figure 1.(a). The
process can enter in three blocking states: $(s_1) \tau_1$ at update call and $\tau_2$ at wait, $(s_2) \tau_1$ at
wait and $\tau_2$ at update call, and $(s_3)$ both $\tau_1$ and $\tau_2$ at update calls. We assume the original
program is deadlock-free and hence we do not consider the possibility of both $\tau_1$ and $\tau_2$
being at the wait state. Out of the three blocking states, the first two states are unsafe while
the third state $s_3$ is safe. Figure 1.(b) represents the possible transitions from the two unsafe
blocking states on releasing the thread which is blocked at a blocking update call. From
state $s_1$, on releasing thread $\tau_1$, the process can either re-enter the same state or move to
state $s_2$ or $s_3$. The process re-enters the same state only if thread $\tau_1$ reaches the end of the
loop without notifying thread $\tau_2$. Assuming that thread $\tau_2$ eventually receives a notify, the
process is guaranteed to move from state $s_1$ to either state $s_2$ or state $s_3$. If the process
moves to state $s_3$ then the safe state is reached. Otherwise, even if the process moves to
state $s_2$, the process can be forced to reach the safe state $s_3$ by releasing thread $\tau_2$.

We can now ask two questions about this approach:

1. Given a patch, is it always possible to force the process to reach a safe state in
   bounded time?

2. When multiple threads are blocked at an unsafe blocking state, which threads do we
   release to take the process to a safe blocking state?

In this report, we address both the questions. We first prove that even though each
thread in the process is guaranteed to reach a program point in a safe state in bounded time,
the first question is undecidable. The proof of undecidability is presented in Section 3.
Undecidability of the first question means it is not possible to find a computable necessary
and sufficient condition to check whether a patch can be applied online in bounded time.
Therefore, we develop a sufficient (conservative) condition for the same. We formulate
our approach to force a process to reach a safe state in Section 4. Our sufficiency condition
depends mainly on the state transition graph containing the blocking states. The same graph
is also used to answer which threads must be released in a given unsafe blocking state.
Section 5 presents the complexity analysis and discusses a way to reduce the computational
complexity of the solution. In Section 6 we describe the related work and in Section 7 we
present our conclusions.

In the next section we discuss the structure of the process considered for online update.

2 Process Structure

A process consists of a program and an execution state. A program is a set of classes, and
a class is a set of methods and data fields. Some classes in the program are marked as
_runnable_, by virtue of which a thread is associated with every object of it. We will use the
term _thread class_ to refer to a class that is marked _runnable_ and the term _thread_ to refer to
a process thread. 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 of the thread class objects.

3
Threads communicate with each other using the shared objects. A locking mechanism is implemented through the synchronized blocks in order to provide exclusive access to a shared object. 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 the wait() method of an object. However, a thread can invoke the wait() method over an object only if it is holding the lock over the same. The thread releases the lock over the object before becoming inactive. Another thread wakes up such an inactive thread by invoking the notify() method from a synchronized block over the same object. If multiple threads are waiting on the same object then the notify() activates one of the waiting threads non-deterministically.

The state of a process consists of the execution stacks of live threads and a heap of objects. Each execution stack in the process has an instruction pointer (ip) associated with it which corresponds to the instruction being executed.

Most of the continuously running processes have threads that run throughout the life of the process. We term such threads as the Infinitely Running (IR) threads. Each IR thread has a loop which the thread executes till the process ends. We term such a loop an infinite loop. In our process model, every thread spawned during an iteration of an infinite loop is a finite thread. That is, the spawned thread terminates within a finite time. For performance reasons, in many processes a pool of threads is maintained and a thread is allocated from the pool for each new thread request. On completing the execution the thread returns to the pool. These threads in the thread pool are finite threads.

We define an infinite loop executed by an IR thread an independent loop if it satisfies the following condition.

**Independent Loop:** A loop is independent if termination of all finite threads which are started, directly or transitively, during an iteration of the loop do not depend on whether the next iteration of the loop starts or not.

We assume that the infinite loops of the IR threads are independent loops.

Now we first discuss the control flow graph representation of a program and then formulate the safe states considered for applying the patches.

### 2.1 Program Graphs

#### 2.1.1 Flow Graph

Flow graph represents the flow of control among the statements of a method. Flow graph of a method $m$ is represented as $G^mc(N^m_c, E^m_c, n^m_s)$, where $N^m_c$ is the set of nodes representing the basic blocks, $E^m_c$ is the set of edges, and $n^m_s$ is the entry node of the method. In the flow graph, we represent a method invocation statement by two nodes, a call node and a return node. Furthermore, we add two nodes $n^m_{lock(a)}$ and $n^m_{unlock(a)}$ at the start and at the exit of a synchronized block, where $a$ is the variable pointing to the lock object of the synchronized block. The superscript of a node represents the method of the node and the subscript of a node differentiates between the various nodes of the flow graph of a method. We do not use superscript and subscript for a node when the context is clear.
Set of edges $\mathcal{E}_m = \mathcal{E}_m^c \cup \mathcal{E}_m^d$, where $\mathcal{E}_m^c$ is the set of control flow edges from the call nodes in the method and $\mathcal{E}_m^d$ is the set of control flow edges from the remaining nodes in the method. An edge $(n_i^m, n_j^m) \in \mathcal{E}_m^d$ if there is direct transfer of control from node $n_i^m$ to node $n_j^m$, and an edge $(n_e^m, n_i^m) \in \mathcal{E}_m^c$ if $n_i^m$ is a call node and $n_e^m$ is the corresponding return node. Every method $m$ has a unique entry node $n_s^m$ and a unique exit node $n_e^m$.

Control points in the execution of a method are defined as program points. In a flow graph, two program points $on_i$ and $no_i$ exist for each node $n \in \mathcal{N}_m$. $on_i$ is the program point immediately before $n$ and $no_i$ is the program point immediately after $n$. We represent a program point by $p_i$. $n(p_i)$ denotes the node to which a program point $p_i$ is associated. We say that a program point $p_d$ is reachable from a program point $p_s$ if a path exists from $n(p_s)$ to $n(p_d)$ in the flow graph, or the point $p_d = n_i \circ$ directly follows the point $p_s = on_i$ through a common node $n_i$. Formally the reachability among the program points is defined as follows:

**Definition 2.1** A program point $p_d$ is directly reachable from a program point $p_s$: denoted by $p_s \rightarrow_m p_d$, iff

(a) $p_s = on_i$ and $p_d = n_i \circ$, or
(b) $p_s = n_i \circ$, $p_d = on_j$, and $(n_i, n_j) \in \mathcal{E}_m$.

A program point $p_d$ is reachable from another program point $p_s$: denoted by $p_s \leftarrow_m p_d$, iff a series of program points $\rho = p_1, p_2, \ldots, p_k$ exists, with $p_s = p_1$, $p_d = p_k$, and $\forall 1 \leq i \leq n \rho_i \leftarrow_m p_{i+1}$. The series of program points $\rho$ is known as a path from point $p_s$ to point $p_d$. Alternatively, $\rho$ is also known as a path from node $n(p_s)$ to node $n(p_d)$.

### 2.1.2 Interprocedural Flow Graph

Interprocedural Flow Graph (IFG) represents the flow of control in a program, i.e. the flow of control within the methods and the flow of control across the methods. IFG joins the flow graphs of the methods through the interprocedural call and return edges. An IFG is defined for each thread class in a concurrent program. A sequential program has single thread of execution and hence only one IFG is defined for a sequential program. We now define the IFG for a sequential program and then extend it for a concurrent program.

Let $G_0^m$ be an isomorphic graph to $G_0^\mathcal{N}$ where a node $n_i^m$ in $N(G_0^m)$ is mapped to the node $n_i^m$ in $N(G_0^\mathcal{N})$. IFG of a sequential program is defined as $G_0^{\mathcal{N}}(N_0, \mathcal{E}_0, n_{s^{main}}^m)$, where $N_0$ is set of nodes, $\mathcal{E}_0$ is set of edges, and $n_{s^{main}}^m$ is the entry node of the main method. The set of nodes $N_0$ is the union of nodes in the flow graphs of all methods in the program, i.e. $N_0 = \bigcup_{\forall m} N(G_0^m)$. The set of edges $\mathcal{E}_0 = \mathcal{E}_0^c \cup \mathcal{E}_0^d$, where $\mathcal{E}_0^c = \bigcup_{\forall m} \mathcal{E}_{m}^c$ and $\mathcal{E}_0^d$ is the set of interprocedural call and return edges. An edge $(n_1, n_2) \in \mathcal{E}_0^d$ iff (a) $n_1$ is a call node 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 node of a method invocation statement which can invoke the method $m$. In the case of a polymorphic call, edges are added in $\mathcal{E}_0^c$ for all possible methods that could be invoked.

A node in an IFG uniquely maps to a node in the flow graph of a method and vice-versa. Therefore, here onwards we refer a node in a flow graph as a node in the IFG and the reverse way without specifying the mapping, unless required. Program points in an IFG are defined in the same way as those in flow graph and hence similar to the nodes, we refer
to the program points in an IFG and the flow graphs interchangeably. We use the function \( m(p_i) \) to represent the method to which program point \( p_i \) belongs and function \( m(n_i) \) to represent the method to which node \( n_i \) belongs.

A program point can be reached in multiple ways during an execution of the program. We define the execution points to differentiate between the various execution contexts in which a program point can be reached from the start of the program. Intuitively, an execution point, i.e., a program point along with its execution context, defines a control point in an IFG. We say that an execution point \( ep \) is reachable from an execution point \( ep_s \) if \( ep_d \) can be reached from \( ep_s \) during the program execution. Now we formally define the execution points and reachability among them.

**Definition 2.2** An execution point \( ep = \langle p, (n_1, \ldots, n_k) \rangle \) represents a program point \( PP(ep) = p \) and its execution context \( EC(ep) = (n_1, \ldots, n_k) \), where the execution context is a sequence of call nodes with \( n_1 \) being the first call node (from the main method) and \( n_k \) being the last call node (which calls method \( m(p) \)). In an execution context, a node \( n_i \) can be followed by another node \( n_{i+1} \) only if a call edge exists in the IFG from the node \( n_i \) to the entry node of method \( m(n_{i+1}) \). The set of all execution points in \( G_{0}^{FG} \) is represented as \( EP(G_{0}^{FG}) \).

An execution point \( ep_2 = \langle p_2, (n_{21}, \ldots, n_{2r}) \rangle \) is directly reachable from an execution point \( ep_1 = \langle p_1, (n_{11}, \ldots, n_{1k}) \rangle \): denoted by \( ep_1 \rightarrow_0 ep_2 \), iff any one of the following conditions is true.

a) Both the execution points have the same execution context: \( EC(ep_1) = EC(ep_2) \), and the program point of \( ep_2 \) is directly reachable from the program point of \( ep_1 \), except over a call edge in the flow graph:

\[
p_1 \rightarrow_{m(p_1)} p_2 \land \bigwedge (n_1, n_2) \left( (n_1, n_2) \in \mathcal{E}_{m(p_1)}^c \land n_1 \circ = p_1 \right)
\]

b) \( ep_2 \) is directly reachable from \( ep_1 \) over a call edge, i.e., a call edge \( (n_p, n_q) \in \mathcal{E}_{0}^c \) exists such that

(i) \( n_p \circ = p_1 \land n_q = p_2 \), and

(ii) \( k = r - 1 \land \forall 1 \leq i \leq k \left( (n_{1i} = n_{2i}) \land (n_{2r} = n_p) \right) \)

c) \( ep_2 \) is directly reachable from \( ep_1 \) over a return edge, i.e., a return edge \( (n_p, n_q) \in \mathcal{E}_{0}^c \) exists such that

(i) \( n_p \circ = p_1 \land n_q = p_2 \), and

(ii) \( r = k - 1 \land \forall 1 \leq i \leq r \left( (n_{1i} = n_{2i}) \land n_{1k} \in N(G_{m(p_2)}^{FG}) \right) \)

An execution point \( ep_d \) is reachable from an execution point \( ep_s \): \( ep_s \rightarrow_0 ep_d \), iff a series of execution points \( \rho = ep_1, ep_2, \ldots, ep_k \) exists, with \( ep_s = ep_1, ep_d = ep_k \), and \( \forall 1 \leq i < n \left( ep_i \rightarrow_0 ep_{i+1} \right) \). The series of execution points \( \rho \) is known as a path from \( ep_s \) to \( ep_d \). Alternatively, \( \rho \) is also known as a path from the node \( n(PP(ep_s)) \) to the node \( n(PP(ep_d)) \). We use the functions \( EP(\rho), P(\rho), E(\rho) \) and \( N(\rho) \) to represent the set of execution points, the set of program points, the set of edges, and the set of nodes on the path \( \rho \).
These functions are defined as follows:

\[
EP(\rho = ep_1, ep_2, \ldots, ep_k) = \{ ep_i | 1 \leq i \leq k \}
\]

\[
P(\rho) = \{ p_j | \exists ep_i (p_j = PP(ep_i) \wedge ep_i \in EP(\rho)) \}
\]

\[
E(\rho = ep_1, ep_2, \ldots, ep_k) = \{ (n_s, n_d) | \exists ep_i (1 \leq i < k \wedge PP(ep_i) = n_s \circ \wedge PP(ep_{i+1}) = n_d \}
\]

\[
N(\rho) = \{ n_i | \circ n_i \in P(\rho) \vee n_i \circ \in P(\rho) \}
\]

For a concurrent program, we define an IFG for each thread class in the program. To generate the IFG of a thread class \( C \), we consider the \texttt{start} method in class \( C \) as the main method; \( G^{IFG}_C = (N_C, E_C, n_s^{\text{start}}; C) \). While computing the interprocedural edges in \( E_C \), we add direct edges from the call nodes to the corresponding return nodes for those method invocation statements which invoke \texttt{start()}, \texttt{wait()}, and \texttt{notify()} methods.

An instruction pointer \( ip \) of a thread execution stack represents as an execution point in the IFG of the thread. The program pointer \( PP(ip) \) represents the program point immediately before the instruction being executed by the thread. Each method frame on the execution stack contains a return address. The statement immediately before the return address in a method frame corresponds to the call node of the method. Execution context \( EC(ip) \) represents the call nodes of the method frames on the execution stack.

### 2.2 Safe and Unsafe Regions

Many errors that can arise due to online update can be avoided by interleaving the updated at specific execution points [4, 5, 9, 6, 14, 12, 11]. We term such execution points as \textit{safe execution points}. Instead of dealing with individual safe execution points and unsafe execution points we group the contiguous safe and unsafe execution points in the sets called \textit{regions}.

**Definition 2.3** A set of execution points \( P \) in a graph \( G^{IFG}_C \) forms a \textit{region} iff for every pair of execution points \( \langle ep_1, ep_2 \rangle \in P \times P \) if a path exists from point \( ep_1 \) to point \( ep_2 \): \( ep_1 \xrightarrow{\rho} ep_2 \), then all execution points on the path \( \rho \) are also in the set \( P \): \( EP(\rho) \subseteq P \). □

Figure 2 conceptually illustrates the safe and unsafe regions in the IFG of a thread class. In this report we consider a process state is safe state for update if ips of all threads are in the safe region. The safe state considered by most of the online update solutions can be mapped in terms of our notion of safe and unsafe regions. For example, most of the existing solutions [4, 9, 15] define a state as a safe state if no method from some set of methods \( M \) is not in execution. We can represent such a safe state by defining the unsafe execution points as follows.
- All execution points whose program point is from a method in set $M$ are unsafe execution points.

- All execution points whose execution context contains a statement from a method in set $M$ are unsafe.

For a safe state to be reachable the IFG of each infinite loop should contain a safe execution point. As per the definitions of the safe state presented in [4, 9, 11] if the IFG of the infinite loop contains a safe execution point then the last execution point in the IFG of the loop is also a safe execution point. Therefore, in this report we assume that the last execution points in the infinite loops are safe execution points. Last execution point in an infinite loop is always reachable and hence it implies that a safe execution point is always reached in every iteration of an infinite loop.

### 3 Undecidability Results

We term the problem of finding reachability of safe states from every reachable state of a process as **attainability problem**. The process illustrated in Figure 1 can attain a safe state. Also, the process can be forced to attain a safe state in a bounded time. Here we are implicitly assuming that during the normal process execution, a thread at a wait state receives a notification in bounded time. Otherwise, the time required for a thread to come out of wait state can be unbounded and hence the process might not reach a safe state in bounded time. From here onwards we assume that the process satisfies the following properties.

**Bounded Wait** During normal process execution, a thread waiting at a `wait` statement receives a notify in bounded time. Similarly, a thread waiting to enter a synchronized block enters the synchronized block in bounded time.

**Bounded Loop** Execution time required for an iteration of an infinite loop is bounded.

Bounded loop implies that a thread completes an iteration of the infinite loop in a bounded number of steps, and if a thread is ready to execute then it gets scheduled to execute in bounded time. Bounded wait property and bounded loop property ensure that the waiting time and the execution time of an infinite loop iteration are bounded, and hence together they ensure that an IR thread completes an iteration of its infinite loop in bounded time. We cannot expect to reach a safe state in bounded time if any of the bounded wait and bounded loop properties does not hold.

Bounded wait property and bounded loop property are necessary to reach a safe state in a bounded time, but not sufficient. The process may keep moving from one unsafe blocking state to another unsafe blocking state in a cycle. Therefore, it may not be possible to give an upper bound on the number of times this cycle takes place before the process reaches a safe blocking state. Figure 3 illustrates a scenario where, by controlling the thread schedule, the process cannot be forced to reach a safe state in bounded time. In this example, thread $\tau_1$ is at the `lock` statement and thread $\tau_2$ is at the `wait` statement. On notifying thread $\tau_2$, thread $\tau_1$ enters a wait state. Thread $\tau_2$, after coming out the unsafe region, needs to enter the unsafe region again to notify thread $\tau_1$. After notifying thread $\tau_1$, thread $\tau_2$ can re-enter the wait state depending on condition $C$. The wait notify can continue for an unbounded
number of times and hence it is not possible to answer how much time the process will take to reach a safe state.

The sequence of transitions made from the unsafe blocking states in order to lead the process to a safe blocking state is called an update schedule. A transition from one blocking state to other blocking state is a step in the schedule. To force a process to reach a safe state in a bounded time, we need a schedule of a bounded length. We term the problem of finding whether a bounded length schedule to reach a safe state exists from each reachable state as the feasibility problem. An update is infeasible if a bounded length schedule does not exist.

As discussed in Section 2, the last execution point in the infinite loop body of each IR thread is a safe execution point. The last execution point in an infinite loop is reached during every iteration of the loop. Moreover, since the process satisfies the bounded wait and bounded loop properties, each IR thread in the process is guaranteed to complete an infinite loop iteration in bounded time. Therefore, each thread in the process will reach a safe point in bounded time. Though each individual thread in a process is guaranteed to reach a safe update point in bounded time, the attainability problem and the feasibility problem are undecidable. Following are the proofs for the same.

**Theorem 1** For an arbitrary program \( \Pi \) and a state \( s_u \) such that \( s_u \) is an unsafe but reachable state of \( \Pi \), it is undecidable whether any safe state of \( \Pi \) is reachable from \( s_u \).

**Intuition:** The proof is similar to the proof for undecidability of halting problem.

**Proof:** Suppose the problem is decidable. That is, if \( S_s \) is the set of all safe states in \( \Pi \) then there exists a function

\[
A(\Pi, s_u) = \begin{cases} 
true & \text{if in program } \Pi, \text{ a state in } S_s \text{ is} \\
false & \text{reachable from state } s_u \\
false & \text{otherwise}
\end{cases}
\]

Now consider a program \( \Pi' \) with two threads as illustrated in Figure 3. Let state \( s_u \) be thread \( \tau_1 \) at \( a.lock \) and thread \( \tau_2 \) at \( a.wait \). A possible safe state of \( \Pi' \) is: both the threads at

![Figure 3: Safe State is Unattainable in Bounded Time](image-url)
the points immediately after the end of the unsafe regions. In \( \Pi' \), a state in \( S_s \) is reachable from \( s_u \) only if condition \( C \) in thread \( \tau_2 \) becomes false in some iteration. Now we define the condition \( C \) as \( A(\Pi', s_u) \). In \( \Pi' \), the condition becomes false only if \( A \) returns that a state in \( S_s \) is not reachable from \( s_u \). Therefore, if \( A \) says \( \Pi' \) cannot reach a state in \( S_s \), then \( \Pi' \) can reach a state in \( S_s \), and if \( A \) says \( \Pi' \) can reach a state in \( S_s \) then \( \Pi' \) can never reach any state in \( S_s \). In either case, \( A \) gives the wrong answer for \( (\Pi', s_u) \). Therefore, \( A \) cannot exist.

Theorem 2  For an arbitrary infinitely running program \( \Pi \) and state \( s_u \) such that \( s_u \) is an unsafe but reachable state of \( \Pi \), it is undecidable whether there exists a bounded length schedule to reach any safe state in \( \Pi \) from \( s_u \).

Proof: The proof is the same as the proof of Theorem 1 with \( A \) modified to return true iff: in program \( \Pi \), a state in \( S_s \) is reachable from state \( s_u \) in bounded time.

Undecidability of attainability problem implies we cannot find a condition that is both necessary and sufficient to check the attainability of a safe state. Similarly, undecidability of feasibility problem implies we cannot find a condition that is both necessary and sufficient to check the feasibility of reaching a safe state. In the following section, we derive a sufficient condition to check feasibility of reaching a safe state. The sufficient condition for feasibility also holds for attainability. The section also presents an algorithm to compute a schedule to reach a safe state if the sufficient condition is met.

4 Reaching a Safe State

We call a state transition diagram consisting of all blocking states and the transitions among them a Blocking State Graph (BSG). We use BSG for checking the feasibility of an update and to compute the update schedule. The update schedule consists of the transitions to be taken from unsafe blocking states to reach a safe state. Remember that, in a blocking state one or more threads are suspended at the update calls. Resumption of some of these suspended threads leads to a state transition in the BSG.

Since the aim is to take the process to a safe state, the update calls must be inserted at the appropriate points in the program and the right transition must be made from each blocking state. That is, we must selectively hold and release the process threads to force the process to enter a safe state. To ensure each IR thread reaches a safe point in bounded time, we add the update calls at the last program points in the infinite loops of the IR threads. We also insert the update calls at the entry of an unsafe region if the unsafe region contains a \textit{wait} statement. This way we try to prevent a thread getting blocked at an unsafe execution point. However, as discussed in Section 1, a thread \( \tau_s \) may need to enter an unsafe region to take another thread \( \tau_u \) out of an unsafe region. In this case, we release all threads that are held at update calls and on which thread \( \tau_u \) is dependent, directly or transitively, to come out of the unsafe region.

The BSG models the controlled execution of the process discussed above. The intuition behind the sufficient condition for update is that the BSG should not contain a cycle such
that every blocking state in the cycle is an unsafe blocking state. If BSG contains such a cycle then the process can loop in the unsafe states for an unbounded time.

For simplicity of exposition, in this section, we first consider a restricted process model where a process consists of infinitely running (IR) threads only. Subsection 4.1 discusses the construction of the BSG of such processes. Subsection 4.2 presents the sufficient condition for feasibility of an update. The sufficient condition is defined in terms of the BSG. In Subsection 4.3, we discuss the construction of the BSG for a process with both IR and finite threads. Subsection 4.4 presents the schedule to force a process to enter a safe state.

4.1 Process with Only Infinitely Running Threads

To compute the BSG of a process, we first compute a process graph. From the process graph, we construct a state transition graph called a State Graph. The state graph represents the transitions among selected states of a process. State graph is later abstracted in terms of Blocking State Graph which consists of only blocking states in the state graph. In the following, we present the construction of the process graph, the state graph, and the blocking state graph of a process.

4.1.1 Process Graph

Process graph represents the flow of control within each thread in the process and the interthread dependencies. We assume that the IR threads in a process are fixed and their list is known. That is, a new IR thread will not start and the existing IR threads will continue to run till the update completes. The set of IR threads in a process can be identified by monitoring the process execution or the set can be provided by the user.

Let $T^{live} = \{ \tau_1, \ldots, \tau_n \}$ be the set of IR threads running in a process. We define an interprocedural flow graph $G_{\tau_i}^{tIFG}$ for each thread $\tau_i$ in $T^{live}$. The graph is isomorphic to the IFG of the thread class of $\tau_i$. A node $n_{mc}^{i}:C$ in $N(G_{\tau_i}^{IFG})$ is mapped to the node $n_{mc}^{i}:C$ in $N(G_{class(\tau_i)}^{IFG})$. The process graph is built by combining the tIFGs of all live threads in the process and adding interthread dependency edges. Let $G^*$ be the union of all tIFGs in the process.

$$G^* = \bigcup_{\tau_i \in T^{live}} G_{\tau_i}^{tIFG}$$

The process graph is defined as $G^{po}(N(G^*), E(G^*) \cup E^n \cup E^i)$, where $E^n$ and $E^i$ are set of interthread notify edges and interthread data dependence edges respectively.

- **Notify edges ($E^n$):** An edge $(n_{ni}^{i}, n_{nwi}^{i}) \in E^n$ iff $n_{ni}^{i}$ is a notify statement, $n_{nwi}^{i}$ is a wait statement, and if execution of $n_{ni}^{i}$ by thread $\tau_i$ can activate thread $\tau_j$ when $\tau_j$ is waiting at node $n_{nwi}^{i}$. Edges $E^n$ define the wait notify dependencies among the threads in the process.

- **Interthread Data Dependence Edges ($E^i$):** In a PG, node $n_{i2}^{i,j}$ is interthread data dependent on node $n_{i1}^{i,j}$ iff the following conditions hold.
  1. $i \neq j$ and threads $\tau_i$ and $\tau_j$ can execute in parallel.
  2. An object field $o.f$ is defined at $n_1$ and used at $n_2$. 


3. If \( n_1 \) is in a synchronized block then the definition at \( n_1 \) reaches the exit of the synchronized block. Similarly, if \( n_2 \) is in a synchronized block then the usage at \( n_2 \) reaches the entry of the synchronized block.

Interthread data dependence is also known as interference dependence [8].

Execution points and program points in the PG are defined in the same way as that of IFG. Due to interthread dependence edges in the PG, an execution point \( e_{\rho}d \) in an tIFG can be reachable from an execution point \( e_{\rho}s \) in another tIFG. Below, we define the reachability among the execution points in PG.

**Definition 4.1** An execution point \( e_{\rho}d \) is directly reachable from an execution point \( e_{\rho}s \): denoted by \( e_{\rho}s \rightarrow e_{\rho}d \), iff (a) both the execution points are from the same tIFG and \( e_{\rho}d \) is directly reachable from \( e_{\rho}s \) in the tIFG, or (b) there exists an interthread dependence edge \((n_1, n_2) \in (E^n \cup E^l)\) such that \( PP(e_{\rho}s) = n_1 \circ \) and \( PP(e_{\rho}d) = n_2. \)

An execution point \( e_{\rho}d \) is reachable from another execution point \( e_{\rho}s \): \( e_{\rho}s \rho \rightarrow e_{\rho}d \), iff a series of execution points \( \rho = e_{\rho}p_1, e_{\rho}p_2, \ldots, e_{\rho}p_k \) exists, with \( e_{\rho}s = e_{\rho}p_1 \wedge e_{\rho}d = e_{\rho}p_k \wedge \forall 1 \leq i < n e_{\rho}p_i \rightarrow e_{\rho}p_{i+1}. \) The series of execution points \( \rho \) is known as a path from \( e_{\rho}s \) to \( e_{\rho}d \). Alternatively, \( \rho \) is also known as a path from node \( n(PP(e_{\rho}s)) \) to node \( n(PP(e_{\rho}d)) \). Similar to the notations for a path in an IFG, we use the functions \( E(\rho) \) and \( N(\rho) \) to represent the set of edges and the set of nodes on the path \( \rho \). Definition of these functions in same as that for a path in an IFG. □

### 4.1.2 State Graph

From the process graph we construct a state graph. State graph models the process states and transitions among them. In a state graph, we do not consider every process state. We consider only those states that are useful to compute the blocking state graph. A blocking state is represented as the positions of the ips of all threads in the process where they could be blocked, together. To check whether a process state is safe or unsafe knowing the positions of the ips of all threads in that state is sufficient. Hence, we model a process state in the state graph as the positions of the ips of all threads in the process.

To compute the blocking state graph, we need to model each state in which a thread can get blocked. That is, the state graph has to capture the positions of the ip of a thread where it can get blocked. A thread can get blocked at a wait statement and at an update call. Hence, we consider these thread states in the state graph. A thread in a wait state comes out of it on receiving a notify. Hence, we also model the position of ips of threads corresponding to notify statements.

For simplicity, we assume that no thread in the process holds a lock if it is in a wait state. Also, while inserting a update call we ensure that an update call is not inside a synchronized block. Hence, at a blocking state a thread cannot be blocked at a lock node. Therefore, we need not consider the thread states corresponding to the lock nodes. We enumerate the states of a thread used in the construction of a state graph in the following.

- **Wait state:** A wait state corresponds to an execution point immediately before a wait statement. The set of all wait states in a thread \( \tau_i \) is defined as:

\[
T^w_{\tau_i} = \{ e_{\rho}k \mid PP(e_{\rho}k) = n_{\tau_i}^i \circ \land n_{\tau_i}^i \text{ is a wait statement} \}
\]
- **Notify state:** A notify state corresponds to an execution point immediately before a notify statement. The set of all notify states in a thread \( \tau_i \) is defined as:

\[
\mathcal{I}^n_{\tau_i} = \{ \text{ep}_k \mid PP(\text{ep}_k) = \circ n^{\circ i}_k \land n^{\circ i}_k \text{ is a notify statement} \}
\]

- **Hold state:** A hold state corresponds to an execution point where the update call is inserted during the update. Recall that, we insert the update calls at the entry of an unsafe region if the unsafe region contains a \textit{wait} statement. The set of entry points of a region \( \mathcal{P} \) is defined as follows:

**Definition 4.2** Let \( \text{ep}_1 \in \mathcal{P} \) be the first execution point on a path from the entry node \( n^{\text{start}:C}_s \) of the IFG to the exit node \( n^{\text{start}:C}_e \) of the IFG. The set of all such execution points in the set \( \mathcal{P} \) over all the paths in the IFG from \( n^{\text{start}:C}_s \) to \( n^{\text{start}:C}_e \) forms the set of entry points of region \( \mathcal{P} \).

Each entry point of an unsafe region that contains a wait state represents a hold state. If the entry point of an unsafe region is in a synchronized block then the point immediately before the start of the synchronized block is used as the hold state.

We also consider the last program point in the body of the infinite loop as a hold state, and the state is termed \textit{End-of-iteration (EoI) state}. EoI state is used to ensure that each IR thread reaches a safe point in bounded time. EoI state also helps us to model the start of a new iteration of an infinite loop.

The set of hold states of a thread \( \tau_i \) is represented as \( \mathcal{I}^h_{\tau_i} \).

- **Post-wait state:** A post-wait state corresponds to an execution point immediately after a wait statement. A post-wait state is used to model the transition of a thread out of a wait state.

\[
\mathcal{I}^w_{\tau_i} = \{ \text{ep}_k \mid PP(\text{ep}_k) = n^{\circ i}_k \circ \land n^{\circ i}_k \text{ is a wait statement} \}
\]

- **Post-notify state:** A post-notify state corresponds to an execution point immediately after a notify statement. A post-notify state is used to model the trigger of notify.

\[
\mathcal{I}^n_{\tau_i} = \{ \text{ep}_k \mid PP(\text{ep}_k) = n^{\circ i}_k \circ \land n^{\circ i}_k \text{ is a notify statement} \}
\]

- **Post-hold state:** A post-hold state is used to model the release of a thread from a hold state. A post-hold state corresponds to the same execution point as that of a hold state. However, to differentiate between a hold and a post-hold state, we represent a post-hold state by a dummy execution point \( \text{ep}' \) immediately after a hold state \( \text{ep} \). We term a post-hold state corresponding to an EoI state as \textit{Post-EoI state}.

\[
\mathcal{I}^h_{\tau_i} = \{ \text{ep}'_k \mid \text{ep}_k \in \mathcal{I}^h_{\tau_i} \}
\]

Set of all states of a thread is given by:

\[
\mathcal{I}_{\tau_i} = \mathcal{I}^n_{\tau_i} \cup \mathcal{I}^w_{\tau_i} \cup \mathcal{I}^h_{\tau_i} \cup \mathcal{I}'_{\tau_i} \cup \mathcal{I}^w_{\tau_i} \cup \mathcal{I}^h_{\tau_i}
\]
State graph \( G^{SG} \) of a process is defined as \( (S, E) \), where \( S \) is the set of states and \( E \) is the set of transitions among the states due to execution of a thread or release of threads from the hold states. Set of states of the process is defined as the cross product of states of all threads:

\[
S = I_{\tau_1} \times I_{\tau_2} \ldots \times I_{\tau_n}
\]

We use the notation \( s_1 : \tau_i \) to denote the state of thread \( \tau_i \) in process state \( s_1 \). The set of transitions \( E \) is defined as \( E^e \cup E^w \cup E^r \), where \( E^e, E^w \) and \( E^r \) represent the set of execution edges, wait notify edges and release edges respectively.

- **Execution edge:** An execution edge \( e_{\tau_i} \) represents a change of state due to the execution of the thread \( \tau_i \). An edge \( e_{\tau_i}(s_1, s_2) \in E^e \) iff: (a) \( \tau_i \) is neither at wait nor at hold in state \( s_1 \), (b) in \( s_2 \), except \( \tau_i \) every other thread is in the same state as that in \( s_1 \), and (c) there exists a path in \( G^{tIFG}_i \) from \( s_1 : \tau_i \) to \( s_2 : \tau_i \), without any state in \( I_{\tau_i} \) on the path.

- **Wait notify edge:** A wait notify edge \( e_{\tau_i} \) represents a change of state due to the execution of a notify statement in thread \( \tau_i \). The execution of a notify statement results in the release of some other thread \( \tau_j \) from its wait state. An edge \( e_{\tau_i}(s_1, s_2) \in E^n \) iff: (a) \( \tau_i \) is in notify state in \( s_1 \) and there exists a thread \( \tau_j \) in wait state in \( s_1 \), (b) process graph contains a notify edge from the notify node in \( \tau_i \) to the wait node in \( \tau_j \) (i.e., from \( n_PP(s_1 : \tau_i) \) to \( n_PP(s_1 : \tau_j) \)), and (c) in \( s_2 \), \( \tau_i \) is in the corresponding post-notify state, \( \tau_j \) is in the corresponding post-wait state, and every other thread is in the same state as that in \( s_1 \).

- **Release edge:** A release edge represents change of state after release of some threads that are at hold. The source state of a release edge is always a blocking state. If thread \( \tau_1 \) is at wait in blocking state \( s_1 \) then an outgoing release edge \( e_{\tau_i} \) from \( s_1 \) is added in the state graph. The edge represents the state transition after release of all threads which are at hold in \( s_1 \) and on which thread \( \tau_i \) is dependent to receive a notify.

To identify the threads on which thread \( \tau_i \) is dependent to receive a notify, we first define dependence among the nodes in the process graph. The \textit{depends} relation from a node \( n_i \) to a node \( n_j \) represents that either the execution of \( n_i \) or the outcome of the execution of \( n_i \) depends on the execution of \( n_j \).

**Definition 4.3** Let \( n_i \) and \( n_j \) be two nodes in the process graph. Node \( n_i \) \textit{directly depends} on node \( n_j \) iff any of the following conditions hold.

(a) \( n_i \) is \textit{control dependent} or \textit{data dependent} on \( n_j \), where control dependency and data dependency is defined as follows.
   - Let \( n_1 \) and \( n_2 \) be nodes in an tIFG. We say \( n_2 \) is \textit{control dependent} on \( n_1 \) iff the execution of \( n_1 \) decides whether \( n_2 \) will be executed or not.
   - Let \( n_1 \) and \( n_2 \) be two nodes in an tIFG. If a variable \( v \) (or an object field \( o.f \)) defined at \( n_1 \) is used at \( n_2 \) and there is a path from \( n_1 \) to \( n_2 \) on which \( v \) (or the object field \( o.f \)) is not redefined, then \( n_2 \) is \textit{data dependent} on \( n_1 \).

(b) \( n_j \) is a wait node and both \( n_i \) and \( n_j \) are from the same tIFG.

(c) An interthread data dependence edge \( (n_j, n_i) \) or a notify edge \( (n_j, n_i) \) exists in the process graph.
The second condition is introduced to reflect that if a thread is blocked at a wait statement then the other nodes in the thread can execute only after the thread comes out of the wait state.

Node $n_i$ depends on node $n_j$, denoted by $\text{dep}(n_i, n_j)$ iff: (a) $n_i$ directly depends on $n_j$, or (b) a node $n_k$ exists in the process graph such that $n_i$ directly depends on $n_k$ and $n_k$ depends on $n_j$. □

Let $\tau_i$ be a thread which is at wait in state $s_1$. A thread $\tau_j$ can notify thread $\tau_i$ if in the process graph, there exists a notify edge from a notify node in $\tau_j$ to the wait node $n(PP(s_1 : \tau_i))$. Let $\mathcal{E}_{s_1 \tau_i}$ be the set of notify edges in the process graph such that $n(PP(s_1 : \tau_i))$ is their destination node. Let $\mathcal{N}^{n}_{s_1 \tau_i}$ be the set of source nodes of the edges in set $\mathcal{E}_{s_1 \tau_i}$.

Threads whose tIFGs contain a node in set $\mathcal{N}^{n}_{s_1 \tau_i}$ can notify thread $\tau_i$. However, a thread which contains a node in set $\mathcal{N}^{n}_{s_1 \tau_i}$ may depend on some other thread to execute the notify statement. For example, in Figure 4, thread $\tau_2$ depends on thread $\tau_3$ to execute the statement $\text{a.notify()}$. Let $G^{PG'}$ be a subgraph of process graph $G^{PG}$ derived after removing the tIFG of thread $\tau_i$ from $G^{PG}$. Let $\mathcal{N}^{d}_{s_1 \tau_i}$ be the set of nodes on which nodes in set $\mathcal{N}^{n}_{s_1 \tau_i}$ depend in $G^{PG'}$. We remove thread $\tau_i$ while computing the dependence relation (i.e., set $\mathcal{N}^{d}_{s_1 \tau_i}$) because a dependence through a node in thread $\tau_i$ is of no interest since $\tau_i$ is at wait. Execution of nodes that are not in set $\mathcal{N}^{d}_{s_1 \tau_i}$ cannot influence the execution of a notify node in set $\mathcal{N}^{n}_{s_1 \tau_i}$. For example, in Figure 4, notify node in thread $\tau_2$ does not depend on the nodes in thread $\tau_4$ and hence execution of thread $\tau_4$ cannot influence execution of notify statement in thread $\tau_2$. Therefore, releasing all threads that contain a node in set $\mathcal{N}^{n}_{s_1 \tau_i} \cup \mathcal{N}^{d}_{s_1 \tau_i}$ is sufficient to ensure that thread $\tau_i$ receives a notify. Set $\mathcal{N}^{n}_{s_1 \tau_i}$
and $\mathcal{N}_{s_1\tau_i}^d$ are formally defined as follows.

\begin{align}
\mathcal{N}_{s_1\tau_i}^n &= \{ n_1 \mid (n_1, n(s_1: \tau_i)) \in \mathcal{E}^n \} \\
&\text{where, } \mathcal{E}^n \text{ is set of notify edges in } G^{PG} \\
\mathcal{N}_{s_1\tau_i}^d &= \{ n_1 \mid \exists n_2 \ (n_2 \in \mathcal{N}_{s_1\tau_i}^n \land \text{dep}(n_2, n_1)) \} \\
&\text{where, } \text{dep}(n_2, n_1) \text{ is defined over } G^{PG'}
\end{align}

We release the smallest set of threads such that $\tau_i$ receives a notify. Consider the process illustrated in Figure 5. Let the process be at a blocking state 1w2h3h, i.e., $\tau_1$ is at wait and $\tau_2$ and $\tau_3$ are at hold. In this state, thread $\tau_1$ directly depends on thread $\tau_2$, and through thread $\tau_2$ also depends on thread $\tau_3$. That is, according to the definition of depends relation, to receive a notify $\tau_1$ depends on both $\tau_2$ and $\tau_3$. However, we can selectively release only thread $\tau_2$. If thread $\tau_2$ enters in a wait state then the process will be in blocking state 1w2h3h. At this state, we release thread $\tau_3$. If thread $\tau_3$ is released in state 1w2h3h then thread $\tau_3$ can itself enter a wait state. In general, if thread $\tau_i$ is at wait and thread $\tau_j$ can notify it then thread $\tau_j$ is released from the hold state. Moreover, if thread $\tau_j$ itself is at wait and thread $\tau_k$ can notify it then thread $\tau_k$ is released because thread $\tau_j$ indirectly depends on thread $\tau_k$. However, if thread $\tau_j$ is not at wait then we do not release thread $\tau_k$.

Let $T_{s_1\tau_i}^{\text{notify}}$ be the set of threads which are at hold in $s_1$ and on which thread $\tau_i$ is dependent. Equation 2 formally defines set $T_{s_1\tau_i}^{\text{notify}}$. Thread $\tau_i$ depends on thread $\tau_j$ which is at hold state in state $s_1$ iff: (a) $\mathcal{N}_{s_1\tau_i}^n \cup \mathcal{N}_{s_1\tau_i}^d$ contains a node from the tFG of thread $\tau_j$ such that in the process graph the wait node $n(PP(s_1: \tau_i))$ of thread $\tau_i$ is reachable from that node over a path $\rho$, and (b) every thread which contains the destination node of a notify edge on the path $\rho$ is in the wait state.

\begin{align}
T_{s_1\tau_i}^{\text{notify}} &= \{ \tau_j \mid s_1: \tau_j \in T_{\tau_j}^h \land \exists n_{\tau_j}^i, n_d, \rho ( \\
&n_{\tau_j}^i \in (\mathcal{N}_{s_1\tau_i}^n \cup \mathcal{N}_{s_1\tau_i}^d) \land n_{\tau_j}^i \notin n(s_1: \tau_i) \land \\
&N(G_i^{\text{tFG}}) \cap N(\rho) = \{ n(s_1: \tau_i) \} \land (n_d, s_1: \tau_i) \in (E(\rho) \cap \mathcal{E}^n) \land \\
&\forall n_b, n_{\tau_j}^b ( (n_b, n_{\tau_j}^b) \in (E(\rho) \cap \mathcal{E}^n) \rightarrow s_1: \tau_i \in T_{s_1\tau_i}^\text{notify} ) ) \}
\end{align}
Figure 6: Example Process Graphs

If \( T_{s_1: \tau_i}^{\text{notify}} \) is an empty set then the state \( s_1 \) must be a deadlock state.

Now we define the release edges. An edge \( e_{\tau_i}(s_1, s_2) \in E^r \) iff: (a) \( s_1 \) is a blocking state and in \( s_1 \) thread \( \tau_i \) is in a wait state, and (b) in \( s_2 \), each thread in set \( T_{s_1: \tau_i}^{\text{notify}} \) is in the corresponding post-hold state and the remaining threads are in the same state as that in \( s_1 \).

An execution edge and a notify edge represent the change in the process state during the process execution. A path in the state graph from \( s_1 \) to \( s_n \) containing only execution edges and notify edges represents the fact that the process can reach state \( s_n \) executing from state \( s_1 \). Release edge \( e_{\tau_i}(s_1, s_2) \) represents the change in the state of the process after all threads that are on hold and on which thread \( \tau_i \) is dependent in state \( s_1 \) to receive a notify are released.

Figure 6 illustrates two process graphs. Each process consists of two IR threads \( \tau_1 \) and \( \tau_2 \). Each thread has a wait statement and a notify statement in an unsafe region. Hence, each thread could be in a hold state, a wait state, a notify state, and corresponding post states. We ignore the EoI and post-EoI states since they are of no interest in these examples. The only difference between the two process graphs in Figure 6 is that in Figure 6.(b) thread \( \tau_1 \) contains a loop inside the infinite loop. Figure 7 represents the state graphs corresponding to these process graphs. Set of states in both the state graphs are the same. Each state graph contains four blocking states. Blocking state \( 1w2w \) is a deadlock state. Blocking states \( 1w2h \) and \( 1h2w \) are unsafe blocking states and blocking state \( 1h2h \) is the safe blocking state. The paths in the state graph among the blocking states are highlighted in the figure.

4.1.3 Blocking State Graph

From state graph we derive the Blocking State Graph (BSG). BSG consist of blocking states in the state graph and possible transitions among them. A BSG is represented as
Figure 7: State Graphs Corresponding to the Process Graphs in Figure 6

\[ G^{BSG}(N, E) \], where \( N \) is the set of blocking states and \( E \) is the set of transitions among them. An edge \((s_1, s_2) \in E\) iff there exists a path \( \rho \) from \( s_1 \) to \( s_2 \) in the state graph such that no blocking state is on the path \( \rho \). The edge \((s_1, s_2) \in E\) is labeled with the label of the first edge on the path \( \rho \). The label indicates which threads should be released to make the transition. Since in a state graph multiple paths could start from a blocking state with the same edge, there could be multiple outgoing edges (transitions) from a state in BSG with the same label. Therefore, in the BSG, a transition from a state could be non-deterministic.

A blocking state in the state graph or in the BSG is an unsafe blocking state if a thread is at an unsafe execution point in that state. In a blocking state, a thread could be either at a
Figure 8: Blocking State Graphs Derived from the State Graphs in Figure 7

hold state or at a wait state. A hold state is always a safe state. However, a wait state could be an unsafe state if it is in an unsafe region.

Figure 8 represents the blocking state graphs corresponding to the state graphs illustrated in Figure 7. Since the process is deadlock-free, the process cannot reach state 1w2w. Hence, the transitions leading to the deadlock state 1w2w can be safely ignored. In Figure 8.(a), from each unsafe blocking state it is possible to reach the safe blocking state in the finite steps by following the suitable transitions. However, in Figure 8.(b), because of a cycle between state 1w2h and state 1h2w the process might not reach the safe state in a bounded number of steps. Now we present the sufficient condition for the feasibility of an update.

4.2 Sufficient Condition for Feasibility of an Update

A strongly connected subgraph (SCS) in the BSG of a process may consist of only unsafe states. From the examples in Figure 8, it can be intuitively understood that a process may not reach a safe state in bounded time if such an SCS exists in the BSG. A process can loop in such an SCS for unbounded time without reaching a safe state. However, in some cases, we can force the process out of such an SCS by taking the selected transitions.

Consider an SCS such that a thread τi is at wait in each state in the SCS. Since τi is at wait in each state of the SCS, a transition labeled eτi is present from every state of the SCS. Taking a transition eτi from a state sk mean releasing all threads that are at hold in sk and on which τi depends to receive a notify. If we keep selecting the transition eτi from each state in the SCS then because of bounded wait property, τi will keep receiving the notifications within bounded times. Hence, τi will be executing during the transitions in the SCS. Therefore, as per the bounded loop property, the thread will complete an iteration of the infinite loop. As a result, the thread will move to a hold state in bounded time. A blocking state where τi is in hold state in not in the SCS and hence the process can be forced to move out of the SCS in bounded time. We make the following claim based on this observation.

Claim 1 If a thread τi is at wait in every state in a strongly connected subgraph then by taking the transition eτi from each state in the subgraph the process can be forced to reach a state outside the subgraph in bounded time.
Now consider an SCS $G_i$ which consists of only unsafe states. Moreover, let $G_i$ be an induced subgraph of BSG. Suppose $G_i$ contains an unsafe state $s_i$ such that a thread $\tau_k$ is at wait in $s_i$ but an outgoing transition $e_{\tau_k}$ from $s_i$ is not present in $G_i$. Since $\tau_k$ is at wait in $s_i$, transition $e_{\tau_k}$ from $s_i$ is present in BSG. Hence, by taking the transition $e_{\tau_k}$ from state $s_i$, the process can come out of $G_i$.

Based on above observations we define an unsafe SCS as follows:

**Unsafe SCS:** An SCS of BSG is unsafe if it satisfies the following three conditions.

- Each state in the SCS is unsafe blocking state.
- None of the threads in process is at wait in all states in the SCS.
- The SCS is an induced subgraph of BSG. Also, if the SCS contains a state $s_1$ and if BSG contains an outgoing edge labeled $e_{\tau_i}$ from $s_1$ then the SCS also contains an outgoing edge labeled $e_{\tau_i}$ from $s_1$.

An SCS which is not unsafe is called a safe SCS. If BSG does not contain an unsafe SCS then the process can be forced to reach a safe state in bounded time.

**Condition for Feasibility:** Each SCS of the BSG of the process should be safe.

To prove the sufficiency of the above condition, we first present an algorithm that gives the transition to be taken from each unsafe state in the BSG, provided the condition for feasibility is met. Later, we prove that by following these transitions a process is guaranteed to reach a safe state in bounded time.

Let $G^{\text{UBSG}}$ be a subgraph of BSG induced by the set of unsafe blocking states in the BSG. Tuple $\langle G^{\text{UBSG}}, \text{BSG} \rangle$ is provided as the input to Algorithm 1. Algorithm 1 selects a transition for each state in $G^{\text{UBSG}}$.

Since the feasibility condition is met, graph $G^{\text{UBSG}}$ does not contain an unsafe SCS. The algorithm first identifies all Strongly Connected Components (SCCs), i.e. all maximal strongly connected subgraphs, in the given subgraph of BSG. Since each SCC is a safe SCS but every node in the SCC is unsafe, each SCC satisfies at least one of the following conditions.

C1 : A thread in the process is at wait in all states in the SCC.

C2 : The SCC contains a state $s_1$ such that in the BSG, there exists an outgoing edge labeled $e_{\tau_i}$ from $s_1$, but the SCC does not contain an outgoing edge labeled $e_{\tau_i}$ from $s_1$.

If C1 holds then the algorithm selects the transition $e_{\tau_i}$ for each state in the SCC. If C1 is false then C2 is true. If C2 holds, the algorithm selects the transition $e_{\tau_i}$ for the state $s_1$ satisfying C2. For the rest of the states in the SCC, the algorithm makes a recursive call by passing a subgraph of the SCC generated by removing state $s_1$ from the SCC. Now we prove that the process can reach a safe blocking state in bounded time by taking these transitions.

---

1 A subgraph $H$ of a graph $G$ is said to be an induced subgraph, if for any pair of vertices $s_1$ and $s_2$ of $H$, $(s_1, s_2)$ is an edge of $H$ iff $(s_1, s_2)$ is an edge of $G$. 

---
Algorithm 1 ComputeUpdateSchedule (Graph $G$, Graph $BSG$)

$G = \text{set of strongly connected components in } G$

for all $G_i$ in $G$ do

$T^w = \text{set of threads which are at wait in every state in } G_i$

if ($T^w \neq \emptyset$) then

$\tau_i = \text{a thread in } T^w$

select transition $e_{\tau_i}$ for each state in $G_i$

else

$G'_i = G_i$

for all $s_k$ in $N(G_i)$ do

$T'_k = \{ \tau_j | \exists s_l (e_{\tau_j}(s_k, s_l) \in E(BSG)) \}$

$T''_k = \{ \tau_j | \tau_j \in T'_k \land \nexists s_l (e_{\tau_j}(s_k, s_l) \in N(G_i)) \}$ {set of threads for which there is no outgoing transition from $s_k$ in $G_i$}

if ($T''_k \neq \emptyset$) then

$\tau_i = \text{a thread in } T''_k$

select $e_{\tau_i}$ as transition from $s_k$

$G''_i = \text{subgraph of } G_i \text{ after removing state } s_k \text{ from it}$

break

end if

end for

if $G''_i \neq G_i$ then

ComputeUpdateSchedule ($G'_i$)

else

Error(“Graph contains an unsafe SCS”)

exit

end if

end if

end for

Theorem 3 If the BSG of a process does not contain an unsafe SCS then the process can reach a safe state in bounded time from any unsafe state by using the transitions selected by Algorithm 1.

Instead of proving the above theorem we prove the following theorem which is essentially the same.

Theorem 4 Let a BSG does not contain an unsafe SCS. Let $G^{UBSG}$ be a subgraph of BSG induced by any set of unsafe states in the BSG. From any state in $G^{UBSG}$, the process can reach a state outside $G^{UBSG}$ in bounded time using the transitions selected by Algorithm 1.

Proof: We present the proof by induction on the order of $G^{UBSG}$. Let $G_i$ be a strongly connected component of $G^{UBSG}$. Each state in $G_i$ is unsafe but $G_i$ is a safe SCS, hence $G_i$ satisfies at least one of the following conditions.

C1 : A thread in the process is at wait in every state in $G_i$.

C2 : $G_i$ contains a state $s_j$ such that in the BSG there exists an outgoing edge from $s_j$ labeled $e_{\tau_i}$, but $G_i$ does not contain an outgoing edge from $s_j$ labeled $e_{\tau_i}$.
**Base Case:** $G^\text{UBSG}$ consists of only one state $s_1$.

$G_i$ contains of only one state $s_1$. Since $s_1$ is an unsafe blocking state, $G_i$ satisfies C1. When C1 is true the algorithm selects the transition corresponding to a thread which is at wait in $s_1$. Hence, as per Claim 1, the process will leave state $s_1$ in bounded time.

**Hypothesis:** Let this theorem be true if order of $G^\text{UBSG}$ is less than $n$.

**Inductive step:** Now we prove that the theorem holds if $G^\text{UBSG}$ is of order $n$.

**Lemma 1** Let $G_i$ be any strongly connected component of $G^\text{UBSG}$. Process will leave $G_i$ in bounded time.

**Proof:** $G_i$ will satisfy at least one of the conditions C1 and C2.

Case 1: Suppose $G_i$ satisfies C1. Since $G_i$ satisfies C1, for each state in $G_i$, the algorithm will select the transition corresponding to the thread which is at wait in every state in $G_i$. As per Claim 1 the process will leave $G_i$ in bounded time.

Case 2: Suppose $G_i$ satisfies C2. Let $s_j$ be a state in $G_i$ such that in the BSG there exists an outgoing edge from $s_j$ labeled $e_{\tau_i}$, but $G_i$ does not contain an outgoing edge from $s_j$ labeled $e_{\tau_i}$. The algorithm will select the transition $\tau_k$ from state $s_j$. Hence, from $s_j$ the process can leave $G_i$ in bounded time. Let $G'_i$ be a subgraph of $G_i$ derived by removing node $s_j$ from $G_i$. The order of $G'_i$ is less than $n$ and hence according to our hypothesis the process will leave $G'_i$ in bounded time. After leaving $G'_i$ the process can either enter state $s_j$ or a state outside $G_i$. In either case the process will leave $G_i$ in bounded time.

As per the above lemma, the process will leave every strongly connected component of $G^\text{UBSG}$ in bounded time. There cannot be a cycle in $G^\text{UBSG}$ that spans more than one strongly connected component. Therefore, the process will leave $G^\text{UBSG}$ in bounded time.

Theorem 3 implies that if the BSG of a process does not contain an unsafe SCS then the update is feasible. While defining the BSG in Subsection 4.1, we have considered the processes with only IR threads. In the following subsection, we present the construction of the BSGs for processes containing both IR threads and finite threads. The condition for feasibility of an update, presented in this subsection, also holds for the BSGs defined in the following subsection.

### 4.3 Process with Infinitely Running and Finite Threads

In this subsection, we present the construction of the process graph and the state graph for a process containing both the IR threads and the finite threads. BSG can be generated from the state graph as explained in Subsection 4.1.

For simplicity we assume that each finite thread in the process is started by an IR thread. We do not consider nested invocation of finite threads, i.e., a finite thread starting another finite thread. We also assume that in an infinite loop there is a single `start()` instruction, and an IR thread starts a single finite thread in an iteration of the infinite loop. However, the `start()` call could be polymorphic. The approach proposed in this section can be extended to relax these assumptions.
### 4.3.1 Process Graph

To construct the process graph, we need to know the number of threads in the process and their thread classes. As discussed in Subsection 4.1, we assume that the IR threads in the process are known and they are fixed. The finite threads in the process can start and terminate during the update. Therefore, in general, statically we cannot determine the number of finite threads which will be running in the process during the update. Hence, during the update, we force that at a time at the most one child thread of an IR thread runs in the process. To achieve this, we schedule the process threads as follows.

As discussed in Section 2, we assume that the process considered for an online patch satisfies the independent loop property. That is, even if an IR thread $\tau_i$ is kept on hold at the end of an infinite loop iteration the threads started during the iteration and in the previous iterations are guaranteed to terminate. Let $\tau_i$ be an IR thread in the process such that $\tau_i$ can start a finite thread. The update procedure holds thread $\tau_i$ at the EoI state, i.e. end of infinite loop iteration, and releases it only after the termination of every child thread of $\tau_i$. In the subsequent iterations, till the update completes, we continue holding $\tau_i$ at the EoI state till its child thread terminates. This way, we ensure that there is at the most one live child thread of each IR thread in the process. Thus, while constructing the process graph, it is sufficient to consider one child thread of each IR thread.

Let $T_{ir} = \{\tau_1, \ldots, \tau_n\}$ be the set of IR threads in a process. Further, let $T_{irP} \subset T_{ir}$ be the set of IR threads that can start a finite thread during an iteration. In the process graph, we consider the IR threads in set $T_{ir}$ and one child thread $\tau_j$ for each thread $\tau_i$ in $T_{irP}$. The construction of the process graph is same as that in Subsection 4.1 except the construction of tIFG of finite threads like $\tau_j$. Construction of the tIFG of a finite thread is presented below.

Let $C_{child}^{\tau_i}$ be the set of thread classes of finite threads that can be started on the executions of $\text{start()}$ statement in IR thread $\tau_i$. The tIFG of child thread $\tau_j$ of $\tau_i$ is generated by combing the IFGs of thread classes in set $C_{child}^{\tau_i}$. Let $G_{child}^{\tau_i}$ be the union of IFGs of all thread classes in set $C_{child}^{\tau_i}$.

$$G_{child}^{\tau_i} = \bigcup_{C \in C_{child}^{\tau_i}} G_{C}^{\text{IFG}}$$

Let $G_{\tau_j}$ be a graph isomorphic to $G_{child}^{\tau_i}$ where each node $n_{m:C}^{\tau_i}$ in $N(G_{\tau_j})$ is mapped to the node $n_{m:C}^{\tau_i}$ in $N(G_{child}^{\tau_i})$. The tIFG of thread $\tau_j$ is represented as $G_{\tau_j}^{\text{tIFG}}$ and the nodes and edges in the tIFG are defined as follows.

- $N(G_{\tau_j}^{\text{tIFG}}) = N(G_{\tau_j}) \cup \{n_{s:j}, n_{e:j}\}$, where $n_{s:j}$ is the start node and $n_{e:j}$ is the exit node of the tIFG.
- $E(G_{\tau_j}^{\text{tIFG}}) = E(G_{\tau_j}) \cup E_{\text{start}}^{\tau_j} \cup E_{\text{exit}}^{\tau_j}$, where $E_{\text{start}}^{\tau_j}$ is the set of edges from the start node $n_{s:j}$ to all nodes $n_{s:j}^{\text{start}}$ in $N(G_{\tau_j}^{\text{tIFG}})$ and $E_{\text{exit}}^{\tau_j}$ is the set of edges from all nodes $n_{e:j}^{\text{start}}$ in $N(G_{\tau_j}^{\text{tIFG}})$ to the exit node $n_{e:j}$.

### 4.3.2 State Graph

As discussed in Subsection 4.1, only selected states of a thread are used in the construction of a state graph. While constructing the state graph for a process with finite threads, we use...
the following additional states along with the thread states discussed in Subsection 4.1.

- **Spawn state**: A spawn state corresponds to an execution point immediately before a `start()` statement. We use the spawn states to model the start of new finite threads.

- **Post-spawn state**: A post-spawn state represents an execution point immediately after a `start()` statement.

- **Start state**: Start state of a finite thread corresponds to the execution point immediately before the start node $n_{s,j}$ in the tIFG of the finite thread. Start state, along with spawn state, is used to model the start of a new finite thread.

- **Terminated state**: Terminated state of a finite thread corresponds to the execution point immediately after the exit node $n_{e:j}$ in the tIFG of a finite thread. Termination state is used to represent absence of a finite thread. Along with the wait and the hold, the terminate state is also considered as a blocking state for a thread.

To summarize, for an IR thread we consider hold, wait, notify, spawn, and corresponding post states. Moreover, each IR thread has a special hold state called EoI state. EoI state is always a safe state. For a finite thread we consider start, terminate, hold, wait, notify, post-hold, post-wait, and post-notify states. Similar to EoI state, a terminated state exists for each finite thread and it is always a safe state. Moreover, EoI states and terminate states are always the reachable states of the threads.

Set of states in a state graph is defined by the cross product of states of all threads in the process. Edges in the state graph are also computed as discussed in Subsection 4.1 with two modifications. First, the state graph contains additional start edges $E_{s}$ which model the transitions in the process states due to start of new finite threads in the process. Second, the definition of release edges $E_{r}$ is slightly modified to incorporate the restriction that an IR thread at the EoI state will be released only if its child thread is at the terminate state. We use the notation $\text{child}(\tau_i)$ to denote the child thread of IR thread $\tau_i$.

- **Start edge** $e_{\tau_i}(s_1, s_2)$: Start edge $e_{\tau_i}$ represents the start of a new finite thread by IR thread $\tau_i$. An edge $e_{\tau_i}(s_1, s_2) \in E_{s}$ iff: (a) $\tau_i$ is at spawn state in $s_1$ and at post-spawn state in $s_2$, (b) $\text{child}(\tau_i)$ is at terminated state in $s_1$ and at start state in $s_2$, and (c) in $s_2$, every other thread is in the same state as that in $s_1$.

- **Release edge** $e_{\tau_j}(s_1, s_2)$: Suppose thread $\tau_i$, which is at wait in state $s_1$, depends on IR thread $\tau_j$ or its child thread to receive a notify. Moreover, let $\tau_j$ be at EoI and its child thread is at terminate state in $s_1$. Equation 3, defines the set of all IR thread like $\tau_j$ in state $s_1$ as $T_{\text{notify}}^{\tau_i;\tau_j}$. The definition of the set is similar to the definition of set $T_{\text{notify}}^{\tau_i;\tau_j}$ presented in Equation 2. For uniformity, we use notation $I_{\tau_j}^{EoI}$ to represent the set of EoI state of IR thread $\tau_j$ and notation $I_{\tau_k}^{t}$ to represent the set of terminate
state of finite thread $\tau_k$. Note that, set $I_{\tau_j}^{eoi}$ is a subset of set $I_{\tau_j}^h$,
\[ T_{s_1:s_2}^{\text{notify}} = \{ \tau_j \mid s_1 \vdash \tau_j \in I_{\tau_j}^{eoi} \land s_1 \vdash \text{child}(\tau_j) \in I_{\text{child}(\tau_j)}^{t} \land \exists n_a, n_d, \rho(n_a \in (N(G_{\text{IFG}}) \cup N(G_{\text{IFG}}^{\text{child}}(\tau_j)))) \land 
N(G_{\text{IFG}}) \cap N(\rho) = \{ n(s_1 : \tau_i) \} \land (n_d, s_1 : \tau_i) \in (E(\rho) \cap E^n) \land 
\forall n_b, n_c^{\text{cl}} ( (n_b, n_c^{\text{cl}}) \in (E(\rho) \cap E^n) \rightarrow s_1 : \tau_l \in T_{\tau_l}^{r} ) \} \]

Now we redefine the release edges. An edge $e_{\tau_i}(s_1, s_2) \in E^r$ iff the following conditions hold.

(a) $s_1$ is blocking state and in $s_1$ thread $\tau_i$ is at wait state
(b) In $s_2$, all threads in set $T_{s_1:s_2}^{\text{notify}}$ except those IR threads which can start a finite thread, and all IR threads in set $T_{s_1:s_2}^{\text{notify}}$ are in corresponding post-hold or post-EoI states.
(c) In $s_2$, the remaining threads are in the same states as that in $s_1$.

A BSG can be derived from a state graph as discussed in the Subsection 4.1. The sufficient condition for feasibility of update presented in the Subsection 4.2 also holds for a process with both the IR and finite threads. The proof for the sufficient condition is similar to the proof presented in Subsection 4.2.

### 4.4 Schedule to Take the Process to a Safe State

Now we present the steps to force a process to reach a safe state. These steps are derived from the approach used to identify the sufficient condition.

#### 4.4.1 Step1: take the process to a state were each IR thread has at most one child thread

During the update, first add the blocking update calls at the EoI states in all IR threads. On executing an update call temporarily suspend the IR thread. Resumed the thread after all child threads of the IR thread terminate. On executing the update call at the end of next iteration of the infinite loop again suspended the IR thread till its child thread, if any, terminates. Continue in this manner till the process reaches a state where each IR thread in the process has at most one live child thread.

#### 4.4.2 Step2: take the process to a blocking state

After reaching a state where each IR thread has at most one live child thread, insert additional update calls at the hold states. If a new finite thread is starts before the process reaches a safe blocking state then added the update calls at the hold states in the new thread. When all threads in the process get suspended, either at the wait or at a blocking update call, the process will enter a blocking state in the BSG.
4.4.3 Step3: take the process to a safe blocking state

If the process is in an unsafe blocking state then release the threads from the hold states according to the transaction selected by Algorithm 1. Continue till the process reaches a safe blocking state.

5 Complexity Analysis and Optimization

Now we discuss the computational complexity of the proposed solution to check the condition for feasibility of an update and to compute the update schedule. Let \( t \) be the number of IR threads in the process and let \( n \) be the maximum number of states of a thread considered in the state graph. The maximum number of threads in the process graph, including both the IR threads and the finite threads, can be at most \( 2t \). Hence, the maximum number of states in the state graph can be \( n^{2t} \). This implies the sizes of the state graph and the blocking state graph are exponential in the number of IR threads in the process. Moreover, the number of states of a thread \( n \) can also be exponential to the number of methods in the program because the number of execution points in an IFG can be exponential to the number of methods. Therefore, the computational complexity to check the feasibility of an update and to compute the update schedule is also exponential in the number of IR threads in the process and the number of methods in the program. We now present some optimizations to address the issue of complexity.

5.1 Number of IR Threads

While constructing a blocking state graph we need not consider all IR threads in the process at the same time. We can divide the set of threads in the process into subsets. The only restriction is all threads involved in an interthread dependency cycle cannot span multiple subsets. Let \( G(N, E) \) be a directed graph where \( N \) represents the set of IR threads is the process and \( E \) represents the interthread dependencies between them. An edge \((\tau_i, \tau_j) \in E\) if the process graph contains an interthread dependence edge from thread \( \tau_i \) or its child thread to thread \( \tau_j \) or its child thread. In graph \( G \), if a thread \( \tau_i \) is not reachable from thread \( \tau_k \) then execution of thread \( \tau_k \) cannot affect the execution of thread \( \tau_i \) and the child threads of \( \tau_i \).

To check the feasibility of an update, we can separately check the sufficient condition for set of threads in each strongly connected component of graph \( G \). If a blocking state graph, which is constructed from the threads in a strongly connected component of graph \( G \), does not satisfy the condition for feasibility of update then the process cannot satisfy the condition for feasibility. Moreover, if all strongly connected components of graph \( G \) satisfy the condition for feasibility of update then the process satisfies it. A schedule to take a process to a safe blocking state from an unsafe blocking state can be derived from the BSGs of the strongly connected components of graph \( G \).

5.2 Number of States of a Thread

The number of states of a thread considered in the state graph depends on number of execution points corresponding to the \texttt{wait} and \texttt{notify} statements and unsafe regions in the program. The number of states of a thread can be reduced by constructing an approximate
state graph. In an IFG, a program point \( p_i \) is reachable from a program point \( p_j \) if there exists a pair of execution point \( ep_i \) and \( ep_j \) such that \( PP(ep_i) = p_i, PP(ep_j) = p_j \), and \( ep_i \) is reachable from \( ep_j \). Hence, the states of a thread representing to the execution points corresponding to a wait statement can be represented by a single state corresponding to the program point immediately before that wait statement. That is, set of executions points having same program point can be represented by a single state. If a state graph generated using these approximate states satisfies the condition for feasibility then the state graph discussed in Section 4 will also satisfy the condition for feasibility.

While constructing this approximate state graph the number of states of thread considered, i.e. \( n \), will be equal to the number of wait and notify statements and number of unsafe regions in the program. With these optimization, the complexity of our approach becomes polynomial in the size of the program and exponential in the number of threads in any largest strongly connected component in graph \( G \). In many real-world applications, the maximum number of IR threads which are circularly dependent on each other is very small (i.e 3 or 4) and hence the proposed approach can be used for practical applications.

6 Related Work

To avoid the errors due to online update different techniques have been proposed [3, 4, 6, 7, 9, 11]. Some techniques [3] use versioning approach where multiple versions of an entity (type/class/function) can exist in the process at the same time. Duggan [3] discusses an approach to avoid the type errors in presence of multiple versions of a type in the process. However, to ensure a correct update, only avoiding the type errors is not sufficient [2, 9, 11, 12].

Most of the online update techniques [4, 9, 11, 12, 15] use mutation approach where only one version of an entity can exist in the process at a point in time. The mutation based approaches restrict the process states where an entity can be replaced. Existing update solutions either depend on the user to specify the safe states [6, 9] or identify the safe states by analyzing the patch [11]. A few solutions, like [12], use a mix approach where user specifies some safe states and the update solution identifies other safe states that are equivalent to the user specified states.

To ensure the process is in a safe state before the update starts, majority of the online update solutions for multithreaded processes use the time-out approach [9, 12, 15]. A serious limitation of the time-out approach is the user may not be sure whether the patch can be applied till the user applies the patch. Moreover, a process can take a long time to reach a safe state.

In our earlier work [11] we have presented an approach which externally synchronizes the process threads and force the process to reach a safe state. We termed a thread affected thread if the interleaving of update with the thread can lead to an error. In [11], we considered only those patches that affect single IR thread and its descendant finite threads. As per the independent loop property discussed is Section 2, all descendant threads of the IR thread are guaranteed to terminate if the IR thread is held at the last program point in the infinite loop. Hence if the process contains the single affected IR thread then holding that thread at the end of the infinite loop is sufficient to take the process to a safe state. However, the same approach cannot be used in the case where multiple IR threads are affected by a patch, which this paper addresses.
Mutation based online update solutions for distributed applications raise a challenge of ensuring that all the nodes of the application are in the safe states before the update starts. To reach a safe state, some update solutions for distributed applications take the help of application [7]. That is, the applications must be aware of online update and designed to cooperate with update manager to reach a safe state. Other update solutions avoids the need of synchronizing the states of different nodes [1, 2]. The synchronization requirement is relaxed either at the cost of restricting the changes allowed in a patch [2] or by leaving it to the users to handle the errors that can arise due to online update [1].

When compared with existing solutions [2, 7, 11], our solution places fewer restrictions on the application and on the changes allowed in a patch and yet removes the uncertainty about the feasibility of applying the patch.

7 Conclusions

In order to avoid the unexpected and erroneous behavior while applying a patch to a running process, the update is interleaved with process execution at specific state called safe state. In this report, we have presented a sufficient condition to check the reachability of a safe state for online update. We have proposed an approach that ensures that a process reaches a safe state for online update in bounded time if it satisfies the sufficient condition. Our approach forces a process to reach a safe state by scheduling the threads in the process in a specific order. However, our current solution does not consider the possibility of a livelock that can arise due to insertion of update calls. In future we will extend our solution to avoid the same.

The work presented in this paper removes the uncertainty about the feasibility of applying a patch to a multithreaded process. As a result, our solution makes online update a more reliable software update solution for the users.

References


