Safety Analysis for Dynamic Update of Object Oriented Programs

Yogesh Murarka* Umesh Bellur** Rushikesh K. Joshi*

*Department of Computer Science and Engineering  
**Kanwal Rekhi School of Information Technology  
Indian Institute of Technology Bombay, Mumbai - 400076, India  
Email: {yogeshm@cse,umesh@it,rkj@cse}.iitb.ac.in

Abstract

Maintenance downtime and overheads for applying patches are major concerns for systems requiring round the clock availability. Hence, methods for carrying out dynamic updates are needed. However, correctness of the system during and after every dynamic update needs to be ensured. This paper defines two safety criteria, type consistency and isolation of process execution for dynamic software update. Updates involving one or more insertions of new classes, removals of old classes and replacements of old classes are considered. The condition for producing a type safe update schedule is defined. The parts of the program whose executions have to be isolated from process update are annotated by the user. Conditions are also provided for ensuring isolation of the update process from execution of annotated parts of the program.

1. Introduction

To address changing business requirements software needs to evolve. Applying a patch to a system normally requires shutting down the corresponding process and restarting it. For applications requiring round the clock availability, maintenance downtime is an important concern. Administrative overhead of maintaining the computational systems is another important issue. Scheduling server downtime and updating it has administration overhead. Upgrading a running system, which will mostly be automatic, will reduce the administration overhead.

While updating the program of a process at run time, two major issues need to be addressed.

1. Careful updation of the process such that no erroneous system behavior is observed either temporarily or permanently.

2. The delay introduced in process execution due to dynamic code updation should be minimized.

Different kinds of errors, such as, type errors and deadlocks can occur if system is not updated properly. If an update makes the system behave in an erroneous or unacceptable manner, temporarily or permanently, then the update is unsafe. Different approaches for ensuring type safety during update for procedural and object oriented programs have been presented [4, 9]. However, ensuring type safety alone is not sufficient. Other approaches [1, 2, 3, 5] ensures stronger safety criteria than type safety, but they either restrict the changes allowed in the patch or depends on some domain specific properties which limits their usability.

Our approach for the dynamic updation of concurrent object oriented programs addresses both the issues of safety and delay. This paper defines two safety criteria for dynamic updates. Firstly, no type error should occur. A type error is said to have occurred when a method invoked on an object is not recognized by the object, or the type of the object is not as expected by the caller. Secondly, part of the process execution should be isolated from update process. The paper presents an update methodology which ensures type safety, isolation of process update from execution, and tries to minimize the delay. The methodology proposes constraints over the update timing of classes and objects of the process to ensure safe update. A procedure to determine constraints to avoid type errors is also presented. The implementation of the proposed methodology is in progress.

Rest of the paper is organized as follows. A model of concurrent process and process update is described in Section 2. Formal definition of safety criteria for the update model is given in Section 3. The constraints over update timing of the classes and objects to ensure safe update are discussed in Section 4 and Section 5. Related work is discussed and compared in Section 6. Section 7 concludes and outlines the future work.

2. Process and Process Update

This section discusses the reference process model and explains an approach of replacing the program of a process.
2.1. Process Model

A program \( \Pi \) is defined as a set of classes. A class \( C \) is defined as a set of methods and fields. Fields are encapsulated by a class and can not be accessed outside the class. We use capital letters \( A, B, C, \ldots \) to represent classes and small letters \( u, v, x, \ldots \) to represent methods. A method \( u \) of a class \( C \) is represented as \( C : u \).

The computation defined by a method is represented as a Control Flow Graph (CFG). A node in a CFG represents an instruction statement \( st \). The CFG of a method \( C : u \) starts with an enter statement \( enter_u \) and ends with an exit statement \( exit_u \).

Information about control flow between methods is represented by a Call Graph (CG). If a statement of method \( C : u \) invokes a method \( D : v \), an edge is present in the CG from \( C : u \) to \( D : v \). A \( uses_{class} \) relationship is defined over a program. A class \( C \) uses class \( D \) if an edge is present in the CG from a method in \( C \) to a method in \( D \). A method \( C : u \) can transitively invoke method \( E : w \) at run time if there exists a path from \( C : u \) to \( E : w \) in the CG of the program.

A process \( \mathcal{P} \) is a program in execution. The state of a process \( \mathcal{P} \) at time \( t_j \) is modeled as \( \mathcal{P}_j = (\Pi_j, T_j) \), where \( \Pi_j \) is a program and \( T_j \) is a set of executing threads in the process at time \( t_j \). All threads of the process are assumed to be executing on a single processor in interleaved fashion. The set of all times is totally ordered.

Let \( \mathcal{T} = \{ \tau_1, \tau_2, \ldots \} \) represent a set of all threads of process \( \mathcal{P} \). In the program of a multithreaded process some classes are marked as \( Thread \). A new executing thread \( \tau_i \) is generated when a class marked as \( Thread \) is instantiated. State of a thread \( \tau_i \) at time \( t_j \) is modeled as \( \tau_i^j = (\Pi_i^j, S_i^j) \). Let the thread \( \tau_i \) be generated by instantiating class \( T \) in thread \( \tau_p \) at time \( t_n \). Set \( \Pi_i^0_j \) contains classes of all methods reachable from methods of class \( T \) from \( \Pi^0 \). It is important to note that the initial set \( \Pi_i^0 \) of \( \tau_i \) will be subset of \( \Pi_i^0 \). \( S_i^j \) is the method execution stack of the thread \( \tau_i \) at time \( t_j \). Stack \( S_i^j = \{ C : u_1 \ldots D : v_i \ldots E : y_1 \} \) is a set of active methods. An active method \( D : v_i \) is the \( i^th \) execution instance of the method \( D : v \).

Process Execution

Execution of a process is defined in term of thread execution. The sequence of instructions executed from beginning of the process till generation of a thread is termed as the execution context of the thread. The sequence of instructions executed by a thread is termed as the execution path of the thread. Execution context of a thread followed by execution path together defines the thread execution line.

When method \( C : u \) is executed, all instructions executed from statement \( enter_u \) to statement \( exit_u \) within a thread are considered as part of method execution. Execution of all threads generated by execution of a method are also considered as part of the method execution.

Execution of a statement in execution path defines an execution step of the thread. An execution step is represented as \( E \). An execution step modifies the stack of the process, but does not affect the program. All execution steps of a thread \( \tau_i \) from state \( (\Pi_i^j, S_i^j) \) to state \( (\Pi_i^k, S_i^k) \) are represented as \( (\Pi_i^j, S_i^j) \xrightarrow{E} (\Pi_i^k, S_i^k) \). As the program does not change during execution, here \( \Pi_i^j = \Pi_i^k \).

2.2. Process Update

A process update occurs when program \( \Pi \) of a process \( \mathcal{P} \) is modified. An old program \( \Pi \) of a process and it’s new program \( \Pi' \) after process update can have following similarities and differences.

- Common class: A class present in \( \Pi \) is present in \( \Pi' \). Some classes are carried forward without any change.
- Deletion: A class present in \( \Pi \) is not present in \( \Pi' \). Such a class is known as a deleted class. A deleted class is removed during the process update. The operation of removing a class \( D \) from the set of classes of a thread \( \tau_i \) is represented as \( rm(D, \tau_i) \).
- Addition: A class present in \( \Pi' \) is not present in \( \Pi \). Such a class is known as an added class. An added class is inserted during the process update. The operation of inserting a class \( A \) in the set of classes of a thread \( \tau_i \) is represented as \( ins(A, \tau_i) \).
- Replacement: A class \( C \) present in \( \Pi \) is replaced by another class in program \( \Pi' \). The new class is modification of the old class being replaced. The possible modifications are additions and removals of fields and methods. Notationally, we use symbol \( C' \) to represent the new modified class of an old class \( C \). Classes \( C \) and \( C' \) are termed as replaced class and replacing class respectively. A replaced class is removed and a replacing class is inserted during the process update. The relation between replaced and replacing classes is termed as replaces.

A \( version_m \) relation is defined between the methods of class \( C \) and class \( C' \) to capture replaceability between them. A method in class \( C \) can be related with only one method in \( C' \). For simplicity of understanding, we use the notation \( C : u \) and \( C' : u \) to represent a method in class \( C \) and its related method in class \( C' \). A statement \( st \) can invoke a method \( C' : u \) if \( st \) invokes \( C : u \) in program \( \Pi \). Similarly, \( st \) can invoke a method \( C' : u \) if \( st \) invokes the method \( C' : u \) in program \( \Pi' \). That is, if \( C : u \) invokes \( D : v \) in program \( \Pi \) and method \( C : u \) and method \( D' : v \) are present.
in some program $\Pi^k$, then $C : u$ can invoke $D' : v$ in the program $\Pi^k$.

2.3. Process Update Methodology

Process update is specified in terms of update points and update sets.

- **Update Point**: A well defined point in the process execution where program of the process can be updated.
- **Update Set**: The set of classes and objects which have to be updated at an update point.

In a multithreaded process, an update point is defined as a collection of thread update points. Thread update point is a well defined point in execution of a thread where an object or class can be updated. An update point is reached during process execution when all the threads reach the corresponding thread update points.

At run time when an update point is reached, classes and objects in the update set are updated. In multithreaded process, an object is either thread local or shared among multiple threads. In case of thread local objects, an object is updated as soon as the thread update point in the corresponding thread is reached. However, for a shared object, thread update points of all the threads sharing the object have to be reached. Therefore, to update a shared object, synchronization among threads sharing the object is needed.

To update the program of a process, classes used by all the executing threads need to be updated. The series of update points defines an update schedule for a process. Update schedule to replace an old set of classes $\Pi_i$ by a new set of classes $\Pi'_i$ in a thread $\tau_i$ is defined as follows.

$$
\ldots \xrightarrow{E^*} \langle \Pi'_i, S'_i \rangle \xrightarrow{(u_{p1} , u_{s1})} \langle \Pi'_i, S'_i \rangle \xrightarrow{E^*} \langle \Pi'_i, S'_i \rangle 
$$

where, $u_{s1}$ is the set of classes which have to be updated in $\Pi'_i$ to get $\Pi^k_i$, and $u_{p1}$ is the corresponding thread update point. When $\tau_i$ reaches $u_{p1}$, it is modified from state $\langle \Pi'_i, S'_i \rangle$ to $\langle \Pi^k_i, S^k_i \rangle$. During the update, the set of classes is modified but the stack remains as it is. State of the stack is preserved by preventing update of all classes whose methods are on the stack. After update, thread $\tau_i$ resumes its execution in state $\langle \Pi^k_i, S^k_i \rangle$ and continues executing till it reaches $u_{p2}$ at which point the process is again updated.

Complete update procedure consists of a series of updates at the end of which $\Pi_i$ becomes subset of $\Pi'$. The program of the process $\Pi^k$ consist of classes from all the threads. A program of the process which is neither $\Pi$ nor $\Pi'$ is known as intermediate program.

Class Versioning as Update Policy

Various class update policies have been discussed by Duggan [4] and Xue Li [7]. In class mutation update policy, all the instances of a class are updated together. Hence, to update a class, thread update points for all the instances of the class in all executing threads have to be reached. Update needs to wait till this rendezvous point is reached by all threads. Synchronizing the thread update points delays process execution. Hence to avoid this delay we use the class versioning update policy allowing different threads to use either the old replaced class or the new replacing class for their instances. In this method, a class needs to be updated in all the threads. A class is updated thread by thread. However, all instances within a specific thread belong to the either old replaced class or the new replacing class.

An improper process update can lead to different errors in a system. The safety criteria for process updates are discussed in the next section.

3. Safety Criteria

Dynamic update can make process execution inconsistent, which can lead to different errors in the system. Two consistency criteria, type consistency and isolation of process update from process execution are discussed in this section. It is assumed that both, the old program $\Pi$ and the new program $\Pi'$ are type consistent. If a process update makes the process inconsistent, the update is unsafe.

3.1. Type Consistency

In Object Oriented languages like Java and C++, a class defines a type. Therefore, modifying a class may change the type defined by it. Rules for type safety have to be defined for a dynamic update. For example, when a new method is added to a class, the type defined by the class will change. The issue is, whether to generate a type error or not if an object accesses an object of modified type assuming it to be of old type?

An intermediate program $\Pi^k_i$ is type inconsistent if a set of classes $\Pi^k_i$ of some thread $\tau_i$ is type inconsistent. Set $\Pi^k_i$ is type inconsistent if the following condition holds.

**Type Inconsistency**: There exists a class $C$ in $\Pi^k_i$ such that $C$ accesses a method $D : v$ in program $\Pi$, and neither $D : v$ nor $D' : v$ are in the set $\Pi^k_i$. Or there exists a class $E$ in $\Pi^k_i$ such that $E$ accesses a method $C' : u$ in program $\Pi'$, and neither $C' : u$ nor $C : u$ are in $\Pi^k_i$.

The above condition says that, for an intermediate program to be type consistent two conditions have to be satisfied. Firstly, if a set $\Pi^k_i$ contains a class $C$ from $\Pi$, all classes accessed by $C$ or their replacing classes should be present in $\Pi^k_i$. And the original or replacing class which is
present in $\Pi^C_{\ell}$ should contain a method or its $version_m$ related method invoked by a method of class $C$. Secondly, if the set $\Pi^C_{\ell}$ contains a class $E$ from $\Pi'$, all classes accessed by $E$ or their replaced classes should be present in $\Pi^C_{\ell}$. And the original or replaced class which is present in the set $\Pi^C_{\ell}$ should contain a method or its $version_m$ related method invoked by a method of class $E$.

If any intermediate program has type inconsistency, the update schedule is type unsafe. For a program in a statically typed language, if an update schedule is type safe, all method invocation calls made during execution are guaranteed to get resolved. Our approach to prevent type inconsistency during a process update is discussed in Section 4.

### 3.2. Isolation of process update

Ensuring type safety will guarantee correct execution of individual methods. However, the overall system outcome might become unacceptable due to dynamic update. For example, consider a bill generation system which uses Pricing, Tax, and Billing classes to generate a bill. Billing class gets the price and the tax for an item from Pricing class and Tax class respectively, and calculates the final amount at the end. If the Tax class is updated during a bill generation, the bill may have different taxes applied on items of same category. Such a bill might not be acceptable.

This inconsistency can be avoided by isolating execution of method $genBill()$ from the process update. A method whose execution should be isolated from the process update is called as encapsulated-method. The isolation of a process update is considered violated if following condition holds.

**Violation of Isolation:** The set of classes used during execution of an encapsulated-method is neither subset of $\Pi$ nor subset of $\Pi'$.

The above condition says that all classes used during an execution of an encapsulated-method should belong either to $\Pi$ or $\Pi'$. The class of a method is by default considered to be used during execution of the method.

The set of encapsulated-methods in $\Pi$ and $\Pi'$ is input provided by the user. Declaring a method as encapsulated-method is similar to defining a transaction boundary, whose execution should be isolated from the process update.

The constraints over thread update points to enforce isolation of a process update from execution of encapsulated-methods are presented in Section 5.

### 4. Update Dependencies for Type Safe Update

This section discusses ordering constraints on thread update points of the classes in terms of update dependency. An update schedule is guaranteed to be type safe if the update schedule satisfies these constraints.

**Update dependency** is a relation over the set consisting of classes to be updated. If there exists an update dependency from a class $C$ to a class $D$, $C$ must be updated after $D$. Update dependency arises due to the $uses_{class}$ relation between the classes. Various scenarios for the $uses_{class}$ relation between classes to be updated and the corresponding update dependencies are depicted in Figure 1.

Assume each class from $\Pi$ and $\Pi'$ that needs to be updated has a replacing or replaced class. If a class $D$ is a deleted class, assume a replacing dummy empty class $\phi$ is present in $\Pi'$. Similarly, assume a replaced dummy empty class $\phi$ is present in $\Pi$ for added class $C'$.

![Figure 1. Update dependency](image)

Replacing a class $C$ by $C'$ can affect class $D$ if $D uses_{class} C$. For each pair of classes $C$ and $D$, such that class $D$ uses the class $C$, a diagonal edge from $D$ to $C'$ is present in Figure 1. A diagonal edge shows compatibility between a class in $\Pi$ and a class in $\Pi'$. According to type
inconsistency described in Section 3, a \textit{uses\_class} relation from \( D \) to \( C' \) can introduce type inconsistency if \( C' \) does not contain a \textit{version\_m} related method of a method in class \( C \) which is invoked by a method in class \( D \). In such case, the edge from \( D \) to \( C' \) is labeled as \textit{can not use}, otherwise the edge is labeled as \textit{can use}.

Figure 1(2) illustrates a case where classes \( C \) and \( D \) have to be replaced by \( C' \) and \( D' \) respectively. And class \( D \) uses class \( C \) but can not use \( C' \). Therefore, replacing \( C \) by \( C' \) in presence of \( D \) will create a type inconsistency. To avoid type inconsistency, class \( D \) should be updated before updating class \( C \). Hence, an update dependency from class \( C \) to class \( D \) exists. In Figure 1(7) class \( C' \) is an added class and class \( D' \) uses the class \( C' \). In this case, if \( D \) is replaced before inserting class \( C' \), type inconsistency will arise. Therefore an update dependency from \( D \) to \( C' \) exists.

From the Figure 1, it can be observed that every \textit{can not use} edge leads to an update dependency. In complex cases, where more than one \textit{uses\_class} edges are considered, a circular update dependency might exist among classes as shown in Figure 2. In the case of circular update dependency all the classes involved have to be updated together.

\begin{figure}[h]
\centering
\includegraphics[width=0.5\textwidth]{figure2.png}
\caption{Circular update dependency}
\end{figure}

Type inconsistency can be avoided by obeying update dependencies amongst classes in an update schedule. Following is a procedure to determine update dependencies.

\section*{Update Dependency Graph}

Update dependencies among all classes in \( \Pi \cup \Pi' \) are captured in terms of a directed graph termed as Update Dependency Graph (UDG). To generate the UDG an Intermediate Graph (IG) is created. Nodes of graph IG represent the classes. All classes in \( \Pi \) and \( \Pi' \), except \textit{carried forward} classes, are included in the IG. An edge \((C, D)\) is included in the IG for each \textit{uses\_class} relation present in \( \Pi \) and \( \Pi' \). For all nodes \( C \) and \( C' \), for each edge \((D, C)\) added in the IG an edge \((D', C')\) is also added. Similarly for each edge \((D', C')\) added in the IG an edge \((D', C)\) is also added.

Each edge in the IG representing \textit{uses\_class} relation neither present in \( \Pi \) nor in \( \Pi' \) is labeled as \textit{can use} and \textit{can not use} depending on whether the relationship will introduce a type inconsistency or not.

The UDG is generated from an IG. A \textit{replacing} class and the corresponding \textit{replaced} classes both are represented by a single node in the UDG. If a \textit{can not use} edge \((D, C')\) is present in the IG, an edge \((C', D)\) is added in the UDG. And if an edge \((D', C)\) is present in the IG, an edge \((D', C')\) is added in the UDG.

UDG is a directed graph which may contain cycles. Acyclic directed Update Dependency Graph (AUDG) is generated from UDG by merging strongly connected components into single nodes. A node in AUDG represents a set of classes which have to be updated together. Updating the nodes from AUDG in reverse topological order will preserve type consistency.

Consider the example of Billing system discussed in Section 3. The old program and new program are represented in Figure 3(a) and Figure 3(b) respectively. In the new program, a class CardReader’ has been added. Other classes Tax, Billing, Payment, and MoneyTx are modified. And classes Price and Cashier remain unchanged. Figure 3(c) shows the intermediate graph IG. Assume some new methods have been added to class Tax’ which are used by Billing’. Hence the \textit{uses\_class} relation from Billing’ to Tax will lead to type inconsistency. Also assume methods from class MoneyTx’ invoked by class CardReader’ are not in MoneyTx. Hence the edge from CardReader’ to MoneyTx is of type \textit{can not use}. Consider labels of other edges as shown in Figure 3(c). The UDG is given in Figure 3(d). As UDG contains a cycle, the nodes forming the cycle are merged into a single node to generate AUDG as shown in Figure 3(e). One possible safe update order derived from AUDG is \{MoneyTx, MoneyTx’\}, \{Tax, Tax’\}, \{CardReader’\}, \{Billing, Billing’, Payment, Payment’\}.

\section*{5. Isolation of process update from encapsulated-method execution}

Isolating a process update from the execution of \textit{encapsulated-methods} is similar to ensuring isolation between database transactions. As per isolation criteria defined in Section 3, set of classes used during execution of an encapsulated-method should be subset of \( \Pi \) or \( \Pi' \). An encapsulated-method can be from (1) \textit{replacing} or \textit{added} class, (2) \textit{replaced} or \textit{deleted} class, or (3) \textit{carried forward} class. This section presents the three conditions over update points of classes to ensure isolated execution of encapsulated-methods with respect to these three cases.

We use \( \text{beginThread}_{\tau_i} \) and \( \text{endThread}_{\tau_i} \) to represent the beginning and the end of execution path in execution line of the thread \( \tau_i \). The part of the execution line of a thread from beginning till \( \text{beginThread}_{\tau_i} \) represents the execution context for the thread. The execution line of the thread ends at \( \text{endThread}_{\tau_i} \).
result in invocation of a method from an old version of some other class. It can be noted that this condition covers all child threads which may be created by the execution of the method. The subcases (without and with child threads), which are covered below are also illustrated in Figure 4(1) and Figure 4(2) respectively.

CM1 In the execution path of any thread \( \tau_i \), let \( ins(A', \tau_i) \) be an operation at thread update point w.r.t. an added or a replacing class \( A' \). Class \( A' \) contains an encapsulated-method \( A' : u \). Let \( DC_u \) be the set of replaced classes such that each of their corresponding replacing classes have at least a method reachable from \( A' : u \) in the CG of \( \Pi' \). For every class \( D \) in set \( DC_u \), and for every method \( D : v \) such that \( D' : v \) is reachable from \( A' : u \) in the CG of program \( \Pi' \), the following conditions should hold.

1. After \( ins(A', \tau_i) \), if a statement \( enter_u \) is present in execution path of the thread \( \tau_i \), statement \( enter_v \) should not be present till the corresponding \( exit_u \) or \( rm(D, \tau_i) \). (Note that after \( rm(D, \tau_i) \) an occurrence of \( enter_v \) binds to \( D' : v \)).

2. After \( ins(A', \tau_i) \) if a statement \( enter_u \) is present in the execution context of a child thread \( \tau_j \) without the corresponding statement \( exit_u \), statement \( enter_v \) should not be present in execution path of thread \( \tau_j \) till \( rm(D, \tau_j) \) or till the end of the thread.

Case 2: Encapsulated-method belongs to replaced or deleted class

For an encapsulated-method \( D : v \) from a deleted or replaced class, all classes used during its execution should be from the program \( \Pi' \). It means that no method from an added or replacing class should be transitively invoked dur-

---

**Figure 3.** (a) Old program, (b) New program, (c) Intermediate graph, (d) Update dependency graph, (e) Acyclic update dependency graph

**Case 1: Encapsulated-method belongs to replacing or added class**

For an encapsulated method \( A' : u \) from an added or replacing class \( A' \), all classes used during its execution should be from program \( \Pi' \). That is, \( A' : u \) should not transitively invoke a method from a deleted or replaced class. This requirement is ensured by the following condition CM1.

For this case, it is sufficient to consider only the set of replaced classes and not the deleted classes. This is because, for \( A' : u \) to transitively invoke a method from a deleted or replaced class, it should first transitively invoke a method, say \( D : v \), from a replaced class. It can be noted that method \( A' : u \) can transitively invoke method \( D : v \) only if there exists a method \( D' : v \) which is reachable from \( A' : u \) in the CG of program \( \Pi' \).

Intuitively, the condition CM1 says that after inserting a new class in a thread, its method invocation should not re-
ing an execution of $D : v$. This requirement is ensured by the following condition CM2.

Similar to Case 1, in this case it is sufficient to consider only the set of replacing classes and not the added classes. This is because, for $D : v$ to transitively invoke a method from added or replacing class, it has to first transitively invoke a method, say $A' : u$, from a replacing class. It can be noted that method $D : v$ can transitively invoke method $A' : u$ only if there exists a method $A : u$ which is reachable from $D : v$ in the CG of program II.

Intuitively, the condition CM2 says that a method invocation of a class that is going to be newly inserted. It can be noted that this condition prevents invocation of a method form an inserted class in all child threads created by the execution of the old method. The subcases of condition CM2 (without and with child threads) are covered below and are also illustrated in Figure 5(1) and Figure 5(2) respectively.

CM2 For a class $D$ containing an encapsulated-method $D : v$, let $AC_v$ be the set of replacing classes such that each of their corresponding replaced classes have at least a method reachable from $D : v$ in the CG of II. For every class $A'$ in the set $AC_v$, and for every method $A' : u$ such that its corresponding method $A : u$ is reachable from $D : v$ in the CG of program II, the following conditions should hold.

1. After $\text{ins}(A', \tau_i)$ occurs in the execution path of a thread $\tau_i$, statement $\text{enter}_u$ should not be present between $\text{enter}_v$ and corresponding $\text{exit}_v$ inside the execution path of the thread $\tau_i$.

2. In the execution context of a child thread $\tau_j$ if a statement $\text{enter}_u$ occurs without its corresponding statement $\text{exit}_v$, statement $\text{enter}_u$ should not be present in the execution path of the thread $\tau_j$ after insertion of $A'$ in the execution line of thread $\tau_j$. (Note that insertion of class $A'$ can occur in the execution context or in the execution path of thread $\tau_j$.)

Case 3: Encapsulated-method belongs to a carried forward class

For a method $E : w$ from a carried forward class, all classes used during an activation $E : w_k$ of $E : w$ should be only from II or only from II'. Let $DC_w$ be set of replaced classes as discussed in Case 1, and $AC_w$ be set of replacing classes as discussed in Case 2. Condition CM3 ensures that if $E : w_k$ has transitively invoked a method of a class in $DC_w$, $E : w_k$ will not transitively invoke a method of a class in $AC_w$, and vice-versa. Figure 6 illustrates condition CM3.

6. Related Work

Styole et. al. [9] and Duggan [4] have presented two different approaches to ensure type safety during dynamic update. They both consider a system is type safe when no code manipulates an instance of new type assuming it to be of old type. This condition for type safety is stronger than condition defined in Subsection 3.1. Styole et. al. have developed a method for generating update schedule for procedural systems. They use type mutation approach which is same as the class mutation. In their approach, once an instance is updated, the change can not be reverted back. They find an update point in sequential process where all changes can be applied at once. The implementation of this method for C language can be found in [8].
Duggan ensures type safety during update for object oriented systems. Duggan uses type versioning approach. In this model, the type of an object can be converted from old to new, and vice versa. Functions used for type conversion are termed as version adapters. When an object is accessed, its expected type is determined from the context, and a version adapter is applied if required. The type of an object keeps on changing as and when required, to avoid type errors. Duggan’s approach makes avoiding inconsistencies like isolation of process update from process execution difficult. A comparison of above two approaches is presented by Hicks and Nettles [6].

Gupta [5] and Bloom and Day [2] guarantee more stronger safety criteria than type safety, but at the cost of restricting the changes allowed. Gupta [5] discusses dynamic update in procedural, object oriented and distributed system model. The safety is ensured by guaranteeing the state of the process after the code update is a reachable state of the new program. Bloom and Day [2] and Ajmani et. al. [1] address safety issue for dynamic update of distributed systems. In their models, a computing node can be updated independently from other computing nodes. To ensure the safety, Bloom and Day allow only changes that are compatible with old versions. Ajmani et. al. allow incompatible changes in new versions. However, an exception is thrown if a caller invokes a method on a node assuming it to be of a version which is not supported by the node.

Dynamic schema evolution in databases [7] deals with modifying schema of a database at run time. Boyapati et. al. [3] ensures isolation of transaction from class updates in OODB. They guarantee that a transaction either completely executes the old code or the new code. When a transaction executing the new code encounters an object of old type, the object is updated to the new type. And when a transaction executing the old code encounters an object of new type, the transaction is aborted. Therefore, execution of individual transactions are always safe during the update process. However, the reversible transactional setting used during class update in OODB is not available in all programs. Also, the delay introduced by this approach is manageable only if the lives of the transactions are few milliseconds.

7. Conclusion and Future Work

Updating the program of a process can result in inconsistent process execution, which could lead to one or more errors. The correctness of the system has to be guaranteed during dynamic software updates. Two safety criteria type consistency and isolation of process update are defined for dynamic update of object oriented programs. Type inconsistency can be avoided by restricting the order in which classes are updated. Process update can be isolated from executions of encapsulated-methods by following the isolation constraints over thread update points.

Development of algorithm to derive an update schedule that satisfies constraints AUDG, CM1, CM2, and CM3 and an implementation of the work are under progress.

Acknowledgment

We would like to thank Hrishikesh Karmarkar, Aditya Kanade, Padmaja Joshi, Shruti Mahambre, and anonymous APSEC reviewers for their useful comments.

References