An Object Proxy-Based Dynamic Layer Replacement to Protect IoMT Applications

Bo Han, Zhao Yin-Liang, and Zhu Chang-Peng

School of Electronic and Information Engineering, Xi’an Jiaotong University, Xi’an 710049, China
School of Liangjiang Artificial Intelligence, Chongqing University of Technology, Chongqing 400054, China

Correspondence should be addressed to Bo Han; bohan@xjtu.edu.cn

Received 24 May 2019; Accepted 26 August 2019; Published 30 September 2019

Copyright © 2019 Bo Han et al. This is an open access article distributed under the Creative Commons Attribution License, which permits unrestricted use, distribution, and reproduction in any medium, provided the original work is properly cited.

The Internet of medical things (IoMT) has become a promising paradigm, where the invaluable additional data can be collected by the ordinary medical devices when connecting to the Internet. The deep understanding of symptoms and trends can be provided to patients to manage their lives and treatments. However, due to the diversity of medical devices in IoMT, the codes of healthcare applications may be manipulated and tangled by malicious devices. In addition, the linguistic structures for layer activation in languages cause controls of layer activation to be part of program’s business logic, which hinders the dynamic replacement of layers. Therefore, to solve the above critical problems in IoMT, in this paper, a new approach is firstly proposed to support the dynamic replacement of layer in IoMT applications by incorporating object proxy into virtual machine (VM). Secondly, the heap and address are used to model the object and object evolution to guarantee the feasibility of the approach. After that, we analyze the influences of field access and method invocation and evaluate the risk and safety of the application when these constraints are satisfied. Finally, we conduct the evaluations by extending Java VM to validate the effectiveness of the proposal.

1. Introduction

Internet of medical things (IoMT), which plays an important role in healthcare industry to construct the safety and healthy system in human society [1], is becoming the critical part in reducing healthcare costs, providing timely medical responses, and increasing the quality of medical treatment. IoMT, in fact, is an ecosystem of connected different kinds of sensors, including the wearable devices, medical devices, and clinical systems. Each part in the ecosystem has its own particular context, which is different from others. On one hand, various contexts should be managed by the healthcare application to provide context-based services. For example, to avoid patients’ information leakage, a function component of a healthcare application is used to describe the medical histories of patients to doctor in detail, instead of collecting the sensitive information. On the other hand, the component can be applied in smart watches to regulate the patients’ personal information. However, due to crosscut, the codes are easily scattered and tangled by context, which increase the costs of healthcare applications’ development and maintenance.

Recently, some researchers have introduced the technologies from academic and industry to improve the effectiveness and performance of IoMT applications. For example, the dynamic layered-recurrent neural network was developed by Turabieh et al. [2] to recover missing data. Data mashup-based web technology was proposed by Elmisery et al. [3] to serve a centralized environmental monitoring service.

In addition, the programming languages are compiler-based, which means that all layers and their activation/deactivation are determined in a development stage. The contexts cannot be completely predicted at the development stage [4], healthcare applications, which limit the applications of language. Therefore, in this paper, we introduce an encapsulation mechanism-based context-orient programming paradigm (COP) in IoMT to reduce the occurrences of above-mentioned crosscuts. In COP, the application’s codes are divided into different parts (i.e.,
layers), which is a new block-structured construction. Each layer is related to other contexts, where the layer activation and deactivation mechanism is provided to ensure that the codes in different layers can be executed. As a result, the application can adjust its behaviors according to context. Some extensions of Java for COP, such as JCOP [5] and EventCJ [6], have been implemented.

Virtual machine (VM), which has been considered as an intermediate layer between the executing application and the hardware [7], is widely used in big data process and analysis [8, 9], as well as increasing the possibility for dynamic code evolution. Object proxy [10] is advocated as a flexible mechanism to dynamically adapt behavior of objects to adapt the behavior of a single object. Therefore, this paper proposes a new approach, which incorporates object proxy into VM, to enhance the flexibility of applications in terms of context-based dynamical behaviour adjustment. Unlike above-mentioned COP languages, based on VM, layer activation and deactivation in our approach are used to monitor the changes of context and dynamically activate or deactivate layers so that the application can adapt its behavior in context.

In the perspective of VM, a layer is a Java class. In order to support layer activation and deactivation mechanism on VM level, the object proxy is used in our approach. An object at runtime is actually an object list, which is composed of an object of a base class and objects of layer classes. The base class object is considered as a proxy object, where each method invocation is redirected to other objects in the object list. As a result, codes in the layer are executed to achieve the layer activation. Similarly, the layer deactivation is also achieved by removing or deactivating other objects in the list.

Furthermore, dynamic layer replacement can be achieved in our approach via object evolution. As an object is represented as a list, when VM monitors the changes of context, VM can evolve an object at runtime by inserting a new object of a layer class into it or deleting an object from it. Once the layer is replaced, VM removes the objects of layered classes in the removed layer from object lists and inserts new objects of layered class in the new loaded layered classes in the new layer into object lists. As a result, the application can adjust its behavior to fit the changes of contexts through dynamic layer replacement.

On the other hand, the heap and stack need to be scanned by the VM to find out all objects that to be evolved, which leads to a lot of overhead for the object evolution at runtime. To relieve the overhead, the objects are evolved lazily in our approach. Furthermore, object evolution can lead to type violation [11]. Therefore, this paper incorporates heap and address into RFT-FJ [12] to model the objects and evolutions, where the semantics of field access and method invocation on objects are described in detail. Next, the impact of the evolution on type safety is analyzed. Afterwards, the constraints on object evolution are proposed to avoid the type violation. Finally, the object evolution is presented by extending Java VM to verify our application.

The remainder of this paper is structured as follows. In Section 2, we illustrate our approach by an application. In Section 3, the related work is given. In Section 4, our approach is described. In Section 5, we present the implementation and evaluation of our approach. Conclusion is finally presented in Section 6.

2. Related Work

2.1. The Internet of Medical Things. Rodrigues et al. [13] reviewed technological advances made so far which can be used in IoMT and then analyzed the challenges to be overcome. Ullah et al. [14] presented a new semantic model for patients’ e-health based on existing technologies, which are used to deploy IoT in the field of medical and smart health care. Turabieh et al. [2] introduced a dynamic L-RNN from artificial intelligence to recover missing data from IoMT applications to guarantee high quality of services to the end users. Kerenc et al. [15] presented place of radar for assisted living in the context of IoT for health and pointed out how important the context is. Elmisery et al. [3] used data mashup from web technology to serve a centralized environmental monitoring service. Haoyu et al. [16] used cloud computing to provide real-time sleep apnea detection. They both introduced technologies from other research areas to IoMT. Similarly, we introduce COP programming paradigm from programming language area to reduce codes’ scatter and tangle in IoMT applications. Jonyia et al. [1] pointed out some challenges in IoMT, where software implementation is still full of challenges. We believe our work makes an important step toward resolving these challenges.

2.2. Object Evolution. In order to support context awareness, object composition has been used to represent objects with some dynamic behaviors [7, 17].

An object evolution approach is proposed in [7]. The approach is based on composition and delegation while our approach is based on composition and proxy. This is the most important difference between them. Compared with delegation, proxy is easier to achieve semantic analysis and implementation of layer activation and deactivation, although it suffers from a little less flexibility.

In [18], an evolution approach based on version consistency is proposed, which is similar to our evolution approach. However, its version consistency is applied for the evolution of components in distributed system while ours is applied for objects. This is the most important difference between the two approaches in terms of version consistency.

In [17], object composition and delegation with a static type discipline are also used for dynamic behavior adaptation of objects. However, the object in [17] is only explicitly composed while the composition is implicit in our approach. Furthermore, object composition in [17] is used for dynamic inheritance and code reuse while object composition in our approach is used for dynamic behavior adaptation of the program.

Granule-oriented programming [19] uses fitness to determine which components are available for applications. In [12], predicate is applied to determine the applicability of
context-specific methods. Our approach also uses predicate for the similar purpose. However, there are some differences between them. Predicate in [19] is not formally defined while fitness in [12] and our approach is formally defined as a predicate method. Moreover, fitness in [12] is applied to select context-specific methods of composed classes while predicate in our approach is applied to select context-specific methods of composed objects.

In [20], various problems of software evolution and their corresponding solutions are presented and classified into several types. Some required operations on evolution are formally described and analyzed to ensure the consistency before and after evolution.

2.3. Context-Oriented Programming. In [6], an approach that incorporates features of AOP is presented to implement features of COP. Both it and our approach share some common features. Firstly, the predicate is used to determine the applicability of the context-specific method. Secondly, layer activation is on a per-instance basis rather than a per-thread basis. However, our approach is VM-based, while the approach presented in [6] is compiler-based. Although the approach incorporates features of AOP, dynamic aspect weaving is not incorporated. Therefore, the approach does not support dynamic layer addition while our approach supports it. This is a significant difference between the two approaches.

ContextFJ [21] is an extension of Featherweight Java, which models language mechanisms for context-oriented programming. Although ContextFJ and our approach both directly express context-dependent behavior, there are still some differences between them. Firstly, ContextFJ and our approach use block-structured constructs and object proxy for dynamic behavior adaptation, respectively. Secondly, ContextFJ does not allow layers to introduce new methods, but our approach does.

3. System Model

In this section, we first describe problems caused by Java-based COP languages by a simple healthcare application and then propose object evolution approach, a VM-centric approach, to solve them.

A healthcare application code snippet written in a Java-based COP language is shown in Figure 1, which is used to display healthcare schedule information for users. If the current device is a doctor’s computer, the application can show more details about medical histories of their patients and must not show patients’ sensitive personal information, protecting their personal privacy. On the other hand, if running on patients’ smart watches, the application can briefly show important medical schedule, including some personal information.

The basic part of the application is a Java program, and classes in the part are called basic ones, not considering runtime context. The rest are two layers L1 and L2. A layer is defined by the keyword layer, consisting of a predicate, and a Java class, called layered class. The predicate is used to check whether or not the layer matches current execution context. If so, the methods in the layered class can be executed.

In compiler-based COP languages, layer activation and deactivation are part of codes, which means if some layers no longer match the current context, an application has to rewrite, recompile, and re-execute. This not only shortens its life cycle but also makes it not adjust its behaviors to the context well.

For solving such problems, our approach proposes that VM is responsible for layer activation and deactivation. That is, each time methods in basic classes are invoked, VM is responsible for selecting and executing methods in proper layers, which match the current execution context. As a result, each time a layer has bugs or no longer matches new execution context, a new layer can easily be dynamically loaded by VM to replace the old layer.

To achieve our approach, the object structure is extended, as shown in Figure 2. An object at runtime is represented into a double linked list. The first element of the liked list is an object of a based class. The other elements are objects of layer classes. For example, when the layer L1 is activated, a runtime object O is composed of O_b and O_L1, an object of the base class calendar and that of the layered class, respectively. VM redirects all method invocations on O_b to these O_L1 as long as L1 matches the current execution environment. Actually, O_b is considered as an object proxy.

On the other hand, if L1 is replaced with L1’, VM evolves O to O’ by replacing O_L1 with O_L1’, a new object created by VM. Since only O_b points to O_L1, the redirection can avoid scan heap and stack.

Furthermore, if there are many objects containing objects of L1, they must be not immediately evolved because they may be accessed very soon. Therefore, our approach evolves these objects in a lazy manner, that is, an object is
evolved only when it is accessed. To check whether or not an object is evolved, a version number is added to structures of a class and an object. The version numbers of a classes and an object are initialized when they are loaded and created, respectively. Specifically, when a layer is replaced, due to context mismatch, the version number of a base class is increased by 1 only if the same name layered class in the layer is loaded. And the version number of the layered class is initialized with that of the base class. When an object is created, its version number is initialized with that of its class. If version numbers of a class and an object are different, then the object will be evolved when it is accessed, and version numbers of all elements in the object are updated. As a result, our approach may avoid scan heap and stack to evolve objects and improve object evolution efficiency.

Our object evolution approach is summarized as follows:

1. For an object O represented as \( \text{O}_b \) if there is a method invocation on it, the version numbers of \( \text{O}_b \) are firstly compared with those of its base class C.
2. If equal, the object O is not evolved and the invocation will be executed. Specifically, if the field “real_obj” in \( \text{O}_b \) points to null, which means \( \text{O}_{CL} \) is null, the method will be resolved in C and executed. Otherwise, the method will resolve the object \( \text{C}_L \) pointed by the real_obj field in \( \text{O}_b \). As a result, methods which defined layered classes are executed.
3. If not equal, which means at least a layer replacement has occurred, then the object O has to be evolved by VM. Specifically,
   (i) If \( \text{O}_{CL} \) is null, then an object of a layered class named C is firstly created, and composed a double linked list with \( \text{O}_b \). Furthermore, the version number of \( \text{O}_b \) will be updated. Finally, VM executes step 1 again.
   (ii) If \( \text{O}_{CL} \) is not null, then \( \text{O}_{CL} \) will be replaced with a new object \( \text{O}'_{CL} \) of a new layered class C from a new layer. This is achieved by redirecting the real_obj field in \( \text{O}_b \) to the newly created object \( \text{O}'_{CL} \) and the field proxy_obj in \( \text{O}'_{CL} \) to \( \text{O}_b \). Figure 2(b) shows the evolution process. Furthermore, the version number of \( \text{O}_b \) will be updated. Finally, VM executes step 1 again.

At runtime, objects may be replaced, according to Step 3, which results in type violation. In the next section, we model objects and their evolutions and then formally describe field access and method invocation on objects. Afterwards, some constraints are proposed to ensure that object evolution does not cause type violation.

4. Object Evolution

We first formally describe classes, objects and field access, and method invocation on objects. Afterwards, we analyze the impact of objects’ evolutions on them and make some constraints to ensure such evolutions do not result in type violation. In this paper, classes are defined as sets and objects are formally described as a set of addresses in a heap.

**Definition 1 (class definition).** A class \( \text{C} \) can be represented by a set \( \{T_f, T \rightarrow T^m\} \) composed of fields and methods, where \( T_f \) is the abbreviation of \( f_1, f_2, \ldots, f_n \) and \( f \) denote a type and a field name, respectively. \( T \rightarrow T^m \) is similar.

Two auxiliary functions \( S_l \) and \( S_m \) return the field set and the method set of a specific class \( \text{C} \), whose definitions are given as follows:

\[
S_l(C) = \begin{cases} 
\emptyset, & \text{if } C = \emptyset, \\
\{f\}, & \text{if } C = \{T_f, T \rightarrow T^m\}, \\
\emptyset, & \text{if } C = \emptyset, \\
[m], & \text{if } C = \{T_f, T \rightarrow T^m\}. 
\end{cases}
\]

**Definition 2 (combination class).** \( \text{C} : \text{C}_L \) is a combination of two same name classes, where \( \text{C} \) and \( \text{C}_L \) denote a base class \( \text{C} \) and a layered class \( \text{C}_L \), respectively. The combination contains all elements of \( \text{C} \) and \( \text{C}_L \). \( \text{O}_b : \text{O}_L \) is an object of \( \text{C} : \text{C}_L \), which can be described as the two-tuple \( \langle \text{C} : \text{C}_L, \overline{T} : l \rangle \) and \( \overline{T} = S_l(C) \cup S_l(C_L) \).

**Definition 3 (object definition).** An object is represented as a two-tuple \( \langle \text{C} : \overline{T}, \overline{l} \rangle \) or \( \langle \text{C} : \text{C}_L, \overline{T} : l \rangle \). The first element is the type of the object, and the second one is its fields and addresses of these fields, where \( l \) denotes an address.
Based on the above definitions, we formally describe field access and method invocation on objects by a computation state. A computation state can be described as the pair of an expression and heap, \((e, H)\), where an expression may be a field access \(l.f\) or a method invocation \(l.m(l)\). Therefore, the reduction relationship can be described by \(e, H \rightarrow e', H\), which indicates that a computation state \(e\) can be reduced to another state \(e', H\). Furthermore, \(\rightarrow \ast\) denotes the reflexive transitive closure of \(\rightarrow\).

The rule for field access (E-PROJ1 and E-PROJ2): for an object, if its address in the heap \(H\) is \(l\), its type is \(C\) or \(C : : C_L\), and the field \(f\) has been defined, then the field access can be reduced to its value.

The rule for the method invocation (E-LINVK, E-LINVK2, E-LINVK3, and E-CINVK): compared with field access, the method invocations are more complex, since they contain context match and object evolution. Their main ideas are (1) if the receiver of a method invocation is only an object of the base class \(C\), the invocation is directly reduced to its method body, where parameters are replaced by arguments, and this is replaced by the receiver and (2) if the receiver of a method invocation is an object of a combination class \(C : : C_L\), the method invocation will be redirected to the same type method in a layered class \(C_L\). If the layer which contains \(C_L\) matches the current context, then the invocation can be reduced to the method body of the method. Otherwise, the object has to be evolved. Specifically, rules for method invocation are described as follows:

(i) E-INVK rule: for an object \(O_b : : O_{CL}\), its address in the heap \(H\) is \(l\), and its type is a base class \(C\). If the method \(m\) is resolved in \(C\), then the invocation is directly reduced to its method body, where parameters \(x\) are replaced by arguments \(l\), and this is replaced by \(l\).

(ii) E-LINVK rule: for an object \(O_b : : O_{CL}\), its address in the heap \(H\) is \(l\), and its type is a combination class \(C : : C_L\). If a method \(m\) is resolved in \(C_L\) and the layer containing \(C_L\) matches the current context, then the invocation can be reduced to the method body in \(C_L\), where parameters \(x\) are replaced by arguments \(l\), and this is replaced by \(l\).

(iii) E-LINVK2 rule: for an object \(O_b : : O_{CL}\), its address in the heap \(H\) is \(l\), and its type is a combination class \(C : : C_L\). If a method \(m\) is resolved in \(C_L\) but the layer does not match the current and a proper layer can be found out or provided, then the object is firstly evolved by VM, that is, an new object of \(C_L\) is created and replaces the object of \(C_L\) in \(O\), and then the object’s type is also updated to \(C : : C_L\). Afterwards, the invocation is reduced to the method body of the same type method in \(C_L\), where parameters \(x\) are replaced by arguments \(l\), and this is replaced by \(l\).

(iv) E-LINVK3 rule: for an object \(O_b : : O_{CL}\), its address in the heap \(H\) is \(l\), and its type is a combination class \(C : : C_L\). If a method \(m\) is resolved in \(C_L\) but the layer does not match the current and a proper layer can be found out or provided, then the object is firstly evolved by VM, that is, an new object of \(C_L\) is created and replaces the object of \(C_L\) in \(O\), and then the object’s type is also updated to \(C : : C_L\). Afterwards, the invocation is reduced to the method body of the same type method in \(C_L\), where parameters \(x\) are replaced by arguments \(l\), and this is replaced by \(l\).

(v) E-CINVK rule: for an object \(O_b : : O_{CL}\), its address in the heap \(H\) is \(l\), and its type is a combination class \(C : : C_L\). If a method \(m\) is invoked by another method in the same layer class, then the context match is not checked and the invocation is reduced to its body, where parameters \(x\) are replaced by arguments \(l\), and this is replaced by \(l\).

In the above-mentioned rules, the function \(mbtype()\) models the semantic of the method resolution in Java. It returns a method body of a method in a class. More details about it are shown in [7, 12]. "predicate() true IN \(C_L\)" represents the layer \(L\) that matches the current context while "predicate() false IN \(C_L\)" represents the layer \(L\) that does not match.

The notation \(\Longrightarrow\) in the E-LINVK3 rule denotes object evolution. Object evolution may result in the type violation when evolved object is being accessed. For example, if E-LINVK3 is interrupted after it executes for a moment, then E-LINVK3 is executed again and an object \(l\) is evolved. When E-LINVK3 is executed again, the method may not exist, due to \(f\)s change, yielding the type violation. Therefore, some constraints should be proposed for object evolution so that the type violation can be avoided.

Next, we firstly formally describe object evolution and present some constraints on it, which are shown in the following definition.

**Definition 4 (valid object evolution).** Assume that an object \(l\) is evolved to \(l\), denoted by \(l \Longrightarrow l\); since a layered class \(C_L\) is replaced by another well-formed \(C_{L'}\), we have \(H(l) = \langle C :: : C_{L'}, f : : l \rangle\), \(H(l) = \langle C :: : C_{L'}, f : : l \rangle\), \(f = S_l(C) \cap S_l(C_{L'})\). Furthermore, the evolution is valid only if the following conditions hold:

1. \(\forall m m\) is in stack, if \(m \in l\), then \(m \in l\)
2. \(\forall f f \in S_m(C_{L'})\), \(f\) is only accessed by \(m \in S_m(C_{L'})\)
3. \(S_m(C_{L1}) \leq S_m(C_{L2})\), and \(\forall m m \in S_m(C_{L1})\) then \(m\) type \(m, C_{L2} \leq m, C_{L2}\)

The first constraint requires that object evolution cannot have influence on running methods. This ensures these methods still exist after object evolution, avoiding runtime errors. The second constraint ensures fields in a layered class are only accessed by methods in the class, which ensures accessing fields do not lose after object evolution. The function \(mtype\) [7, 12] returns the type of a method in a class. The third constraint requires \(C_{L2}\) to have the same methods as \(C_{L1}\), which ensures there are no dynamic-type violations after objects’ evolution.
Based on the above definition, if base classes and layer classes are well typed, the following theorem can ensure type violation does not occur after object evolution. We denote expressions before and after objection evolution by \( e \) and \( \varepsilon \), respectively.

**Theorem 1.** Assume that \( e \) contains \( l \) and that \( l \implies \varepsilon \) results in \( e \implies \varepsilon \); if \( e \) has the property of type-safe and all layer class \( C_L \) in each replacement are well typed, then \( \varepsilon \) has the property of type-safe too.

The theorem intuitively indicates that if any component of a well-typed expression \( e \) is replaced with another well-typed component, then \( e \) is still well typed. This is very similar to industrial products in the real world. For example, if a component of a qualified car is replaced with another qualified one, then the car is still qualified.

**Proof.** We prove the theorem by induction.

1. **Basic.** If \( l \) is unchanged, then \( e \) is unchanged.
2. **Reduction:** assume that \( e \implies \varepsilon \) results from \( l \implies l' \), which is caused by \( C_{L1} \) replacing \( C_{L1} \). If \( e \) is type-safe, \( \varepsilon \) is type-safe.

   \[
   \text{E-PROJ}_1 \quad l.f, \overline{H} \longrightarrow l', \text{if } \overline{H}(l) = \langle C, \overline{T} : \overline{l} \rangle
   \]

   \[
   \text{E-PROJ}_2 \quad l.f, \overline{H} \longrightarrow l', \text{if } \overline{H}(l) = \langle C :: C_{L1}, \overline{T} : \overline{l} \rangle
   \]

   \[
   \text{E-INV} \quad l.m(\overline{l}), \overline{H} \longrightarrow \langle \text{this} \implies l, \overline{x} \longrightarrow \overline{l} e, \text{if } \overline{H}(l) = \langle C, \overline{T} : \overline{l} \rangle \text{ and } mbody(m, C) = (\overline{x}, e)
   \]

   \[
   \text{E-CINV} \quad l.m(\overline{l}), \overline{H} \longrightarrow \langle \text{this} \implies l, \overline{x} \longrightarrow \overline{l} e, \text{if } \overline{H}(l) = \langle C :: C_{L1}, \overline{T} : \overline{l} \rangle \text{ and } mbody(m, C :: C_{L1}) = (\overline{x}, e)
   \]

   \[
   \text{E-LINV} \quad l.m(\overline{l}), \overline{H} \longrightarrow \langle \text{this} \implies l, \overline{x} \longrightarrow \overline{l} e, \text{if } \overline{H}(l) = \langle C :: C_{L1}, \overline{T} : \overline{l} \rangle, mbody(m, C :: C_{L1}) = (\overline{x}, e), \text{ predicate } (\text{true} \text{ IN } C_{L1})
   \]

   \[
   \text{E-LINV}_2 \quad l.m(\overline{l}), \overline{H} \longrightarrow \langle \text{this} \implies l, \overline{x} \longrightarrow \overline{l} e, \text{if } \overline{H}(l) = \langle C :: C_{L1}, \overline{T} : \overline{l} \rangle, \text{ predicate } (\text{false} \text{ IN } C_{L1}), (\exists C_{L'1} : \text{predicate } (\text{true} \text{ IN } C_{L'1} \land m \in C_{L'1})), mbody(m, C) = (\overline{x}, e)
   \]

   \[
   \text{E-LINV}_3 \quad l.m(\overline{l}), \overline{H} \longrightarrow l.m(\overline{l}), \text{where } \overline{T} = S_l(C) \cup S_l(C_{L1}) \text{ if } \overline{H}(l) = \langle C :: C_{L1}, \overline{T} : \overline{l} \rangle, \text{ predicate } (\text{false} \text{ IN } C_{L1}, \exists C_{L'1} : \text{predicate } (\text{true} \text{ IN } C_{L'1} \land m \in C_{L'1}), \forall f e, (\).
   \]

(ii) If \( l \) contains a dynamic-type violation, then we have \( C :: C_{L1} \notin C :: C_{L1} \). Assume that \( l' \) is type is \( C :: C_{L1} \) and \( l' \)'s type is \( C :: C_{L2} \). According to the third constraint of valid object evolution and the subclass relationship record, we have \( C :: C_{L1} \subset C :: C_{L2} \), which is contradicted with the assumption. So, the dynamic-type violation is nonexistent. The theorem holds.

\( \Box \)
5. Implementation and Evaluation

In this section, we briefly describe how to extend a Java virtual machine to implement runtime object evolution, a VM-centric approach.

5.1. Implementation. JamVM is a compact Java virtual machine (JVM) which conforms to JVM specification version 2. In comparison to most other VMs, JamVM is extremely small; hence, it is easier for us to extend. In order to implement object evolution, we mainly extend JamVM’s class loader, interpreter, resolution, and memory allocation.

JamVM represents objects with the structure object. In order to support object evolution based on object proxy, we added two pointers real_obj and proxt_obj into the structure. The extended structure is shown in Figure 2(a). Each time an object is created, these pointers are initialized to null. At runtime, they are used to create a double-linked object list, a runtime object. Furthermore, a variable of integer type, flag, is added into the structure. If its value is zero, VM does not check whether or not a layer matches the current context. Otherwise, the check is executed.

In our approach, objects are lazily evolved. As mentioned above in Section 3, lazy evolution is implemented by comparisons of version numbers of a class and an object. Therefore, we also added a variable of integer type, version, into the object and classblock which is a structure to represent as loaded classes. Afterwards, we extend the class loader so that version may correctly initialize each time a class is loaded. In general, each time a base class is loaded, its version’s value is initialized with 0. Each time a layered class is loaded or removed due to a layer replacement, its version’s value is increased by 1. On the other hand, each time a layered class is loaded, its version’s value is initialized with that of a base class with the same name. We also extend the new instruction so that when an object is created, its version’s value is initialized with the version’s value of its class.

The interpreter is responsible for dealing with method invocation. The normal execution process of method invocation is illustrated in the left of Figure 3. Each time dealing with an instruction for method invocation, the resolution module firstly resolves the invoked method and then the execution module executes the resolved method. However, if the receiver is an object list, the process should be changed, as shown in Figure 3. Particularly, a compare module and evolution module is added into method invocation process.

We extend semantics, three instructions in the interpreter module of JamVM to achieve the change. The three instructions are invokevirtual, invokespecial, and invokestatic, respectively, which are responsible for doing with method invocations in different conditions.

(i) If proxy_obj and real_obj of the receiver of the invocation are null, the receiver is only an object of a base class. Firstly, version numbers of the object and its class are compared. If equal, the method resolution just looks for the invoked method in the based class and executes its body. This is consistent with the semantic of E-INVK. Otherwise, the object is evolved. If not equal, VM evolves the object. If evolution succeeds, context match is checked. If it matches, then the resolution module looks for its method body in real_obj and executes it. Otherwise, the module looks for its method body in the object and executes it.

(ii) For the receiver, if its proxy_obj is null but its real_obj is not null, which means that the receiver is an object list containing at least two objects, the method invocation occurs on the first object of the list, an object of a base class. In this case, version numbers of the first object in the list and its class need to be compared. If they are not equal, the receiver is evolved by VM. If the evolution succeeds and context match is checked, then the resolution module looks for its method body in real_obj and executes it. Otherwise, the module looks for its method body in the object and executes it.

(iii) For the receiver, if its a proxy_obj, the method invocation occurs on an object of layered class. In this case, version numbers of the first object in the list and its class need to be compared. If they are not equal, the receiver is evolved by VM. If the evolution succeeds, context match is checked, then the resolution module looks for its method body in real_obj and executes it. Otherwise, the module looks for its method body in the object and executes it.

For object evolution by triggering layer replacement, the interpreter searches a specific directory to look for a proper layer which holds the following conditions:

(i) Matches the current context
(ii) Contains a layer class, which has the same type method with the current invoked method

Once VM finds out a proper layer, layered classes in it are loaded and new objects are created to evolve the receiver of the current method invocation.

In this section, our implementation proves that our approach is feasible. In the future work, we plan to support object evolution in a multithread context.

5.2. Evaluation. In this section, we discuss our runtime measurements. For illustration purposes, methods in classes and layers are called as plain methods and layered methods, respectively. We set up micromeasurements to assess the performance of the execution of the layered method. Our measurements were all run on 2.2 GHz Intel Core2Duo with 3 GB of RAM, running Ubuntu 10.04 LTS.

Java Grande benchmark framework [14] contains all kinds of method invocations and object creations and access in Java to evaluate performance of a Java VM, for example, class/instance method invocations and class/instance object creations. To evaluate our approach, we add at least two layers for each kind of method invocations and object creations and access in Java Grande benchmark framework, respectively. One of the two layers fits the current execution
environment while the others do not fit. On the other hand, both JamVM and our extension to JamVM run on Ubuntu 10.04 LTS operation system with Java 1.6 version.

Based on our experiment environment, we first evaluate the performance of the execution of the layered method and then compare the execution performance of method invocations on an extension to JamVM and on standard JamVM. The benchmark includes synchronized and nonsynchronized plain instance methods that calculate the power of a given number. The main method in the benchmark contains four types of plain method calls: synchronized and nonsynchronized instance methods, and calls to synchronized and nonsynchronized class methods. Each type of method call is executed $10^9$ times.

5.2.1. Performance of Layer Replacement Based on Object Evolution. After 17 times execution of benchmark, the results are shown in Figure 4. The results show that the layer replacement efficiency caused by class method invocation is 468/ms. The layer replacement efficiency caused by instance method invocation is 414/ms. The difference between the two efficiencies are caused by the following reasons. Firstly, class method invocation has nothing to do with object evolution; hence, new object creations and object list constitutions are unnecessary. Moreover, compared to the synchronized method, the influence of an asynchronous method on the layer replacement is smaller. The reason is because execution of unlocking the asynchronous method needs more time.

5.2.2. Performance of Method Call Based on Object Evolution. The layer replacement based on object evolution is implemented by the extension of a virtual machine, which could impact the execution performance of method invocations on objects. In order to evaluate the influence, we compare the execution performance of method invocations on the standard JamVM and its extension by running benchmark 17 times on them, respectively.

The experiment results are shown in Figure 5. The results show that the execution performance of method invocation on the extended virtual machine is from 6% to 10% lower than that on the standard JamVM. Particularly, the execution performance of instance method invocations on the extension is close to 90% of that on standard JamVM. The execution performance of synchronization class method invocations on the extension is close to 94% of that on standard JamVM. The main reason for the low performance is that the extension JamVM has to check whether or not the receiver of a method invocation contains objects of layered classes. Furthermore, it needs to check whether or not the invoked method matches the current context.

5.3. Discussion. Our approach supports layer replacement by extending object represented in Java VM and semantics of method resolution, which gives rise to overhead. In space complexity, an object in our approach may cost extra tens of bytes because it contains an object list. In time complexity, as mentioned above in Implementation section, our approach usually does two extra operations. The first one is to locate an object in an object list, and the second one is to execute the fitness method. They are the root cause to overhead, which ranges from 6% to 10% in our experiment. In the future, we intend to introduce a cache mechanism to reduce an object in size and avoid fitness method execution as soon as possible.

6. Conclusion

Healthcare applications have to run on various medical devices, which have their own particular contexts. These
contexts crosscut healthcare applications’ codes, making them more scattered and tangled and increasing costs of their development and maintenance. Some COP languages address crosscut concern; however, they have some deficits that new layers cannot be loaded at runtime to replace layers which no longer match the current context.

In this paper, we propose a new approach to support context-based dynamic behavior adjustment of applications. Unlike existing COP languages, our approach uses object proxy in the VM level to support layer activation and deactivation. Furthermore, as our approach is based on VM, it more easily leverages VM to dynamically replace layers so that applications can better adapt new context. In order to verify our approach, we first formally describe it and prove it, ensuring type-safety properties of applications only if some constraints are held. Finally, we present how to implement our approach by extending Java VM.

Data Availability

The experiment platform (i.e., extended JamVM) and results used to support the findings of this study have been deposited in https://github.com/zcp/jvm_extension.git, and they are available from the corresponding author upon request.

Conflicts of Interest

The authors declare that there are no conflicts of interest regarding the publication of this paper.

Acknowledgments

This research was supported in part by a grant from NSFC under Grant nos. 61571286 and 61640219.

References


