Axioms and Abstract Predicates on Interfaces in Specifying/Verifying OO Components (Technique Report)[⋆]

Hong Ali, Liu Yijing, and Qiu Zongyan

LMAM and Department of Informatics, School of Math., Peking University
{hongali,liuyijing,qzy}@math.pku.edu.cn

Abstract. Abstraction is essential in component-based design and implementation of systems, however, it brings also challenges to the formal specification and verification. In this paper we develop a framework to support the abstract specification for the interfaces of components and their interactions and the related verification. We show also that the abstract specification on the interface-level can be used to enforce the correct implementation of the components. We take one practical application of the well-known MVC architecture as a case study. Although our work focuses on the OO based programs, some concepts and techniques developed in the work might be useful more broadly.

Keywords: Component, Specification, Verification, Abstract Predicate, Axiom, MVC

1 Introduction

Component-based design and composition have been widely respected and used in implementing large-scale software systems. The related methodologies emphasize abstraction, interaction based on clear and abstract interfaces, separation of interfaces from implementations, interchangeability of components, etc. The main ideas of these techniques are information hiding, modularity, insulation, and so on, to support more flexible and robust development and integration of complex systems.

Separation of interfaces from concrete components is one of the most important techniques in component-based system (CBS) development. Interfaces serve as a layer to insulate components and a media to connect them, and provide enough information to the clients. This separation makes twofold benefits: on one hand, clients are designed only based on interfaces of the components which they use, that make them independent of details of the components. On the other hand, the components to be used need only to implement the interfaces that may provide wider design choices.

However, although the interface-based techniques are very useful and effective in supporting good component design and flexible integration, they bring also challenges to formal specification and verification. In common practice, interface declarations provide only syntactic and typing information. For verifying behaviors of systems, we

⋆ Supported by NNSF of China, Grant No 61272160, 61100061, and 61202069.

must include semantic specifications for interfaces. How and in which form the specifications are provided becomes a new problem, due to an obvious quandary: for one thing, we need to protect the abstraction provided by the component interfaces, thus the specification should not leak details of the implementation. For another, we need the specifications to provide enough information for the behaviors of the components, to support the reasoning of their clients. Obviously, if the specification involves real implementation details, it will block the modification and replacement of the components, or ask for re-specifying/re-verifying large portion of the system on account of modification, either in the development or in the maintenance.

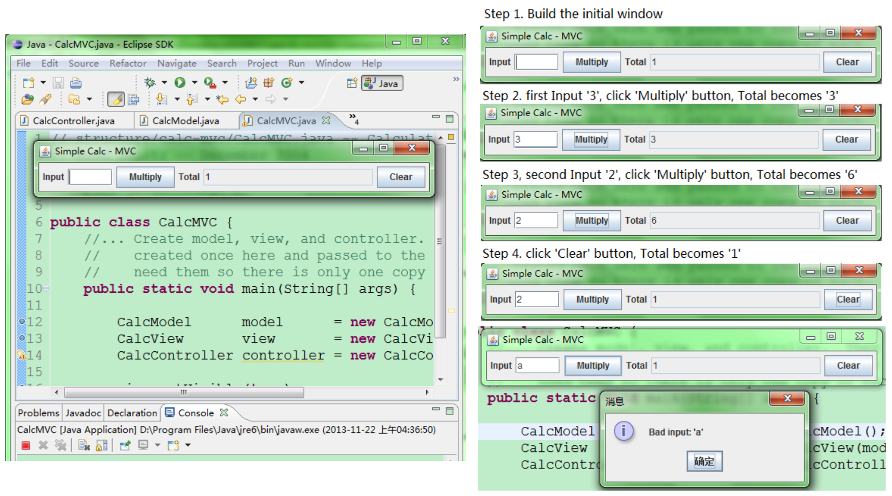
In this paper, we present an approach for specifying a group of co-related OO components abstractly by giving the specifications for their interfaces. The specifications consist of two aspects: a pair of abstract pre/post conditions which is expressed upon abstract predicates (named *specification predicates*) for each method, and a group of *axioms* over the predicates which describes relations over different predicates thus gives constraints on the implementations over methods and classes.

Based on our previous work [22], we build the theoretical foundation and define how a program with these specifications is correct. We give some new rules to form a more complete inference system for reasoning OO programs, then whether a group of classes forms a correct implementation of a group of relative interfaces can be proved. Also, we support the proof for client codes based only on the interfaces. As the proof involves no information from the implementation, modular verification is well achieved. We illustrate our approach by specifying and verifying a simple multiplication calculator designed following the MVC architecture and built upon closely co-related interfaces and classes. Due to the page limit, we leave some details in our report [10].

In the rest of the paper, we will analyze the problems with a MVC calculator in more details in Section 2, and introduce the concept *axioms* and basic languages in Section 3. We build the formal framework in Section 4, and then show the case study in Section 5. At last, we discuss some related work and conclude in Section 6. The basic inference rules of the framework are given in Appendix A, and the detail verification of the rest methods in the implementation is shown in Appendix B.

2 Abstractly Specify/Verify Co-related Components: Problem

To give an intuition for the problem of this study, we use a simple MVC (Model-View-Controller) architecture design for an integer-multiplication calculator displayed in **Fig. 1**. This MVC calculator consists of three components. A model component which is independent of views, encapsulates the application logic (such as multiplying and reset algorithms, and so on) of the calculator. A view component requests the product value from the model and represents it in a certain style. And a controller component listens to user actions on the view and passes it to the model. Here the view is designed as a user interface with two buttons for detecting user requests to the model by the controller, where "Multiply" for multiplication and "Clear" for reset; and two textboxes for representing user-input integer in "Input" and the product value (also as one multiplier for the next multiplying) in "Total".



```
public class CalcMVC {
    //... Create model, view, and controller.
    //    created once here and passed to the
    //    need them so there is only one copy
    public static void main(String[] args) {

        CalcModel      model      = new CalcMo
        CalcView       view       = new CalcVi
        CalcController controller = new CalcCo

```

```
CalcMVC [Java Application] D:\Program Files\Java\jre6\bin\javaw.exe (2013-11-22 上午04:36:50)
```

Step 1. Build the initial window

Step 2. first Input '3', click 'Multiply' button, Total becomes '3'

Step 3, second Input '2', click 'Multiply' button, Total becomes '6'

Step 4. click 'Clear' button, Total becomes '1'

Bad input: 'a'

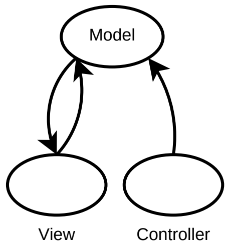
Fig. 1. An Execution of the MVC Multiplication Calculator

We show an execution of this calculator by several steps as in **Fig. 1** (the right side). First, we run the program and get an initial frame with a default value 1 for "Total" and blank "Input". Second, we input an integer 3 for "Input" and click the "Multiply" button to call the model to multiply 3 and 1, and immediately the "Total" value becomes 3. Third, we input another integer 2 and click "Multiply" again, and this time the "Total" becomes 6 (that is $2 \star 3$). Then, we click "Clear" button to reset "Total" to its initial value 1. Noninteger inputs like char 'a' would be catched as exceptions by reporting a message like in the lower right corner of **Fig. 1**.

As the view styles of a model could be various (e.g., text-based, graphical, or web interface) and would not affect the intrinsic properties (e.g., separating model from its views, interactions among three components) of MVC architecture, we simplify the user-interface view in **Fig. 1** and let the views output total values textually in our case. Further, to make this MVC calculator more extensible, modular and reusable, we abstract it with interfaces as depicted in **Fig. 2**. Three interface declarations with method signatures (the right part) are respectively given for the components. For example, MI is the interface of model, having a method $Multiply(i)$ for multiplying input i[1] and a certain integer (e.g., the current value displaying in "Total" textbox when inputting i), a method $Reset()$ for resetting total value, $addView(v)$ adding a view onto the model, $getState()$ returning the current state value, etc. Three components here are closely related: views register on some model, the controller deals with user inputs and asks its model to update the state (i.e., total value), the model then notifies all registered views

[1] Here we assume that, all inputs i and return values which are declared as primitive **Int** in our case are actually of **BigInteger** type in Java. Because as defined in Java Language Specification, semantics of arithmetic operations in **BigInteger** exactly mimic those of Java's integer arithmetic operators. We simply use **Int** type to declare and operate big integers, while the overflow error by multiplying two big integers can be eliminated.

i1) A MVC architecture consists of three components: one model, several views and one controller;
i2) Model embeds application logic and relative data state;
i3) Views can register on a model, flush themselves accordingly, and paint in each of their own way;
i4) Controller deals with user inputs, passes it for updating model that further affects all related views.



```
interface $MI$ {
  void $addView(VI\ v)$;
  Int $getState()$;
  void $Multiply$(Int $i$);
  void $Reset()$; }
interface $VI$ {
  void $update(MI\ m)$;
  void $paint()$; }
interface $CI${
  void $multiplyPerformed()$;
  void $resetPerformed()$; }
```

Fig. 2. MVC Architecture and Component Interfaces for Multiplication Calculator

for its state change, and the views need to get model's state after notified, etc. But, how can we specify these independent of the implementations?

We follow the ideas of abstract predicates [5, 17, 20] and Separation Logic, and introduce predicate $model(m, vs, st)$ to assert that model m has a registered view set vs and a state st, $view(v, m, st)$ to assert view v has registered on m and its state is st, and $controller(c, m, st)$ to assert controller c monitors m and its state is st. We specify the MVC interfaces as given in **Fig. 3** according to the (informal) requirements, where we have some additional predicates which will be explained below. Here we use a brief form "$\langle pre\rangle\langle post\rangle$" after method signatures to represent pre/post conditions instead of more common "**requires** pre; **ensures** $post$;" pairs. In addition, the types for predicate parameters are omitted, which can be added in real implementations.

The specification of $addView$ in MI says that the calling view will be added into the view set of the model, and its state will be updated following the model. The specification of $getState$ means that it simply returns the state of the model. For methods $Multiply$ and $Reset$, things become more complex, because the methods will not only affect the model, but also its related views. We introduce a predicate $MVs(m, vs, st)$ to

```
interface $MI$ {
  void $addView(VI\ v)$
  $\langle model(\mathbf{this}, vs, st) * view(v, \mathbf{this}, -)\rangle$
  $\langle model(\mathbf{this}, vs \cup \{v\}, st) *$
     $view(v, \mathbf{this}, st)\rangle$;
  Int $getState()$ $\langle model(\mathbf{this}, vs, st)\rangle$
  $\langle model(\mathbf{this}, vs, st) \wedge \mathsf{res} = st\rangle$;
  void $Multiply$(Int $i$)
  $\langle MVs(\mathbf{this}, vs, st)\rangle$
  $\langle \exists st' \cdot MVs(\mathbf{this}, vs, st')$
     $\wedge\ st' = product(st, i)\rangle$;
  void $Reset()$ $\langle MVs(\mathbf{this}, vs, st)\rangle$
  $\langle MVs(\mathbf{this}, vs, 1)\rangle$; }
```

```
interface $VI$ {
  void $update(MI\ m)$
  $\langle view(\mathbf{this}, m, -) * model(m, vs, st)\rangle$
  $\langle view(\mathbf{this}, m, st) * model(m, vs, st)\rangle$;
  void $paint()$
  $\langle view(\mathbf{this}, m, st)\rangle\langle view(\mathbf{this}, m, st)\rangle$; }
interface $CI${
  void $multiplyPerformed()$
  $\langle MVC(\mathbf{this}, m, vs, st)\rangle$
  $\langle \exists st' \cdot MVC(\mathbf{this}, m, vs, st')\rangle$;
  void $resetPerformed()$
  $\langle MVC(\mathbf{this}, m, vs, st)\rangle$
  $\langle MVC(\mathbf{this}, m, vs, 1)\rangle$; }
```

Fig. 3. Interfaces with Formal Specifications for MVC Arch.

```
void buildviews (MI m, CI c) ⟨$\exists i \cdot model(m, \emptyset, i) * controller(c, m, i)$⟩
                             ⟨$\exists r_1, r_2 \cdot MVC(c, m, \{r_1, r_2\}, 1)$⟩ {
  VI $v_1$ = new View(m);                          // add a new view to model m
  Int n = 1;
  while (n < 4){ c.multiplyPerformed();  // process a user input for multiply
      n++; }
  VI $v_2$ = new $View_2$(m);                      // add another view to model m
  c.resetPerformed();                     // process another user input for reset
}
```

Fig. 4. A Client Procedure Using the MVC Architecture

assert the state of a bundle of one model and its related views, thus *Multiply* and *Reset* modify this state as desired. Predicate $product(st, i)$ encapsulates the algorithm of multiplying st and i. Method *update* in *VI* is called by a model that will flush the view's state and cause some other related actions (e.g., the view painting by method *paint*). At last we consider *multiplyPerformed* and *resetPerformed* in *CI* which can be called by clients of the MVC components. They brings also problems, because both will affect all components here. To specify states of this bundle of components, we introduce another predicate $MVC(c, m, vs, st)$ to assert that we have a bundle of a controller c, a model m, a set of views vs with internal state st. These specifications go the similar way as shown in literature, e.g. [7], and ours [22].

Having the interface declarations, we can go ahead to define classes to implement them, and then build concrete MVC instance(s), and write client codes to use the implementation. We postpone the implementation for a moment, and first consider some client codes and their verifications. Because interfaces should be fences for hiding implementation details, on the semantic side, we should support verifying client codes without knowing the implementation details of the interfaces.

Fig. 4 gives a client method, where we assume some implementing classes have been built. From its formal parameters, we get a pair of connected model and controller objects, and require the result state satisfies assertion $\exists r_1, r_2 \cdot MVC(c, m, \{r_1, r_2\}, 1)$. *View* and $View_2$ are classes implementing *VI*. For the constructor of *View*, we assume it satisfies a specification $\{model(m, vs, st)\}\, View(m) \{model(m, vs \cup \{\mathbf{this}\}, st) * view(\mathbf{this}, m, st)\}$. Here **this** refers to the new created object which is assigned to variable v by "$v = \mathbf{new}\ \ View(m)$;". It is similar for $View_2$.

For verifying the client method, we list a part of reasoning in **Fig. 5**. The constructions go well, then we meet problems in line (5'). To step into the while loop and verify the first command $c.multiplyPerformed()$ according to the specification of $multiplyPerformed()$ in interface *CI*, we need to check whether the current program state specified in line (5") not only satisfies the loop condition but also assertions like $MVC(\ldots)$ with some parameters. Moreover, a loop invariant containing $MVC(\ldots)$ should be inferred in line (7). However, with the abstract specifications, we cannot deduce out a $MVC(\ldots)$ assertion from an assertion building of predicate symbols $model(\ldots), view(\ldots), controller(\ldots)$. Neither can we know what are the things to

(1) $\{model(m, \emptyset, i) * controller(c, m, i)\}$

(2) $VI\ v_1 = \mathbf{new}\ \ View(m);$

(3) $\{model(m, \{v_1\}, i) * view(v_1, m, i) * controller(c, m, i)\}$

(4) $\mathbf{Int}\ n = 1;$

(5) $\{n = 1 \wedge model(m, \{v_1\}, i) * view(v_1, m, i) * controller(c, m, i)\}$

(5′) $\{\exists i, n_1 \cdot n = 1 \wedge n = n_1 \wedge n_1 < 4 \wedge model(m, \{v_1\}, i) * view(v_1, m, i)*$
$controller(c, m, i)\}$
$\Downarrow$???

(5″) $\{\exists i, n_1 \cdot n = 1 \wedge n = n_1 \wedge n_1 < 4 \wedge \underline{MVC(c,\ \overset{[1]}{\ldots})\ \overset{[2]}{\ldots}}\}$

(6) $\mathbf{while}\ (n < 4)\{$

(7) $\{\exists i', n_1' \cdot n = n_1' \wedge n_1' < 4 \wedge \underline{MVC(c,\ \overset{[1']}{\ldots})\ \overset{[2']}{\ldots}}\}$ [Loop Invariant]

(8) $c.multiplyPerformed();$

......

Fig. 5. Verification of the Client Code

fill segments $\overset{[1]}{\ldots}$, $\overset{[2]}{\ldots}$, $\overset{[1']}{\ldots}$ and $\overset{[2']}{\ldots}$, then we cannot go ahead. This means clearly that something is missed in our specifications.

In reasoning component-based programs, such problems are common. Because we want to have interfaces independent of implementations to support flexible system designs and replaceable components, the specifications on the interface-level can be written only in terms of some abstract symbols, here the predicate names and parameters. Although we have some intentions for each symbol, abstract expressions cannot reveal them in the method specifications. Besides, verifying client codes based on the specifications asks for more information about the components, but we cannot expose the predicate definitions which are closely related to the implementations that have not presented yet or may be multiple ones for a given interface.

To solve the problem here, adding an assertion like "$(model(m, vs, st) * (\circledast_{v \in vs}\ view(v, m, st)) * controller(c, m, st)) \Leftrightarrow MVC(c, m, vs, st)$" to the specifications seems helpful. By applying it on (5') with parameter substitutions $(\{v_1\}, i/vs, st)$, we can easily infer the desired assertion in (5") to be like $\exists i, n_1 \cdot n = 1 \wedge n = n_1 \wedge n_1 < 4 \wedge MVC(c, m, \{v_1\}, i)$ and the loop invariant $\exists i', n_1' \cdot n = n_1' \wedge n_1' < 4 \wedge MVC(c, m, \{v_1\}, i')$ in line (7). Then the following proof can go on. However, to ensure the soundness of our proof, we should prove such added assertion is true with regard to the given implementation before using it in code reasoning. In the following, we will call these assertions *axioms*, and present a framework on how to specify them, how to verify programs with *specification predicates* and axioms, and how to apply them in proving implementations and client codes.

3 Axioms and Languages

In our framework, a specification predicate is abstract on the interface level, which may have a definition(s) in each class that implements the interface. On the other hand, an *axiom* is a logic statement expressed based on constants, logical variables, and predicates combined by logical connections and qualifiers. It gives restrictions and/or relations over abstract predicates. Similar to the situation in the First-Order Logic, a set of axioms defines what is its "model". Here a "model" should be a set of class definitions with specifications, where the relative predicates get their definitions.

As in other logic, to have a model, a set of axioms should be consistent.

Definition 1 (Consistency of Axioms). *Assume $\mathcal{A}_G$ is the set of axioms of a program G. $\mathcal{A}_G$ is consistency,* if $\mathcal{A}_G \not\models \mathbf{false}$.

An inconsistent set of axioms cannot have any implementation. However, due to the incompleteness of the inference system for our logic (similar to the classical Separation Logic), the inconsistency is generally undecidable. Because inferring an axiom may need because some more rules which are lack in the current inference system, or there exists no implementing models for this axiom. We can also define non-redundancy for a set of axioms, however, that is not important and we omit it.

To conduct the design for components, we declare a set of interfaces to outline the system, and specify methods using pre/post conditions based on abstract predicates. Thus we use axioms to restrict/relate the predicates, which put further restrictions on the implementations. How to choose the predicates and axioms is the matter of the designers'thoughts about the requirements. The axioms describe general properties of the later-coming implementations. Another important role of the axioms is to support abstract-level reasoning for client codes. In this aspect, two forms of axioms are most important: implications and equivalences, because they support the *substitution rule* in reasoning.

Now we give a brief introduction to our assertion and programming language with specification annotations of VeriJ.

The assertion language in VeriJ is a Separation Logic (SL) revised to fit the needs of OO programs, as given in **Fig. 6** (upper part). Here we have variables (v), constants, numeric and boolean expressions. ρ denotes pure (heap-free) assertions and η the heap assertions, where r denotes references which serve as logical variables here. We have first-order logic and separation logic connectors, and qualifiers. For example, $\psi_1 * \psi_2$ means the current heap (satisfying $\psi_1 * \psi_2$) can be split into two parts, while one satisfies assertion ψ_1 and another satisfies ψ_2; and $\psi_1 \mathrel{-\!\!*} \psi_2$ means, if any heap satisfying ψ_1 is added to the current heap (satisfying $\psi_1 \mathrel{-\!\!*} \psi_2$), then the whole heap would satisfy ψ_2. $\circledast_i$ is the iterative version of $*$.

Some OO specific assertion forms are included, where $r_1 = r_2$ tells r_1, r_2 are identical; $r : T$ and $r <: T$ assert the object which r refers to is exactly of the type T or a subtype of T; $v = r$ asserts the value of variable or constant v is r; $\mathbf{emp}$ asserts an empty heap; the singleton assertion $r_1.a \mapsto r_2$ means the heap is exactly a field a of an object (referred by r_1) holding value r_2; and $\mathsf{obj}(r, T)$ asserts the heap contains exactly an object of type T and r refers to it. The existence of empty objects in OO prevents us to use $r.a \mapsto -$ or $r.a \hookrightarrow -$ to assert the existence of some objects in the current heap as in typical Separation Logic. Thus we introduce $\mathsf{obj}(r, T)$ to solve this problem. We use over-lined form to represent sequences, as in user-defined predicate $p(\overline{r})$. We may extend it with set or other mathematical notations as needed.

VeriJ is a subset of sequential Java (μJava [23]) with specifications as annotations, especially predicate definitions, method specifications and axioms (lower part of **Fig. 6**).

$$
\begin{array}{ll}
\rho ::= \mathit{bool\_exps} \mid r_1 = r_2 \mid r : T \mid r <: T \mid v = r & \eta ::= \mathbf{emp} \mid r_1.a \mapsto r_2 \mid \mathsf{obj}(r, T) \\
\multicolumn{2}{l}{\psi ::= \rho \mid \eta \mid p(\overline{r}) \mid \neg\psi \mid \psi \vee \psi \mid \psi * \psi \mid \psi \mathrel{-\!\!*} \psi \mid \circledast_i \psi_i \mid \exists r \cdot \psi}
\end{array}
$$

$$
\begin{array}{ll}
\begin{array}{lll}
T & ::= & \mathbf{Bool} \mid \mathbf{Object} \mid \mathbf{Int} \mid C \mid I \\
v & ::= & \mathbf{this} \mid x \\
e & ::= & \mathbf{true} \mid \mathbf{false} \mid \mathbf{null} \mid v \mid \mathit{numeric\_exps} \\
b & ::= & \mathbf{true} \mid \mathbf{false} \mid e < e \mid e = e \mid \neg b \mid b \vee b \\
c & ::= & \mathbf{skip} \mid x := e \mid v.a := e \mid x := v.a \\
 & \mid & x := (C)v \mid x := v.m(\overline{e}) \mid x := \mathbf{new}\, C(\overline{e}) \\
 & \mid & \mathbf{return}\ e \mid c; c \mid \mathbf{if}\, b\, c\, \mathbf{else}\, c \mid \mathbf{while}\, b\, c \\
\pi & ::= & \langle\psi\rangle\langle\psi\rangle \qquad A ::= \mathbf{axiom}\, \psi
\end{array}
&
\begin{array}{lll}
P & ::= & \mathbf{def}\ p(\mathbf{this}, \overline{r}) \\
M & ::= & T\ m(\overline{T\ z}) \\
L & ::= & \mathbf{interface}\ I\ [:\ \overline{I}]\ \{\overline{P;}\ \overline{M\ [\pi];}\} \\
K & ::= & \mathbf{class}\ C : C\ [\rhd\overline{I}]\ \{\ \overline{T\ a;} \\
 & & \quad \overline{P : \psi;}\ \overline{C(\overline{T\ z})\ [\pi]\ \{\overline{T\ y;}\ c\}} \\
 & & \quad \overline{M\ [\pi]\ \{\overline{T\ y;}\ c\}}\ \} \\
G & ::= & \overline{A;}\ \overline{(L \mid K)}\ K
\end{array}
\end{array}
$$

Fig. 6. VeriJ Assertion and Programming Language with Specification Annotations

Here C denotes class name, I for interface name, a and m for field and method names respectively. We omit Java access control issues. Here are some explanations:

- We assume $\mathbf{Bool}, \mathbf{Int}$ as primitive types, which are neither a supertype nor a subtype of any type. Mutation, field accessing, casting, method invocation, and object creation are all taken as assignment commands. Return-value command can just appear as the last command in a method and has a built-in variable res to record the value of e. Sequential, conditional, and loop commands are common as in Java programming language.
- **def** $p(\mathbf{this}, \overline{r})$ declares a specification predicate with signature $p(\mathbf{this}, \overline{r})$, whose body ψ in a class may be directly defined or inherited from the superclass. $\mathbf{this}$ is always explicitly written as the first parameter of predicates to denote the current object. Recursive definitions of predicates are allowed for recursive data structures in programs. As methods, a subclass inherits predicate definitions from its superclass if it does not override them. Similarly, sub-interfaces inherit predicate declarations. In addition, a class must implement each predicate in its implemented interface(s), either by giving directly a definition, or inheriting one from its superclass. Predicates are used in method specifications and axioms to provide data abstraction.
- **axiom** ψ introduces an axiom ψ into the global scope, while all variables are implicitly universal-quantified, and can be instantiated in axiom application. No program variables are allowed in axioms. We will give more details in Section 4.
- π is a pair of specification for constructors or methods in the form of $\langle \mathit{pre} \rangle \langle \mathit{post} \rangle$. Specifications in a supertype can be inherited or overridden in subtypes. When a method does not have an explicit specification, it may inherit several specification pairs from the supertypes of its class. If a non-overriding method is not explicitly specified, the default specification "$\langle \mathbf{true} \rangle \langle \mathbf{true} \rangle$" is assumed. In addition, we assume sub-interface will not redeclare the same methods as in its super-interfaces.
- As in Java, each class has a superclass, possibly $\mathbf{Object}$, but may implement zero or more interfaces. A class can define some specification predicates and if it implements an interface which declares some predicates, it must directly define each predicate with a body or inherit one from its superclass. We assume all methods are public. For simplicity method overloading is omitted here.

– A program G consists of a sequence of axiom definitions, and then a sequence of class and interface declarations, where at least one class presents.

For typing and reasoning a program G, a static environment Γ_G, or simple Γ without ambiguity, is built to record useful information in G. We need also type-checking specification parts in programs, e.g., a predicate definition body involves only its parameters; each predicate in an axiom is an application of some declared predicate in some interfaces; the pre/post conditions of a method are well-formed; predicates declared in an interface must be realized in its implementation classes, etc. For these, we may introduce types into the specifications. The environment construction and type-checking are routine and ignored here. We will only consider well-typed programs below, and assume some components of Γ are usable: $\Theta(C.m)$ fetches the body of method m in class C, $\Pi(T.m)$ gives the method specification(s) of m in type T; $\Phi(C.p(\mathbf{this}, \overline{r}))$ gets the body assertion of p in class C; and $\mathcal{A}$ is the set of all axioms in G.

We assume that in a program, each predicate is uniquely named (this can be achieved by suitable renaming). Thus, if several definitions for one predicate name p appear in different classes, they are local definitions for p fitting the needs of each individual class. We assume all definitions for p have the same signature, that is, no overloading.

4 Verifying Programs wrt Axioms and Method Specifications

In this section, we develop the framework for reasoning about VeriJ programs with specifications, especially interface-based design and axioms. We extend the inference rule set developed in our previous work [18, 22] (Ref. **Appendix A**).

4.1 Verifying Implementations wrt Axioms

Because of the possible existence of subclass overriding and multi-implementing classes for one interface, multiple definitions for the same predicate are common in programs. On the other hand, axioms are global properties/requirements over a program. To judge whether a group of classes obeys a set of axioms, we should define clearly what the predicate applications denote in axioms. We can obtain predicate definitions from Γ. However, as a predicate may have multiple definitions, we must determine which of them is used to unfold a specific predicate application. Thus we define a substitution for a predicate application in a program as follows:

Definition 2 (Predicate Application Substitution). *Suppose p is a specification predicate, and $\{C_j\}_{j=1}^k$ is the set of classes in program G where p is defined. We define the expansion for the application $p(r, \overline{r'})$ in axioms as a substitution:*

$$\delta_{p,\Gamma} \;\widehat{=}\; [\, \textstyle\bigvee_j (r : C_j \wedge \mathsf{fix}(C_j, p(r, \overline{r'}))) \,/\, p(r, \overline{r'}) \,] \tag{1}$$

where:

$$\mathsf{fix}(D, \psi) = \begin{cases} \neg \mathsf{fix}(D, \psi'), & \textit{if } \psi \textit{ is } \neg\psi'; \\ \mathsf{fix}(D, \psi_1) \otimes \mathsf{fix}(D, \psi_2), & \textit{if } \psi \textit{ is } \psi_1 \otimes \psi_2, \textit{ where } \otimes \in \{\vee, *, -\!\!*\}; \\ \exists r \cdot \mathsf{fix}(D, \psi'), & \textit{if } \psi \textit{ is } \exists r \cdot \psi'; \\ D.q(r_0, \overline{r}), & \textit{if } \psi \textit{ is } q(r_0, \overline{r}); \\ \psi, & \text{otherwise.} \end{cases} \tag{2}$$

We substitute application $p(r, \overline{r'})$ in axioms by a disjunction, while each element consists of a type assertion $(r : C_j)$ and a type fixed body generated by fix. Because the body may contain applications of other predicate(s) or recursive application(s) of the same predicate, we need to fix their meaning by type and also avoid infinite expansion. Here fix carries on type D down over the formula. The special $D.q$ form is used to suspend the unfolding thus prevent infinite expansion. We introduce the following rule to enable further unfolding and a new round of the fixing:

$$\frac{\Phi(D.q(\mathbf{this}, \overline{r})) = \psi}{\Gamma \vdash D.q(r_0, \overline{r'}) \Leftrightarrow \mathsf{fix}(D, \psi)[r_0, \overline{r'}/\mathbf{this}, \overline{r}]} \qquad \textsf{[EXPAND]}$$

For a predicate set Ψ, we define the substitution for Ψ based on all the substitutions for $p \in \Psi$. Based on **Def. 2**, we have the following definition to connect axioms with interface and class declarations in a program.

Definition 3 (Well-Supported Axiom). *Suppose N is a sequence of class/interface declarations, and ψ is an axiom mentioning only types and relative predicates defined in N. We say ψ is* well supported *by N, if $\Gamma_N \models \psi\delta_{\mathsf{preds}(\psi),\Gamma_N}$.*

Here Γ_N provides predicate definitions, and $\delta_{\mathsf{preds}(\psi),\Gamma_N}$ is the substitution built from $\mathsf{preds}(\psi)$ (a subset[2] of predicates occurring in ψ,) according to **Definition 2**, and used to obtain the assertion to be validated. Because $\delta_{\mathsf{preds}(\psi),\Gamma_N}$ is completely determined by Γ_N, we will write the fact simply as $\Gamma_N \models \psi$. Generally, for a set of axioms $\mathcal{A}$, we say $\mathcal{A}$ is *well supported* by N and write $\Gamma_N \models \mathcal{A}$, if $\Gamma_N \models \psi$ for every $\psi \in \mathcal{A}$.

Now we define whether a program G with its axiom set $\mathcal{A}$ is *well-axiom-constrained*:

Definition 4 (Well-Axiom-Constrained Program). *A program $G = (\overline{A}; N)$ is a well-axiom-constrained program, if $\Gamma_N \models \mathcal{A}$, where $\mathcal{A}$ is built from $\overline{A}$.*

Note that both well-supported and well-axiom-constrained are semantic concepts. As a version of separation logic, we have given a set of inference rules for our logic and proven its soundness result in [18], which contains the basic inference rules for FOL and SL (of course, it is incomplete as the classical SL). Because of the soundness, we can use the rules to prove the well-supported (and well-axiom-constrained) property by a two-step procedure:

1. Construct a substitution for each axiom in the program according to **Definition 2**, and use it to obtain a logic formula to be proven;
2. Try to use the inference rules to prove the formulas obtained in Step-1.

If we can prove an axiom ψ under environment Γ_N by the above procedure, we know that ψ is well-supported by N, and will write this fact as $\Gamma_N \vdash \psi$. We take it similar for $\Gamma_N \vdash \Psi$. Due to the soundness result, we have, if $\Gamma_N \vdash \Psi$, then $\Gamma_N \models \Psi$.

Because axioms are state-independent assertions (i.e., free of program variables), to prove whether an axiom is supported by a program, we need at most consulting

[2] We can obtain it from the complete set of predicates in the axiom by analyzing the axiom formula based on the given implementation. Not all predicates need to be unfolded in constructing the substitution for proving the axiom.

predicate definitions. After the proving, we know that the axioms are globally true over the implementation, thus can safely apply them in reasoning client programs which utilize the objects via the interfaces. We will demonstrate this in next section.

Now we give some properties with proofs that might facilitate the verification wrt axioms, due to the fact that some forms of program extension can keep the well-supportedness relation. First, we can verify axioms one by one, or in groups:

Lemma 1. *Assume $\mathit{\Gamma}_N \vdash \mathit{\Psi}$ and $\mathit{\Gamma}_N \vdash \mathit{\Psi}'$, then $\mathit{\Gamma}_N \vdash \mathit{\Psi} \cup \mathit{\Psi}'$.* □

Proof. By **Def. 3**, we know each axiom can be proven independent of other axioms under environment $\mathit{\Gamma}_N$. Thus if N supports two axiom groups $\mathit{\Psi}, \mathit{\Psi}'$, N also support their union set. □

Note that, **Lemma 1** also tells, a new axiom added for a program only brings proof obligations on itself but nothing on existing well-supported axioms. Thus the well-supported property of preceding axioms is modularly kept.

Then we give some cases of program extension with unchanged axiom set. We will use $N \nmid \mathit{\Psi}$ to mean N contains no declaration/definition of predicates in $\mathit{\Psi}$.

Lemma 2. *(1) Assume $\mathit{\Gamma}_N \vdash \mathit{\Psi}$, and N' is a sequence of interface declarations. If NN' is still a well-typed declaration sequence, then $\mathit{\Gamma}_{NN'} \vdash \mathit{\Psi}$.*
(2) If $\mathit{\Gamma}_N \vdash \mathit{\Psi}$, for a sequence of interface/class declarations N' where $N' \nmid \mathit{\Psi}$ and $N\,N'$ is still a well-typed declaration sequence, then $\mathit{\Gamma}_{NN'} \vdash \mathit{\Psi}$.
(3) If $\mathit{\Gamma}_N \vdash \mathit{\Psi}$ and $\mathit{\Gamma}_{N'} \vdash \mathit{\Psi}'$, where $N \nmid \mathit{\Psi}'$ and $N' \nmid \mathit{\Psi}$, and $N\,N'$ is still a well-typed declaration sequence, then $\mathit{\Gamma}_{NN'} \vdash \mathit{\Psi} \cup \mathit{\Psi}'$.

Proof. For (1), as we assume each predicate declared for a program has a distinct name, and interfaces give no definition for declared predicates, thus N' has no semantic effects on the well-supported axiom set $\mathit{\Psi}$ wrt N. So we trivially have $\mathit{\Gamma}_{N'} \vdash \mathit{\Psi}$ without any proof obligation. Then by **Lemma 1**, we get $\mathit{\Gamma}_{NN'} \vdash \mathit{\Psi}$.

For (2), it is similar to (1), because N' is assumed to provide no new semantic interpretations for axioms in $\mathit{\Psi}$, thus N' makes no effect on their truth values.

For (3), applying (2) in this lemma for two assumption pairs "$\mathit{\Gamma}_N \vdash \mathit{\Psi}, N' \nmid \mathit{\Psi}$", and "$\mathit{\Gamma}_{N'} \vdash \mathit{\Psi}', N \nmid \mathit{\Psi}'$", we can get $\mathit{\Gamma}_{NN'} \vdash \mathit{\Psi}$ and $\mathit{\Gamma}_{NN'} \vdash \mathit{\Psi}'$ respectively. Then by **Lemma 1**, we have the conclusion. □

In the case (2) of this lemma, a special case is that N' consists of some class declarations without any predicate definition and they use existing classes in N to implement their behaviors. We call such classes in N' *simple clients* of N. Clearly, all the axioms in $\mathit{\Psi}$ are true assertions for them and may be used in their verifications. If some *client* classes use existing classes and support another independent set of axioms, they fall into the coverage of the case (3) in this lemma. As a result, the new set of classes (with their axioms) can be directly combined with the existing classes (and axioms), because their verifications do not touch other implementation details.

However, in general case, adding a new class onto existing class/interface sequence may bring proof obligations wrt some related axioms. If this new comer also supports these related axioms (if any), we call it "a proper extension class" wrt existing components. Here we use *proper* instead of *correct* because we have not verified the methods in the classes according to their specifications yet. We can check whether an extension class is *proper* by the following definition:

Definition 5 (Proper Extension Class). *If $\Gamma_N \vdash \Psi$, K is a class declaration, and $N\,K$ is still well-typed. We say K is a* proper extension class *wrt (Ψ, N), if*

(1) $K \nmid \Psi$; or
(2) K provides a definition(s) for one (or more) predicate(s) appearing in an axiom subset Ψ_1 in Ψ, and $\Gamma_{N\,K} \vdash \Psi_1$.

Then for the well-supported axioms Ψ in existing components (Ψ, N), we have,

Lemma 3. *(1) If $\Gamma_N \vdash \Psi$ and K is a proper extension class wrt (Ψ, N), then $\Gamma_{N\,K} \vdash \Psi$.*
(2) If $\Gamma_N \vdash \Psi$, N' is a sequence of class/interface declarations, and for any class declaration K in N' such that $N' = N'_1\,K\,N'_2$, K is a proper extension class with respect to $(\Psi, N\,N'_1)$, then $\Gamma_{N\,N'} \vdash \Psi$.

Proof. For (1), we prove it according to the two cases of K in **Def. 5**. For case (1), because $K \nmid \Psi$, we have $\Gamma_{N\,K} \vdash \Psi$ by applying the case (2) in **Lemma 2**; for case (2), we know $\Gamma_{N\,K} \vdash \Psi_1$ by reverifying each axiom in Ψ_1 and $K \nmid (\Psi \setminus \Psi_1)$. Then we get $\Gamma_{N\,K} \vdash (\Psi \setminus \Psi_1)$ by the case (2) in **Lemma 2**. Therefore, we can have $\Gamma_{N\,K} \vdash \Psi_1 \cup (\Psi \setminus \Psi_1)$ (where $\Psi_1 \cup (\Psi \setminus \Psi_1) = \Psi$) by **Lemma 1**. To summarize, this case holds.

For (2), it can be proven by applying the above (1) in this lemma inductively on the position of K in sequence N'. □

In conclusion, **Lemmas 2, 3** reflect some cases of program extension where the *well-supported* property of axioms can be modularly maintained. Especially, **Lemma 3** also reflect a kind of proof obligation for ensuring behavioral subtypes, that is each extended subclass may need to check supporting the existing axioms as necessary. Further, if a new implementation modifies definitions of some predicates appearing in certain axioms, those related axioms should be firstly reverified to be well-supported before proving the correctness of this implementation. Of course, situations could become more complicated, and at that time, we might need to go back to the basic definitions.

4.2 Verifying Methods and Behavioral Subtyping

The second part of verification is relatively common: to verify that each method satisfies its specifications, i.e., the components are correctly implemented. We have given a set of rules for method verification, and list them in Appendix A with brief explanations.

Because of the existence of interfaces, multi-implementation, and inheritance, a class definition takes generally the form:

$$\textbf{class}\ C : B\ \rhd I_1, \ldots, I_k\ \{\ldots\ T\ m(\ldots)[\langle P\rangle\langle Q\rangle]\{\ldots\}\ \ldots\} \qquad (3)$$

where class C inherits B as its superclass and implements interfaces $I_1, \ldots, I_k$. For the m, it can be a new one, or one overriding another definition for m accessible in B; and it can be defined with an explicit specification $\langle P\rangle\langle Q\rangle$, or inherit its specifications from

B, or even from the interfaces. In addition, the definition should implement specification(s) for m in the interfaces, if exist(s). Also, C may inherit a method from B (but not define it) to implement a declared method in some interface(s) (I_i(s)).

An available method in class C may have a definition with explicit specification in C, or only a definition and inherited specifications from C's supertypes, or an inherited definition with also inherited specification from its superclass. These facts tell us that two interrelated problems must be resolved in verifying a method: (1) determining a specification and using it to verify the method body; (2) verifying that the method fits the need of both the superclass and the implemented interfaces. We consider them in the following, and first introduce some notations and definitions.

We think an interface defines a type, and a class defines a type with implementation. We will use $C, B, \ldots$ for class names, I for interface names, T for type names, to avoid simple conditions. We use $(T, T') \in \mathsf{super}$ to mean that T' is a direct supertype of T, and $T <: T'$ as the transitive and reflective closure of super. We use $\mathsf{super}(C)$ to get all supertypes of C, thus for example (3), $\mathsf{super}(C) = \{I_1, \ldots, I_k, B\}$. When C implements $I_1, I_2, \ldots$, and defines method m without giving a specification, m in C may have multiple specifications if more than one of the I_i has specifications for m. We write $\langle\varphi\rangle\langle\psi\rangle \in \mathit{\Pi}(T.m)$ in semantic rules to mean that $\langle\varphi\rangle\langle\psi\rangle$ is one specification of m in T, and write $\mathit{\Pi}(T.m) = \langle\varphi\rangle\langle\psi\rangle$ when $\langle\varphi\rangle\langle\psi\rangle$ is the only specification.

In semantics, we use $\mathit{\Gamma}, C, m \vdash \psi$ to state that ψ holds in method m of class C under $\mathit{\Gamma}$. Clearly, here ψ must be a state-independent formula. We use $\mathit{\Gamma}, C, m \vdash \{\varphi\}c\{\psi\}$ to say that command c in m of C satisfies the pair of precondition φ and postcondition ψ. We write $\mathit{\Gamma} \vdash \{\varphi\}C.m\{\psi\}$ (or $\mathit{\Gamma} \vdash \{\varphi\}C.C\{\psi\}$) to state that $C.m$ (constructor of C) is correct wrt $\langle\varphi\rangle\langle\psi\rangle$ under $\mathit{\Gamma}$. For methods with multiple specifications, we use $\mathit{\Gamma} \vdash C.m \rhd \mathit{\Pi}(C.m)$ to say that $C.m$ is correct wrt its every specification.

For OO programs, behavioral subtyping is crucial in verification. To introduce it here, we define a refinement relation between method specifications.

Definition 6 (Refinement of Specification). *Given two specifications $\langle\varphi_1\rangle\langle\psi_1\rangle$ and $\langle\varphi_2\rangle\langle\psi_2\rangle$, we say that the latter refines the former in context $\mathit{\Gamma}, C$,* iff *there exists an assertion R which is free of program variables, such that $\mathit{\Gamma}, C \vdash (\varphi_1 \Rightarrow \varphi_2 * R) \wedge (\psi_2 * R \Rightarrow \psi_1)$. We use $\mathit{\Gamma}, C \vdash \langle\varphi_1\rangle\langle\psi_1\rangle \sqsubseteq \langle\varphi_2\rangle\langle\psi_2\rangle$ to denote this fact. For multiple specifications $\{\pi_i\}_i$ and $\{\pi'_j\}_j$, we say $\{\pi_i\}_i \sqsubseteq \{\pi'_j\}_j$* iff *$\forall i\ \exists j \cdot \pi_i \sqsubseteq \pi'_j$.*

Liskov [16] defined the condition for specification refinement as $\varphi_1 \Rightarrow \varphi_2 \wedge \psi_2 \Rightarrow \psi_1$. We extend it by considering the storage extension (specified in R) and multiple specifications as above. It follows also the *nature refinement order* proposed by Leavens [14].

The behavioral subtyping relation should also be verified for interfaces with inheritance relations. Assume I has a super-interface I', and method m in I has a new specification $\langle\varphi\rangle\langle\psi\rangle$ overriding its counterpart $\langle\varphi'\rangle\langle\psi'\rangle$ in I', we must verify $\mathit{\Gamma}, I \vdash \langle\varphi'\rangle\langle\psi'\rangle \sqsubseteq \langle\varphi\rangle\langle\psi\rangle$ holds on the logic level, because of no method body involved.

Now we can define a class to be *correct* with two aspects of proof obligations wrt given specifications. That is, every defined method meets its specifications, and each subclass is a behavioral subtype of its superclass. We will use the inference rules listed in Appendix A to prove these. Note that in the rules for methods, we include premises for verifying the behavioral subtyping relation.

Definition 7 (Correct Class). *A class C defined in program G is* correct*, iff,*

- *for each method m defined in C, we have $\Gamma_G \vdash C.m \rhd \Pi(C.m)$, and for the constructor of C with $\Pi_G(C.C) = \langle\varphi\rangle\langle\psi\rangle$, we have $\Gamma_G \vdash \{\varphi\}C.C\{\psi\}$;*
- *if C is defined as a subclass of class D in G, then C is a behavioral subtype of D.*

Then, we define a program with axioms to be *correct* as follows:

Definition 8 (Correct Program). *Program G is* correct*, iff,*

(1) G is well-axiom-constrained according to **Def. 4**.
(2) Each class C defined in G is correct according to **Def. 7**.

It is easy to conclude, our extended verification framework with axioms of VeriJ is sound because the assertion logic used and all inference rules have been proven sound.

5 Case Study

Having the enriched specification and verification framework, in this section we will re-examine the MVC example discussed in Section 2, to see how the problems mentioned there can be tackled naturally and the two roles of axioms.

5.1 Specifying the MVC Architecture

Following the guideline in **Fig. 2**, we have declared interfaces MI, CI, VI in **Fig. 3** to embody the calculator design. Some specification predicates with respective purposes as we explained have been introduced to form a foundation for formal method specification. Each predicate should have a declaration, as "$\mathbf{def}\ \mathit{model}(\mathbf{this}, vs, st)$;" in the interface, but we omit them here to save space. These declarations introduce predicate names with parameters, all as abstract symbols, and their concrete meaning (possibly multiple) will be defined later in implementing class(es) of the interfaces.

However, not any definition for the predicates is acceptable, and some predicates may have interconnections with others. In order to reflect our anticipation in correctly specifying the calculator requirements, preventing wrong implementations, and providing enough information for client verifications, we need to constrain definitions of the predicates in later implementations and their correct uses in specifications by revealing their relations or properties. Applying our approach in Section 4, we specify a set of axioms labeled as [a1-a3] according to the requirements in **Fig.2**:

$$\begin{aligned} &\mathbf{axiom}\ \mathit{MVC}(c,m,vs,st) \Leftrightarrow\ \mathit{model}(m,vs,st) * (\circledast_{v\in vs}\mathit{view}(v,m,st)) \\ &\qquad\qquad * \mathit{controller}(c,m,st); \qquad [a1] \\ &\mathbf{axiom}\ \mathit{MVs}(m,vs,st) \Leftrightarrow\ \mathit{model}(m,vs,st) * (\circledast_{v\in vs}\mathit{view}(v,m,st)); \qquad [a2] \\ &\mathbf{axiom}\ \mathit{MVC}(c,m,vs,st) \Leftrightarrow\ \mathit{MVs}(m,vs,st) * \mathit{controller}(c,m,st); \qquad [a3] \end{aligned}$$

The axioms form a part of specifications to capture important interactions or properties of the MVC architecture, and constrain the forthcoming implementations. Semantically, any implementation should fulfill them, and the definitions for the methods declared in the interfaces must obey these constraints which will generate proof

```
class Model ▷ MI{
  def model(this, vs, st) :
    this.total ↦ st * this.views ↦ vs;
  def MVs(this, vs, st) :
    model(this, vs, st) * (⊛_{v∈vs} view(v, this, st));
  def product(n, m) : n ⋆ m;
  Int total; List⟨VI⟩views;
  Model()⟨emp⟩⟨model(this, ∅, 1)⟩
  { this.total = 1;
    this.views = new ArrayList⟨VI⟩(); }
  Int getState(){ Int s;
    s = this.total; return s; }
  void addView( VI v){
    views.add(v); v.update(this); }
  void Multiply(Int i){
    this.total = this.total * i; this.Notify(); }
  void Reset() {
    this.total = 1; this.Notify(); }
  void Notify()⟨model(this, vs, st)*
  (⊛_{v∈vs} view(v, this, st′))⟩⟨MVs(this, vs, st)⟩
  { for( VI v : views) v.update(this); }
}
class View ▷ VI{
  def view(this, m, st) :
    this.model ↦ m * this.vtotal ↦ st;
  MI model; Int vtotal;
  View(MI m) ⟨model(m, vs, st)⟩
  ⟨model(m, vs ∪ {this}, st) * view(this, m, st)⟩
  { this.model = m; this.vtotal = 0;
    m.addView(this); }
  void update(MI m){
    if (m==this.model)
      this.vtotal = m.getState();
    this.paint(); }
```

```
  void paint(){
    Int v = this.vtotal;
    System.out.println("Product is: ")+v; }}
class View_2 ▷ VI{
  def view(this, m, st) :
    this.model ↦ m * this.vtotal ↦ (st + 1);
  MI model; Int vtotal;
  View_2(MI m)⟨model(m, vs, st)⟩
  ⟨model(m, vs ∪ {this}, st)*
    view(this, m, st)⟩ {...}
  void update(MI m){
    if (m==this.model)
      this.vtotal = m.getState() + 1;
    this.paint(); }
  void paint(){...}
}
class Controller ▷ CI{
  def controller(this, m, st) :
    this.model ↦ m * this.state ↦ st;
  def MVC(this, m, vs, st) :
    model(m, vs, st) * (⊛_{v∈vs} view(v, m, st))*
    controller(this, m, st);
  MI model; Int state;
  Controller(MI m) ⟨model(m, vs, st)⟩
  ⟨controller(this, m, st) * model(m, vs, st)⟩
  { this.model = m; this.state = m.getState(); }
  void multiplyPerformed(){
    Scanner scan = new Scanner(System.in);
    System.out.print("Enter a number: ");
    Int i = scan.nextInt(); model.Multiply(i);
    Int j = model.getState(); this.state = j; }
  void resetPerformed(){
    this.state = 1; model.Reset(); }
}
```

Fig. 7. An Implementation of the MVC Calculator Interfaces

obligations. In this way, although the interfaces provide no behavior definitions, their implementations have been connected formally by the predicates and axioms.

In **Fig. 7**, we give four classes *Model*, *Controller*, *View* and $View_2$ which implement the interfaces and form an implementation of the MVC calculator. We use class *Scanner* in package "java.util.Scanner" and call its method $nextInt()$ to get an integer from the input stream for multiplying. Abstractly, we specify method $Scanner.nextInt()$ as "$\langle \mathbf{true} \rangle \langle \exists n \cdot \mathsf{res} = n \rangle$". All predicates declared in the interfaces are defined with bodies in relative classes that give also specific meaning for the axioms. For example, axiom [a1] tells the whole MVC can be divided into a model object, its controller object and its view-object set; [a2] means the model-views aggregate structure consists of a model object and its view-object set; and [a3] says the whole MVC can also be viewed as consisting of a model-views aggregate structure with a controller object.

5.2 Verifying Implementations with Axioms and Method Specifications

Now we consider verifying the implementation before verifying client codes. **Def. 8** lists two parts of work for concluding the correctness of the implementation: (1) checking it supports axioms [a1-a3] by applying the two-step procedure given in Section 4; (2) checking each declared method satisfies its specifications. Due to limited space, we only give detailed proofs of axiom [a1] and called methods by client here, and leave other proofs in Appendix B. Also we present some discussions about the proof steps which may be deduced interactively or automatically in a proof assistant.

For axiom [a1], we construct a substitution[3] under Γ of the implementation:

$$\begin{aligned}&\delta_{\{MVC,controller\},\Gamma} \mathrel{\widehat{=}} [\\ &\quad c : Controller \wedge \mathsf{fix}(Controller, MVC(c,m,vs,st))/MVC(c,m,vs,st),\\ &\quad c : Controller \wedge \mathsf{fix}(Controller, controller(c,m,st))/controller(c,m,st)\ ]\end{aligned}$$

That is because only class $Controller$ defines predicates $MVC(\ldots)$ and $controller(\ldots)$. By applying this substitution on the assertion of [a1] (simply denoted as ψ), we get the following logic formula (4)[4] to prove,

$$\begin{aligned}\psi\delta_{\{MVC,controller\},\Gamma} = {}& c : Controller \wedge \mathsf{fix}(Controller, MVC(c,m,vs,st))\\ &\Leftrightarrow model(m,vs,st) * (\circledast_{v\in vs} view(v,m,st)) *\\ &\ \ c : Controller \wedge \mathsf{fix}(Controller, controller(c,m,st))\end{aligned} \tag{4}$$

Using the definition of fix and inference rules, we know

$c : Controller \wedge \mathsf{fix}(Controller, MVC(c,m,vs,st)) \Leftrightarrow Controller.MVC(c,m,vs,st)$[5]

and similar for $\mathsf{fix}(Controller, controller(\ldots))$. Thus we can reduce (4) to[6]:

$$\begin{aligned}Controller.MVC(c,m,vs,st) \Leftrightarrow{}& model(m,vs,st) * (\circledast_{v\in vs} view(v,m,st)) *\\ & Controller.controller(c,m,st)\end{aligned} \tag{5}$$

Then, from Γ, we have $\Phi(Controller.MVC(\mathbf{this},m,vs,st)) = model(m,vs,st) * controller(\mathbf{this},m,st) * (\circledast_{v\in vs} view(v,m,st))$. Using rule [EXPAND] we get[7]

$$\begin{aligned}&Controller.MVC(c,m,vs,st) \Leftrightarrow \mathsf{fix}(Controller, model(m,vs,st) *\\ &\quad (\circledast_{v\in vs} view(v,m,st)) * controller(\mathbf{this},m,st))[c/\mathbf{this}]\\ &\Leftrightarrow (\mathsf{fix}(Controller, model(m,vs,st)) * \mathsf{fix}(Controller, \circledast_{v\in vs} view(v,m,st)) *\\ &\quad \mathsf{fix}(Controller, controller(\mathbf{this},m,st)))[c/\mathbf{this}]\\ &\Leftrightarrow model(m,vs,st) * (\circledast_{v\in vs} view(v,m,st)) *\\ &\quad Controller.controller(\mathbf{this},m,st)[c/\mathbf{this}]\\ &\Leftrightarrow model(m,vs,st) * (\circledast_{v\in vs} view(v,m,st)) * Controller.controller(c,m,st)\end{aligned}$$

Thus, we have proven that [a1] is well supported.

[3] This substitution is constructed heuristically by analyzing the axiom assertion with related predicate definitions in Γ. Substitutions for predicates which need not to unfold can be reduced because they do not affect the proof result. However, in a prover, this substitution would be mechanically constructed for expanding each predicate appearing in the axiom.

[4] This formula can be gained automatically by applying the constructed substitution $\delta_{\{MVC,controller\},\Gamma}$ on ψ of [a1] in a proof assistant.

[5] This equivalence can be interactively inferred by applying fix function as a tactic.

[6] This step can be gained interactively by applying the above equivalence with (4).

[7] The below proof steps can be deduced interactively by applying the definition of $Controller.MVC(\ldots)$ and [EXPAND] rule as tactics on the left side of (5).

For other axioms, we will give their proof processes with necessary labels for explanation as below. For example, label "[Def. 2]" means **Def. 2** is applied for the previous formula of this labeled line; "[Def. of $\mathsf{fix}(D,\psi)$]" means the definition of function fix is applied; "[Rule [EXPAND]]" says the inference rule [EXPAND] is used; "[Def. of $\Phi(\mathit{Model}.\mathit{MVs}(\ldots))$]" tells folding/unfolding the definition of predicate $\mathit{MVs}(\ldots)$ in class *Model* is applied; and labels like "[(L-a2)]" give just a tag to the formula in the current line. In the deduction steps without explicit labels, inference rules of our framework would be used.

For axiom [a2], we can construct a substitution

$$\begin{aligned}\delta_{\{\mathit{MVs},\mathit{model}\},\Gamma} \mathrel{\widehat{=}} [&m : \mathit{Model} \wedge \mathsf{fix}(\mathit{Model}, \mathit{MVs}(m,vs,st))/\mathit{MVs}(m,vs,st),\\ &m : \mathit{Model} \wedge \mathsf{fix}(\mathit{Model}, \mathit{model}(m,vs,st))/\mathit{model}(m,vs,st)]\end{aligned}$$

and then have the following deduction in respect to the two directions of [a2]:
(‡ Deducing from the left side of [a2]:)

$$\begin{aligned}
&\mathit{MVs}(m,vs,st)\delta_{\{\mathit{MVs},\mathit{model}\},\Gamma}\\
\Leftrightarrow\ & m : \mathit{Model} \wedge \mathsf{fix}(\mathit{Model}, \mathit{MVs}(m,vs,st)) && \text{[Def. 2]}\\
\Leftrightarrow\ & \mathit{Model}.\mathit{MVs}(m,vs,st) && \text{[Def. of } \mathsf{fix}(D,\psi)\text{]}\\
\Leftrightarrow\ & \mathsf{fix}(\mathit{Model}, \Phi(\mathit{Model}.\mathit{MVs}(\mathbf{this},vs,st)))[m/\mathbf{this}] && \text{[Rule [EXPAND]]}\\
\Leftrightarrow\ & \mathsf{fix}(\mathit{Model}, \mathit{model}(\mathbf{this},vs,st) * (\circledast_{v\in vs}\mathit{view}(v,\mathbf{this},st)))[m/\mathbf{this}]\\
& && \text{[Def. of } \Phi(\mathit{Model}.\mathit{MVs}(\ldots))\text{]}\\
\Leftrightarrow\ & (\mathit{Model}.\mathit{model}(\mathbf{this},vs,st) * (\circledast_{v\in vs}\mathit{view}(v,\mathbf{this},st)))[m/\mathbf{this}] && \text{[Def. of } \mathsf{fix}(D,\psi)\text{]}\\
\Leftrightarrow\ & \mathit{Model}.\mathit{model}(m,vs,st) * (\circledast_{v\in vs}\mathit{view}(v,m,st)) && \text{[(L-a2)]}
\end{aligned}$$

(‡ Deducing from the right side of [a2]:)

$$\begin{aligned}
&(\mathit{model}(m,vs,st) * (\circledast_{v\in vs}\mathit{view}(v,m,st)))\delta_{\{\mathit{MVs},\mathit{model}\},\Gamma}\\
\Leftrightarrow\ & (m : \mathit{Model} \wedge \mathsf{fix}(\mathit{Model}, \mathit{model}(m,vs,st))) * (\circledast_{v\in vs}\mathit{view}(v,m,st)) && \text{[Def. 2]}\\
\Leftrightarrow\ & \mathit{Model}.\mathit{model}(m,vs,st) * (\circledast_{v\in vs}\mathit{view}(v,m,st)) && \text{[Def. of } \mathsf{fix}(D,\psi)\text{] [(R-a2)]}
\end{aligned}$$

Seeing that the formula with tag (L-a2) equals to the one with tag (R-a2), the left side of axiom [a2] equals to its right side. Thus, [a2] is well supported under Γ.

For axiom [a3], we take the constructed substitution $\delta_{\{\mathit{MVC},\mathit{MVs},\mathit{model},\mathit{controller}\},\Gamma}$ in proving [a1] and [a2], and have:
(‡ Deducing from the left side of [a3]:)

$$\begin{aligned}
&\mathit{MVC}(c,m,vs,st)\delta_{\{\mathit{MVC},\mathit{MVs},\mathit{model},\mathit{controller}\},\Gamma}\\
\Leftrightarrow\ & (c : \mathit{Controller} \wedge \mathsf{fix}(\mathit{Controller}, \mathit{MVC}(c,m,vs,st)))\delta_{\{\mathit{model}\},\Gamma} && \text{[Def. 2]}\\
\Leftrightarrow\ & (\mathit{Controller}.\mathit{MVC}(c,m,vs,st))\delta_{\{\mathit{model}\},\Gamma} && \text{[Def. of } \mathsf{fix}(D,\psi)\text{]}\\
\Leftrightarrow\ & (\mathsf{fix}(\mathit{Controller}, \Phi(\mathit{Controller}.\mathit{MVC}(\mathbf{this},m,vs,st)))\delta_{\{\mathit{model}\},\Gamma})[c/\mathbf{this}]\\
& && \text{[Rule [EXPAND]]}\\
\Leftrightarrow\ & (\mathsf{fix}(\mathit{Controller}, \mathit{model}(m,vs,st) * (\circledast_{v\in vs}\mathit{view}(v,m,st)) *\\
&\qquad \mathit{controller}(\mathbf{this},m,st))\delta_{\{\mathit{model}\},\Gamma})[c/\mathbf{this}] && \text{[Def. of } \Phi(\mathit{Controller},\mathit{MVC}(\ldots))\text{]}\\
\Leftrightarrow\ & ((\mathit{model}(m,vs,st) * (\circledast_{v\in vs}\mathit{view}(v,m,st)) * \mathit{Controller}.\mathit{controller}(\mathbf{this},m,st))\\
&\qquad \delta_{\{\mathit{model}\},\Gamma})[c/\mathbf{this}] && \text{[Def. of } \mathsf{fix}(D,\psi)\text{]}\\
\Leftrightarrow\ & ((\mathit{model}(m,vs,st)\delta_{\{\mathit{model}\},\Gamma}) * (\circledast_{v\in vs}\mathit{view}(v,m,st)) *\\
&\qquad \mathit{Controller}.\mathit{controller}(\mathbf{this},m,st))[c/\mathbf{this}]\\
\Leftrightarrow\ & ((m : \mathit{Model} \wedge \mathsf{fix}(\mathit{Model}, \mathit{model}(m,vs,st))) * (\circledast_{v\in vs}\mathit{view}(v,m,st)) *\\
&\qquad \mathit{Controller}.\mathit{controller}(\mathbf{this},m,st))[c/\mathbf{this}] && \text{[Def. 2]}
\end{aligned}$$

$$\begin{aligned}
&\Leftrightarrow (\mathit{Model.model}(m, vs, st) * (\circledast_{v\in vs}\mathit{view}(v, m, st)) * \\
&\qquad \mathit{Controller.controller}(\mathbf{this}, m, st))[c/\mathbf{this}] \text{ [Def. of } \mathsf{fix}(D, \psi)] \\
&\Leftrightarrow \mathit{Model.model}(m, vs, st) * (\circledast_{v\in vs}\mathit{view}(v, m, st)) * \\
&\qquad \mathit{Controller.controller}(\mathbf{this}, m, st)[c/\mathbf{this}] \\
&\Leftrightarrow \mathit{Model.model}(m, vs, st) * (\circledast_{v\in vs}\mathit{view}(v, m, st)) * \\
&\qquad \mathit{Controller.controller}(c, m, st) \qquad \text{[(L-a3)]}
\end{aligned}$$

(‡ Deducing from the right side of [a3]:)

$$\begin{aligned}
&(\mathit{MVs}(m, vs, st) * \mathit{controller}(c, m, st))\delta_{\{\mathit{MVC},\mathit{MVs},\mathit{model},\mathit{controller}\},\Gamma} \\
&\Leftrightarrow (m : \mathit{Model} \wedge \mathsf{fix}(\mathit{Model}, \mathit{MVs}(m, vs, st))) * (c : \mathit{Controller} \wedge \\
&\qquad \mathsf{fix}(\mathit{Controller}, \mathit{controller}(c, m, st))) \qquad \text{[Def. 2]} \\
&\Leftrightarrow \mathit{Model.MVs}(m, vs, st) * \mathit{Controller.controller}(c, m, st) \text{ [Def. of } \mathsf{fix}(D, \psi)] \\
&\Leftrightarrow \mathsf{fix}(\mathit{Model}, \Phi(\mathit{Model.MVs}(\mathbf{this}, vs, st)))[m/\mathbf{this}] * \\
&\qquad \mathit{Controller.controller}(c, m, st) \qquad \text{[Rule [EXPAND]]} \\
&\Leftrightarrow \mathsf{fix}(\mathit{Model}, \mathit{model}(\mathbf{this}, vs, st) * (\circledast_{v\in vs}\mathit{view}(v, m, st)))[m/\mathbf{this}] * \\
&\qquad \mathit{Controller.controller}(c, m, st) \qquad \text{[Def. of } \mathit{Model.MVs}(\ldots)] \\
&\Leftrightarrow (\mathit{Model.model}(\mathbf{this}, vs, st) * (\circledast_{v\in vs}\mathit{view}(v, m, st)))[m/\mathbf{this}] * \\
&\qquad \mathit{Controller.controller}(c, m, st) \qquad \text{[Def. of } \mathsf{fix}(D, \psi)] \\
&\Leftrightarrow \mathit{Model.model}(m, vs, st) * (\circledast_{v\in vs}\mathit{view}(v, m, st)) * \\
&\qquad \mathit{Controller.controller}(c, m, st) \qquad \text{[(R-a3)]}
\end{aligned}$$

Because the formula with (L-a3) equals to the one with (R-a3), the equivalence of axiom [a3] is proven. Thus [a3] is also well supported under Γ.

In conclusion, each specified axiom can be independently well-supported by the given implementation of MVC. Additionally, a well-supported axiom can be applied for checking other axioms. For example, axiom [a3], can be deduced out from conjunction of axioms [a1] and [a2] if they two are proven firstly. Then these axioms can be used in verifying the implementation and client codes. Because they are (dual directions) equivalences, we can equivalently substitute one assertion (or part of a whole one) if which is an instance of an axiom's one side assertion, into another assertion instantiating the other side of the axiom.

As the next step, we turn to verify that each method is correct wrt its specifications. Readers can refer our Appendix B for verifications of most methods. We give detail proofs of methods $\mathit{Controller.resetPerformed}()$, $\mathit{Controller.multiplyPerformed}()$ and $\mathit{View.View}(m)$ here in **Fig. 8**, because they are called in the illustrating client method. In the proof, labels like "[Def. of $C.p(\ldots)$]" with similar meaning explained in proving axioms above are used. And one more label as "[Axiom [a3][$\mathbf{this}/c$] (R/L)]" is written to mean using axiom [a3] from its left side (L) to right side (R) by substituting parameter c to $\mathbf{this}$. Finally, we conclude these three methods are correct.

Having proven that all axioms are well-supported and all methods satisfy their specifications, we know the given implementation is correct for the MVC architecture.

5.3 Verifying Client Methods

At last, we resolve the verification of the client method in **Fig. 9**, by using the above extended specifications on interfaces including axioms [a1-a3] and similar labels as in

Proving $View.View(\mathbf{m})$:
$\{model(m, vs, st)\}$
$\mathbf{this}.model = m;\ \mathbf{this}.vtotal = 0;$
$\{model(m, vs, st)*$
$\quad \mathbf{this}.model \mapsto m * \mathbf{this}.vtotal \mapsto 0\}$
$\{model(m, vs, st) * view(\mathbf{this}, m, 0)\}$
[Def. of $View.view(\ldots)$]
$m.addView(\mathbf{this});$
$\{model(m, vs \cup \{\mathbf{this}\}, st)*$
$\quad view(\mathbf{this}, m, st)\}$

Proving $Controller.resetPerformed()$:
$\{MVC(\mathbf{this}, m, vs, st)\}$
$\{MVs(m, vs, st) * controller(\mathbf{this}, m, st)\}$
[Axiom [a3][$\mathbf{this}/c$] (R/L)]
$\{MVs(m, vs, st) * \mathbf{this}.model \mapsto m*$
$\quad \mathbf{this}.state \mapsto st\}$
[Def. of $Controller.controller(\ldots)$]
$\mathbf{this}.state = 1;$
$\{MVs(m, vs, st) * \mathbf{this}.model \mapsto m*$
$\quad \mathbf{this}.state \mapsto 1\}$
$\{MVs(m, vs, st) * controller(\mathbf{this}, m, 1)\}$
[Def. of $Controller.controller(\ldots)$]
$model.Reset();$
$\{MVs(m, vs, 1) * controller(\mathbf{this}, m, 1)\}$
$\{MVC(\mathbf{this}, m, vs, 1)\}$
[Axiom [a3][$\mathbf{this}/c$] (L/R)]

Proving $Controller.multiplyPerformed()$:
$\{MVC(\mathbf{this}, m, vs, st)\}$
$Scanner\ scan = \mathbf{new}\ Scanner$(System.in);
System.out.print("Enter a number: ");
$\mathbf{Int}\ i = scan.nextInt();$
$\{\exists n \cdot i = n \wedge MVC(\mathbf{this}, m, vs, st)\}$
$\{\exists n \cdot i = n \wedge MVs(m, vs, st)*$
$\quad controller(\mathbf{this}, m, st)\}$
[Axiom [a3][$\mathbf{this}/c$] (R/L)]
$model.Multiply(v);$
$\{\exists\, n, st' \cdot i = n \wedge st' = product(st, i) \wedge$
$\quad MVs(m, vs, st') * controller(\mathbf{this}, m, st)\}$
$\{\exists\, n, st' \cdot i = n \wedge st' = product(st, i) \wedge$
$\quad model(m, vs, st') * (\circledast_{v \in vs} view(v, m, st'))*$
$\quad controller(\mathbf{this}, m, st)\}$
[Axiom [a2][st'/st] (R/L)]
$\mathbf{Int}\ j = model.getState();$
$\{\exists\, n, st' \cdot i = n \wedge st' = product(st, i) \wedge j = st'$
$\quad \wedge model(m, vs, st') * (\circledast_{v \in vs} view(v, m, st'))$
$\quad * controller(\mathbf{this}, m, st)\}$
$\{\exists n, st' \cdot i = n \wedge st' = product(st, i) \wedge j = st'$
$\quad \wedge model(m, vs, st') * (\circledast_{v \in vs} view(v, m, st'))$
$\quad * \mathbf{this}.model \mapsto m * \mathbf{this}.state \mapsto st\}$
[Def. of $Controller.controller(\ldots)$]
$\mathbf{this}.state = j;$
$\{\exists n, st' \cdot i = n \wedge st' = product(st, i) \wedge j = st'$
$\quad \wedge model(m, vs, st') * (\circledast_{v \in vs} view(v, m, st'))$
$\quad * \mathbf{this}.model \mapsto m * \mathbf{this}.state \mapsto j\}$
$\{\exists n, st' \cdot i = n \wedge st' = product(st, i) \wedge$
$\quad model(m, vs, st') * (\circledast_{v \in vs} view(v, m, st'))*$
$\quad controller(\mathbf{this}, m, st')\}$
[Def. of $Controller.controller(\ldots)$]
$\{\exists n, st' \cdot i = n \wedge st' = product(st, i) \wedge$
$\quad MVC(\mathbf{this}, m, vs, st')\}$
[Axiom [a3][$\mathbf{this}/c$] (L/R)]
$\{\exists st' \cdot MVC(\mathbf{this}, m, vs, st')\}$

Fig. 8. The Verification of Three Methods

method verifications. It shows that, we can finish the verification of the client now, thus it makes a correct application of the MVC calculator.

5.4 Discussions

To conclude, we discuss two aspects with the case study: (1) the abstraction of axioms, (2) the proof steps which can be mechanically done interactively or automatically in an interactive proof assistant.

The Abstraction of Axioms Our axioms are specified on interface hierarchy to constrain the whole system implementations according to the informal requirements. They have only logical variables and abstract predicate applications, and thus are more abstract than predicate definitions in two aspects. For one thing, predicate definitions may contain concrete implementing information like instance fields but axioms would not. For another, especially the first parameter of any predicate application in an axiom can be of the type which declares the predicate and also its any subtype. However, in a predicate definition, its type must be the current one declares the predicate. To check the correctness of any implementation with its specifications, we should check each axiom with the implementation ahead. Then we can apply axioms in reasoning codes independent of any concrete implementation including abstract predicate definitions.

(1) $\{\exists i \cdot \mathit{model}(m, \emptyset, i) * \mathit{controller}(c, m, i)\}$

(2) $\mathit{VI}\ v_1 = \textbf{new}\ \ \mathit{View}(m);$

(3) $\{\exists i \cdot \mathit{model}(m, \{v_1\}, i) * \mathit{view}(v_1, m, i) * \mathit{controller}(c, m, i)\}$

(4) $\textbf{Int}\ n = 1;$

(5) $\{\exists i \cdot n = 1 \wedge \mathit{model}(m, \{v_1\}, i) * \mathit{view}(v_1, m, i) * \mathit{controller}(c, m, i)\}$

(5′) $\{\exists i, n_1 \cdot n = 1 \wedge n = n_1 \wedge 1 < 4 \wedge \mathit{model}(m, \{v_1\}, i) * \mathit{view}(v_1, m, i) *$
$\mathit{controller}(c, m, i)\}$

(5″) $\{\exists i, n_1 \cdot n = 1 \wedge n = n_1 \wedge n_1 < 4 \wedge \mathit{MVC}(c, m, \{v_1\}, i)\}$
[Axiom [a1] [$\{v_1\}, v_1, i/vs, v, st$] (L/R)]

(6) $\textbf{while}\ (n < 4)$

(7) $\{\ \{\exists i', n_1' \cdot n = n_1' \wedge n_1' < 4 \wedge \mathit{MVC}(c, m, \{v_1\}, i')\}$ [—-loop invariant]

(8) $c.\mathit{multiplyPerformed}();$

(9) $\{\exists i'', n_1' \cdot n = n_1' \wedge n_1' < 4 \wedge \mathit{MVC}(c, m, \{v_1\}, i'')\}$

(10) $n{+}{+};$

(11) $\{\exists i'', n_1', n_2 \cdot n = n_2 \wedge n_2 = n_1' + 1 \wedge n_2 \leq 4 \wedge \mathit{MVC}(c, m, \{v_1\}, i'')\}$

(11′) $\{\exists i'', n_2 \cdot n = n_2 \wedge n_2 \leq 4 \wedge \mathit{MVC}(c, m, \{v_1\}, i'')\}\ \}$

(12) $\{\exists j \cdot n = 4 \wedge \mathit{MVC}(c, m, \{v_1\}, j)\}$

(12′) $\{\exists j \cdot \mathit{model}(m, \{v_1\}, j) * \mathit{view}(v_1, m, j) * \mathit{controller}(c, m, j)\}$
[Axiom [a1][$\{v_1\}, v_1, j/vs, v, st$] (R/L)]

(13) $\mathit{VI}\ v_2 = \textbf{new}\ \ \mathit{View}_2(m);$

(14) $\{\exists j \cdot \mathit{model}(m, \{v_1, v_2\}, j) * \mathit{view}(v_1, m, j) * \mathit{view}(v_2, m, j) * \mathit{controller}(c, m, j)\}$

(14′) $\{\exists j \cdot \mathit{MVC}(c, m, \{v_1, v_2\}, j)\}$ [Axiom [a1][$\{v_1, v_2\}, j/vs, st$] (L/R)]

(15) $c.\mathit{resetPerformed}();$

(16) $\{\mathit{MVC}(c, m, \{v_1, v_2\}, 1)\}$

(16′) $\{\exists r_1, r_2 \cdot \mathit{MVC}(c, m, \{r_1, r_2\}, 1)\}$

Fig. 9. The Correct Proof of the Client Method

In the case study, it seems that the definition of predicate like $\mathit{MVC}(\textbf{this}, m, vs, st)$ duplicates the assertion of axiom [a1]. Actually, in [a1], the type of the first parameter c in $\mathit{MVC}(c, m, vs, st)$ and $\mathit{controller}(c, m, st)$ would be the interface type CI, or any implementing class type (like $\mathit{Controller}$ and its subtypes) for CI. However, in the definition of $\mathit{MVC}(\textbf{this}, m, vs, st)$ in class $\mathit{Controller}$, the type of parameter $\textbf{this}$ is just the current $\mathit{Controller}$.

Automatically or Interactively Proving? By encoding the inference rules in framework VeriJ (including the ones in OOSL) into the inference system of an interactive proof assistant, we can interactively reason our axioms and implementations with specifications. Some simple proof steps can be done automatically while most should be interactively deduced with suitable tactics like axioms, predicate definitions, concept definitions, special inference rules and so on.

For example, in the case study, each label [...] as an annotation attached to the end of a formula is an interactive tactic applied onto the formula in the previous line. Then we can gain the labeled formulas. Any formula without explicit labels is deduced by using basic inference rules. We give detailed explanations for each proof step as footnotes in proving axiom [a1], and simplify others as labels.

6 Related Work and Conclusion

In this paper, we focus on specifying and verifying OO programs which are built on interactive components through clearly defined interfaces. We propose the axioms to relate abstract predicates for the specification of interfaces. These axioms semantically constrain the implementations and interactions in component-based systems (CBSs), and support the verification of clients which are defined based on interfaces abstractly (i.e., without consulting the concrete implementations nor the hidden predicate definitions) and modularly (i.e., avoiding reverification).

In the enriched foundation of framework VeriJ, we require checking each system implementation in two aspects: first the well-supportedness of each axiom, and then the correctness of each method with its specifications. Behavioral subtyping property is also ensured by checking specification refinement relations. Further we well support information hiding and extensibility in specifying and verifying OO programs.

To our limited knowledge, there exist some works on specifying and reasoning CBSs in different ways. Leavens *et al.* [12] combined model variables [4] and model programs to specify interfaces of CBSs, and extended the behavioral subtyping concept for CBSs. However, they only pointed out subtypes should obey the specifications of instance methods, no formalization details for the modular reasoning was given. Henzinger *et al.* [6] gave a type system for component interaction by checking component compatibility with some interface automata but touched no real semantics. Our idea requires compatible components to meet both method specifications and global axioms specified in/on interfaces. Their refinement is only from the interface designs to implementations, without behavioral subtyping as what we consider. Aguirre *et al.* [2] used algebraic specification of Guttag *et al.* [9] with temporal logic to define the ADTs of components, and specified interactions of components in special specification modules. Except the axioms relating actions of components in the ADTs, they also gave some axioms independent of particular subsystem declarations to express properties of their class instances and associations. Poetzsch-Heffter *et al.* [21] adopted model variables and pure methods in specifying interfaces of encapsulated components too. They specified invariants expressing properties of components but the behavioral subtype relation for components was absent.

On the other side, *object invariant*s in JML [11] and Spec# [3], and *axiom*s in MultiStar [24] are also specified to constrain subclasses through specification inheritance as ours act. However, compared with axioms, object invariants are less abstract and less powerful because: (1) they only hold at particular program points in operations while axioms hold always; (2) they constrain operations but axioms constrain logical data abstractions; (3) verifications for axioms which are done ahead of method verifications do not involve methods, but invariants do; (4) axioms are expressed as logical predicates, but invariants are in term of fields and pure methods in implementation; etc. Moreover, Dafny, an automatic program verifier for functional correctness developed by Leino [15], uses invariants in terms of valid mathematical functions and ghost variables, but it supports neither subtyping nor interface-based design.

Using the technique of abstract predicate family of Parkinson *et al.* [20], Van Staden *et al.* [24] expressed separately properties for individual classes and entire multiple hierarchies correspondingly in *export*s and *axiom*s in MultiStar. We uniform their two kinds into our axioms, where each predicate application encapsulates all its polymorphic definitions in implementations and its meaning can be determined by applying fix function and inference rules. Differing from *exports* in MultiStar, we can inherit individual properties to restrict subclasses. As we can inherit and reuse predicate definitions from superclass, our specification and verification is less complex but more modular. Still, we avoid infinite expansion of recursive predicate definitions in proving axioms and method specifications which is not considered in MultiStar. *Export*s in jStar [7] expressed interactive objects from different classes and enabling client verifications, however, it cannot restrict subclasses.

Using the technique of abstract predicate family of Parkinson et al. [20], Van Staden et al. [24] separately expressed properties for individual classes and entire multiple hierarchies correspondingly in *export*s and *axiom*s in MultiStar. We uniform their two kinds into our axioms, where each predicate application encapsulates all its polymorphic definitions in implementations and its meaning can be determined by applying fix function and inference rules. Differing from exports in MultiStar, we can inherit individual properties to restrict subclasses. As we can inherit and reuse predicate definitions from superclass, our specification and verification is less complex but more modular. Still, we avoid infinite expansion of recursive predicate definitions in proving axioms and method specifications which is not considered in MultiStar. *Export*s in jStar [7] which expressed interactive objects from different classes and enabling client verifications cannot restrict subclasses.

As future work, we would investigate more challenges [13] such as object invariant, frame problem in specifying and verifying OO programs. We attempt to apply our approach for more interactive programs such as design patterns [8] and web applications, and consider specifying and verifying CBSs with problems like adaptation [1], composition [19], and so on. Meanwhile, we are working on an implementation of our theoretical framework using Coq.

References

1. Adler, R., Schaefer, I., Trapp, M., Poetzsch-Heffter, A.: Component-based modeling and verification of dynamic adaptation in safety-critical embedded systems. ACM Transactions on Embedded Computing Systems 10(10) (2011)
2. Aguirre, N., Maibaum, T.: A temporal logic approach to component-based system specification and reasoning. In: Proceedings of the 5th ICSE Workshop on Component-Based Software Engineering. Citeseer (2002)
3. Barnett, M., Leino, K.R.M., Schulte, W.: The Spec# programming system: An overview. In: CASSISD'04, pp. 49–69. Springer (2005)
4. Cheon, Y., Leavens, G., Sitaraman, M., Edwards, S.: Model variables: Cleanly supporting abstraction in design by contract. Software: Practice and Experience 35(6), 583–599 (2005)
5. Chin, W.N., David, C., Nguyen, H.H., Qin, S.: Enhancing modular OO verification with separation logic. In: POPL'08. pp. 87–99. ACM (2008)
6. De Alfaro, L., Henzinger, T.A.: Interface automata. In: ESEC/FSE'01. vol. 26, pp. 109–120. ACM (2001)
7. Distefano, D., Parkinson, M.J.: jStar: Towards practical verification for java. In: OOPSLA'08. pp. 213–226. ACM (2008)
8. Gamma, E., Helm, R., Johnson, R., Vlissides, J.: Design Patterns, Elements of Reusable Object-Oriented Software. Addison-Wesley Reading (1994)
9. Guttag, J.V., Horowitz, E., Musser, D.R.: Abstract data types and software validation. Communications of the ACM 21(12), 1048–1064 (1978)

10. Hong, A., Liu, Y., Qiu, Z.: Axioms and abstract predicates on interfaces in specifying/verifying OO components. Tech. rep., School of Math., Peking University (2013), available at https://github.com/zyqiu/tr/blob/master/OO-components-rep.pdf
11. Leavens, G.T., Baker, A.L., Ruby, C.: Preliminary design of JML: A behavioral interface specification language for Java. ACM SIGSOFT Software Engineering Notes 31(3), 1–38 (2006)
12. Leavens, G.T., Dhara, K.K.: Concepts of behavioral subtyping and a sketch of their extension to component-based systems. In: Leavens, G.T., Sitaraman, M. (eds.) Foundations of Component-Based Systems, chap. 6, pp. 113–135. Cambridge University Press (2000)
13. Leavens, G.T., Leino, K.R.M., Müller, P.: Specification and verification challenges for sequential object-oriented programs. Formal Aspects of Computing 19, 159–189 (2007)
14. Leavens, G.T., Naumann, D.A.: Behavioral subtyping is equivalent to modular reasoning for object-oriented programs. Tech. rep., Department of Computer Science, Iowa State University (2006)
15. Leino, K.R.M.: Dafny: An automatic program verifier for functional correctness. In: LPAR'10. vol. 6355 of LNAI, pp. 348–370. Springer (2010)
16. Liskov, B., Wing, J.M.: A behavioral notion of subtyping. ACM Transactions on Programing Languages and Systems 16(6), 1811–1841 (1994)
17. Liu, Y., Hong, A., Qiu, Z.: Inheritance and modularity in specification and verification of OO programs. In: TASE'11. pp. 19–26. IEEE Computer Society (2011)
18. Liu, Y., Qiu, Z.: A separation logic for OO programs. In: FACS'10. vol. 6921 of LNCS, pp. 88–105. Springer (2012)
19. Lumpe, M., Schneider, J.G.: A form-based meta-model for software composition. Science of Computer Programming 56(1), 59–78 (2005)
20. Parkinson, M.J., Bierman, G.M.: Separation logic, abstraction and inheritance. In: POPL'08. pp. 75–86. ACM (2008)
21. Poetzsch-Heffter, A., Schäfer, J.: Modular specification of encapsulated object-oriented components. In: FMCO'05. vol. 4111 of LNCS, pp. 313–341. Springer (2006)
22. Qiu, Z., Hong, A., Liu, Y.: Modular verification of OO programs with interfaces. In: ICFEM'12. vol. 7635 of LNCS, pp. 151–166. Springer (2012)
23. Qiu, Z., Wang, S., Long, Q.: Sequential μJava: Formal foundations. Tech. Rep. 2007–35, School of Math., Peking University (2007), avaliable at http://www.mathinst.pku.edu.cn/index.php?styleid=2, Preprints
24. Van Staden, S., Calcagno, C.: Reasoning about multiple related abstractions with multistar. In: OOPSLA'10. pp. 504–519. ACM (2010)

A Inference Rules of VeriJ Framework

In this appendix, we give a brief introduction on the inference rules for verifying VeriJ programs. More details can be found in [17, 22].

Basic inference rules are given in **Fig. 10**. We skip explaining many simple rules here. Rules [H-DPRE], [H-SPRE] are key to show our idea that specification predicates have scopes, thus may have multi-definitions crossing the class hierarchy for the polymorphism. If a predicate invoked is in scope (in its class or the subclasses), it can be unfolded to its definition. These rules support hiding implementation details in predicate definition. However, these two rules are different. [H-DPRE] says if r is of type D, then in any subclass of D, $p(r, \overline{r'})$ can be unfolded to the body of p in D. [H-SPRE] is for the static binding, where $\mathsf{fix}(D, \psi)$ (in combine with $D.p(r, \overline{r'})$) gives the *instantiation* of ψ in D (seeing Section 4), and provides a static explanation for ψ. In

$$\text{[H-THIS]}\ \Gamma, T, m \vdash \mathbf{this} : T \qquad \text{[H-SKIP]}\ \Gamma \vdash \{\varphi\}\mathbf{skip}\{\varphi\} \qquad \text{[H-ASN]}\ \Gamma \vdash \{\varphi[e/x]\}x := e; \{\varphi\}$$

$$\text{[H-MUT]}\ \Gamma \vdash \{v = r_1 \wedge e = r_2 \wedge r_1.a \mapsto -\}v.a := e; \{v = r_1 \wedge e = r_2 \wedge r_1.a \mapsto r_2\}$$

$$\text{[H-LKUP]}\ \Gamma \vdash \{v = r_1 \wedge r_1.a \mapsto r_2\}x := v.a; \{x = r_2 \wedge v = r_1 \wedge r_1.a \mapsto r_2\}$$

$$\text{[H-CAST]}\ \Gamma \vdash \{v = r \wedge r <: N\}x := (N)v; \{x = r\} \qquad \text{[H-RET]}\ \Gamma \vdash \{\varphi[e/\mathsf{res}]\}\mathbf{return}\ e; \{\varphi\}$$

$$\text{[H-SEQ]}\ \frac{\Gamma \vdash \{\varphi\}c_1\{\psi\},\ \Gamma \vdash \{\psi\}c_2\{R\}}{\Gamma \vdash \{\varphi\}c_1\ c_2\{R\}} \qquad \text{[H-COND]}\ \frac{\Gamma \vdash \{b \wedge \varphi\}c_1\{\psi\},\ \Gamma \vdash \{\neg b \wedge \varphi\}c_2\{\psi\}}{\Gamma \vdash \{\varphi\}\mathbf{if}\ b\ c_1\ \mathbf{else}\ c_2\{\psi\}}$$

$$\text{[H-ITER]}\ \frac{\Gamma \vdash \{b \wedge I\}c\{I\}}{\Gamma \vdash \{I\}\mathbf{while}\ b\ c\{\neg b \wedge I\}} \qquad \text{[H-FRAME]}\ \frac{\Gamma, C, m \vdash \{\varphi\}c\{\psi\} \quad \mathsf{FV}(R) \cap \mathsf{MD}(c) = \emptyset}{\Gamma, C, m \vdash \{\varphi * R\}c\{\psi * R\}}$$

$$\text{[H-CONS]}\ \frac{\begin{array}{c}\Gamma, C, m \vdash \varphi \Rightarrow \varphi', \quad \Gamma, C \vdash \psi' \Rightarrow \psi \\ \Gamma, C, m \vdash \{\varphi'\}c\{\psi'\}\end{array}}{\Gamma, C, m \vdash \{\varphi\}c\{\psi\}} \qquad \text{[H-EX]}\ \frac{\begin{array}{c}\Gamma, C, m \vdash \{\varphi\}c\{\psi\} \\ r \text{ is free in } \varphi, \psi\end{array}}{\Gamma, C, m \vdash \{\exists r \cdot \varphi\}c\{\exists r \cdot \psi\}}$$

$$\overset{\text{[H-OLD]}}{\frac{\forall \langle\varphi\rangle\langle\psi\rangle \in \Pi(T.m) \bullet \Gamma, T, m \vdash (\overline{z = r} \wedge \varphi[\overline{r}/\overline{z}]) \Rightarrow \psi'}{\Gamma, T, m \vdash \psi'[\mathbf{old}(e)/e]}} \qquad \overset{\text{[H-SPRE]}}{\frac{C <: D, \quad \Phi(D.p(\mathbf{this}, \overline{a})) = \psi}{\Gamma, C, m \vdash D.p(r, \overline{r'}) \Leftrightarrow \mathsf{fix}(D, \psi)[r, \overline{r'}/\mathbf{this}, \overline{a}]}}$$

$$\overset{\text{[H-DPRE]}}{\frac{r : D, \quad C <: D, \quad \Phi(D.p(\mathbf{this}, \overline{a})) = \psi}{\Gamma, C, m \vdash p(r, \overline{r'}) \Leftrightarrow \psi[r, \overline{r'}/\mathbf{this}, \overline{a}]}} \qquad \overset{\text{[H-PDPRE]}}{\frac{r : D, \quad \Phi(D.p(\mathbf{this}, \overline{a})) = \psi}{\Gamma, C, m \vdash p(r, \overline{r'}) \Leftrightarrow \psi[r, \overline{r'}/\mathbf{this}, \overline{a'}]}}$$

Fig. 10. Basic Inference Rules

fact, [H-SPRE] is the typed version of [EXPAND] given in Section 4; [H-DPRE] and [H-PDPRE] are similar but deal with dynamic binding.

Rules related to methods and constructors are given in **Fig. 11**, where we assume a default side-condition that local variables $\overline{y}$ are not free in φ, ψ, that can be provided by renaming. The rules reflect our idea in Section 4.2 and divide three cases in verifying methods. They ensure the behavioral subtyping property in a program.

[H-MTHD1] is for verifying methods with a specification (and a definition). It demands that $C.m$'s body meets its specification, and asks to check the refinement between specification of m in C with each of C's supertypes, if exist. Here we promote Π to type set, thus $\Pi(\mathsf{super}(C))(m)$ gives specifications for m in C's supertypes. If there is no, this check is true by default. [H-MTHD2] is for verifying methods defined in classes without specifications. [H-MINH] is for verifying inherited methods. [H-CONSTR] for constructors is similar. However, a constructor cannot have multi-specifications. Here $\mathsf{raw}(\mathbf{this}, C)$ specifies $\mathbf{this}$ refers to a newly created raw object of type C, and then c modifies its state, where $\mathsf{raw}(r, C)$ has a definition:

$$\mathsf{raw}(r, C) \mathrel{\widehat{=}} \begin{cases} \mathsf{obj}(r, C), & N \textit{ has no field} \\ r : C \wedge (r.a_1 \mapsto \mathsf{nil}) * \cdots * (r.a_k \mapsto \mathsf{nil}), & \text{fields of } C \text{ is } a_1, \ldots, a_k \end{cases}$$

Last two rules are for method invocation and object creation. Note that $T.n$ may have multiple specifications, and we can use any of them in proving client code. Due to the *behavioral subtyping*, it is enough to do the verification by the declared type of variable v. Because [H-INV] refers to only specifications, recursive methods are supported.

Here we see how the information given by the developers affects verification. A method specification is a specific requirement and induces some special proof obligations. It connects the implementation with surrounding world: the implemented interfaces, the superclass, and the client codes. When no specification is given, we need to verify more by considering all the possibilities.

$$\text{[H-MTHD1]}\ \frac{\begin{array}{c} C \text{ has a specification for } m, \quad \Theta(C.m) = \lambda(\overline{z})\{\mathsf{var}\ \overline{y}; c\}, \quad \Pi(C.m) = \langle\varphi\rangle\langle\psi\rangle \\ \Gamma, C, m \vdash \{\mathbf{this}: C \wedge \overline{z = r} \wedge \overline{y = \mathsf{nil}} \wedge \varphi[\overline{r}/\overline{z}]\}c\{\psi[\overline{r}/\overline{z}]\} \\ \Gamma, C \vdash \Pi(\mathsf{super}(C))(m) \sqsubseteq \langle\varphi\rangle\langle\psi\rangle \end{array}}{\Gamma \vdash \{\varphi\}C.m\{\psi\}}$$

$$\text{[H-MTHD2]}\ \frac{\begin{array}{c} C \text{ defines } m \text{ without specification}, \quad \Theta(C.m) = \lambda(\overline{z})\{\mathsf{var}\ \overline{y}; c\} \\ \forall\ \langle\varphi\rangle\langle\psi\rangle \in \Pi(C.m) \bullet \Gamma, C, m \vdash \{\mathbf{this}: C \wedge \overline{z = r} \wedge \overline{y = \mathsf{nil}} \wedge \varphi[\overline{r}/\overline{z}]\}c\{\psi[\overline{r}/\overline{z}]\} \end{array}}{\Gamma \vdash C.m \triangleright \Pi(C.m)}$$

$$\text{[H-MINH]}\ \frac{\begin{array}{c} C \text{ inherits } D.m, \quad \forall\ \langle\varphi\rangle\langle\psi\rangle \in \Pi(C.m) \bullet \Gamma, C \vdash \langle\varphi\rangle\langle\psi\rangle \sqsubseteq \langle\mathsf{fix}(D,\varphi)\rangle\langle\mathsf{fix}(D,\psi)\rangle \\ \forall\ I \in \mathsf{super}(C) \wedge \Pi(I.m) = \langle\varphi'\rangle\langle\psi'\rangle \bullet \Gamma, C \vdash \langle\varphi'\rangle\langle\psi'\rangle \sqsubseteq \Pi(C.m) \end{array}}{\Gamma \vdash C.m \triangleright \Pi(C.m)}$$

$$\text{[H-CONSTR]}\ \frac{\begin{array}{c} \Pi(C.C) = \langle\varphi\rangle\langle\psi\rangle, \quad \Theta(C.C) = \lambda(\overline{z})\{\mathsf{var}\ \overline{y}; c\} \\ \Gamma, C, C \vdash \{\overline{z = r} \wedge \overline{y = \mathsf{nil}}\ \wedge \mathsf{raw}(\mathbf{this}, C) * \varphi[\overline{r}/\overline{z}]\}c\{\psi[\overline{r}/\overline{z}]\} \end{array}}{\Gamma \vdash \{\varphi\}C.C\{\psi\}}$$

$$\text{[H-INV]}\ \frac{\Gamma, C, m \vdash v : T, \quad \langle\varphi\rangle\langle\psi\rangle \in \Pi(T.n)}{\Gamma, C, m \vdash \{v = r \wedge \overline{e = r'} \wedge \varphi[r, \overline{r'}/\mathbf{this}, \overline{z}]\}\ x := v.n(\overline{e})\ \{\psi[r, \overline{r'}, x/\mathbf{this}, \overline{z}, \mathsf{res}]\}}$$

$$\text{[H-NEW]}\ \frac{\Pi(C'.C') = \langle\varphi\rangle\langle\psi\rangle}{\Gamma, C, m \vdash \{\overline{e = r'} \wedge \varphi[\overline{r'}/\overline{z}]\}\ x := \mathbf{new}\, C'(\overline{e})\{\exists r \cdot x = r \wedge \psi[r, \overline{r'}/\mathbf{this}, \overline{z}]\}}$$

Fig. 11. Inference Rules related to Methods and Constructors

B Other Method Verifications

In this appendix, we prove other methods in the given implementation of MVC calculator architecture. As class $View_2$ acts like class $View$ except method $update(MI\ m)$, we only verify this method for $View_2$ here. Inference rules presented in Appendix A and well-supported axioms listed in Section 5 would be used in the below deductions. Also we use the similar labels in Section 5.

Note that some methods need to invoke others, such as $Controller.Controller(m)$ and $VI.update(m)$ calling $MI.getState()$ and $VI.paint()$, $Model.addView(v)$ and $Model.Notify()$ calling $VI.update(m)$, $Model.Multiply(i)$ and $Model.Reset()$ calling $MI.Notify()$, $Controller.resetPerformed()$ calling $MI.Reset()$. Therefore, we should prove the called methods ahead of the calling ones.

First, we prove methods $Model.Model()$, $Model.getState()$ and $View.paint()$ as follows, both of which are basic and call no method in this implementation.

Proving $Model.Model()$:
$\{\mathbf{emp}\}$
$\mathbf{this}.total = 1;$
$\{\mathbf{this}.total \mapsto 1;\}$
$\mathbf{this}.views =$
  $\mathbf{new}\ ArrayList\langle VI\rangle();$
$\{\mathbf{this}.state \mapsto 1 *$
  $\mathbf{this}.views \mapsto \emptyset\}$
$\{model(\mathbf{this}, \emptyset, 1)\}$
[Def. of $Model.model(\ldots)$]

Proving $Model.getState()$:
$\{model(\mathbf{this}, vs, st)\}$
$\{\mathbf{this}.views \mapsto vs *$
  $\mathbf{this}.total \mapsto st\}$
[Def. of $Model.model(\ldots)$]
$s = \mathbf{this}.total;$
$\{s = st \wedge \mathbf{this}.views \mapsto vs * \mathbf{this}.total \mapsto st\}$
$\{s = st \wedge model(\mathbf{this}, vs, st)\}$
[Def. of $Model.model(\ldots)$]
$\mathbf{return}\ \ s;$
$\{\mathsf{res} = st \wedge model(\mathbf{this}, vs, st)\}$

Proving $View.paint()$:
$\{view(\mathbf{this}, m, st)\}$
$\{\mathbf{this}.model \mapsto m * \mathbf{this}.vtotal \mapsto st\}$
[Def. of $View.view(\ldots)$]
$\mathbf{Int}\ v = \mathbf{this}.vtotal;$
$\{\exists v \cdot v = st \wedge \mathbf{this}.model \mapsto m * \mathbf{this}.vtotal \mapsto st\}$
System.out.println(“Product is: ” + v);
$\{\mathbf{this}.model \mapsto m * \mathbf{this}.vtotal \mapsto st\}$
$\{view(\mathbf{this}, m, st)\}$
[Def. of $Model.model(\ldots)$]

Thus these methods are correct. Similar to $\mathit{View.paint}()$, we can prove $\mathit{View}_2\mathit{.paint}()$ is also correct. Because in our current implementation, only class *Model* implements interface *MI*, two classes *View* and View_2 implements *VI*, we can respectively prove another three methods $\mathit{Controller.Controller}(m)$, $\mathit{View.update}(m)$ and $\mathit{View}_2\mathit{.update}(m)$ depending on the correctness of method $\mathit{Model.getState}()$ and $\mathit{View.paint}()$.

```
Proving $Controller.Controller(\mathbf{m})$ :
$\{model(m, vs, st)\}$
$\mathbf{this}.model = m;$
$\{model(m, vs, st)*$
  $\mathbf{this}.model \mapsto m\}$
$\mathbf{this}.state = m.getState();$
$\{model(m, vs, st)*$
  $\mathbf{this}.model \mapsto m*$
  $\mathbf{this}.state \mapsto st\}$
$\{model(m, vs, st)*$
  $controller(\mathbf{this}, m, st)\}$
                    [Def. of $Controller.controller(\ldots)$]
```

```
Proving $View.update(\mathbf{m})$ :
$\{view(\mathbf{this}, m, -) * model(m, vs, st)\}$
$\{\mathbf{this}.model \mapsto m*$
  $\mathbf{this}.vtotal \mapsto -$
  $*model(m, vs, st)\}$
                    [Def. of $View.view(\ldots)$]
if $(m{=}{=}\mathbf{this}.model)$
  $\{(m = m) \wedge \mathbf{this}.model \mapsto m$
    $*\mathbf{this}.vtotal \mapsto -$
    $*model(m, vs, st)\}$
  $\mathbf{this}.vtotal = m.getState();$
  $\{\mathbf{this}.model \mapsto m * \mathbf{this}.vtotal \mapsto st*$
    $model(m, vs, st)\}$
  $\{view(\mathbf{this}, m, st) * model(m, vs, st)\}$
                    [Def. of $View.view(\ldots)$]
$\{view(\mathbf{this}, m, st) * model(m, vs, st)\}$
$\mathbf{this}.paint();$
$\{view(\mathbf{this}, m, st) * model(m, vs, st)\}$
```

```
Proving $View_{\mathbf{2}}.update(\mathbf{m})$ :
$\{view(\mathbf{this}, m, -) * model(m, vs, st)\}$
$\{\mathbf{this}.model \mapsto m * \mathbf{this}.vtotal \mapsto -$
  $*model(m, vs, st)\}$
                    [Def. of $View_2.view(\ldots)$]
if $(m{=}{=}\mathbf{this}.model)\{$
  $\{(m = m) \wedge \mathbf{this}.model \mapsto m*$
    $\mathbf{this}.vtotal \mapsto - * model(m, vs, st)\}$
  $\mathbf{this}.vtotal = m.getState() + 1;$
  $\{\mathbf{this}.model \mapsto m * \mathbf{this}.vtotal \mapsto (st + 1)*$
    $model(m, vs, st)\}$
  $\{view(\mathbf{this}, m, st) * model(m, vs, st)\}$
                    [Def. of $View_2.view(\ldots)$]
$\{view(\mathbf{this}, m, st) * model(m, vs, st)\}$
$\mathbf{this}.paint();$
$\{view(\mathbf{this}, m, st) * model(m, vs, st)\}$
```

Thus, methods $\mathit{Controller.Controller}(m)$ and $\mathit{VI.update}(m)$ are correct too. At last, we prove $\mathit{Model.Notify}()$ and $\mathit{Model.addView}(v)$ by using $\mathit{VI.update}(m)$, and then the rest two complicated methods and $\mathit{Model.Reset}()$ by $\mathit{MI.Notify}()$.

```
Proving $Model.Notify()$ :
$\{model(\mathbf{this}, vs, st)*$
  $(\circledast_{v \in vs} view(v, \mathbf{this}, st')))\}$
for$(VI\ v : views)\{$
  $\{\exists\, vs_1, v, v_1, v_2, vs_2 \cdot (vs = vs_1 \cup \{v\} \cup vs_2)\wedge$
    $model(\mathbf{this}, vs, st) * (\circledast_{v_1 \in vs_1}$
    $view(v_1, \mathbf{this}, st)) * view(v, \mathbf{this}, st')*$
    $(\circledast_{v_2 \in vs_2} view(v_2, \mathbf{this}, st'))\}$
  $v.update(\mathbf{this});$
  $\{\exists\, vs_1, v, v_1, v_2, vs_2 \cdot (vs = vs_1 \cup \{v\} \cup vs_2)\wedge$
    $model(\mathbf{this}, vs, st) * (\circledast_{v_1 \in vs_1}$
    $view(v_1, \mathbf{this}, st)) * view(v, \mathbf{this}, st)*$
    $(\circledast_{v_2 \in vs_2} view(v_2, \mathbf{this}, st'))\}$
  $\{\exists\, vs_1', v', v_1', v_2', vs_2' \cdot (vs = vs_1' \cup \{v'\} \cup vs_2')\wedge$
    $(vs_1' = vs \cup \{v\}) \wedge (vs_2 = vs_2' \cup \{v\})\wedge$
    $model(\mathbf{this}, vs, st) * (\circledast_{v_1' \in vs_1'}$
    $view(v_1', \mathbf{this}, st)) * view(v', \mathbf{this}, st')*$
    $(\circledast_{v_2' \in vs_2'} view(v_2', \mathbf{this}, st'))\}$
}
$\{model(\mathbf{this}, vs, st)*$
  $(\circledast_{v \in vs} view(v, \mathbf{this}, st))\}$
$\{MVs(\mathbf{this}, vs, b)\}$
                    [Def. of $Model.MVs(\ldots)$]
```

```
Proving $Model.addView(\mathbf{v})$ :
$\{model(\mathbf{this}, vs, st) * view(v, \mathbf{this}, -)\}$
$\{\mathbf{this}.views \mapsto vs * \mathbf{this}.total \mapsto st*$
  $view(v, \mathbf{this}, -)\}$
                    [Def. of $Model.model(\ldots)$]
$views.add(v);$
$\{\mathbf{this}.views \mapsto (vs \cup \{v\})*$
  $\mathbf{this}.total \mapsto st * view(v, \mathbf{this}, -)\}$
$\{model(\mathbf{this}, vs \cup \{v\}, st)*$
  $view(v, \mathbf{this}, -)\}$
                    [Def. of $Model.model(\ldots)$]
$v.update(\mathbf{this});$
$\{model(\mathbf{this}, vs \cup \{v\}, st)*$
  $view(v, \mathbf{this}, st)\}$
```

Proving $Model.Reset()$:
$\{MVs(\mathbf{this}, vs, st)\}$
$\{model(\mathbf{this}, vs, st)*$
$(\circledast_{v \in vs} view(v, \mathbf{this}, st))\}$
[Def. of $Model.MVs(\ldots)$]
$\{\mathbf{this}.views \mapsto vs * \mathbf{this}.total \mapsto st*$
$(\circledast_{v \in vs} view(v, \mathbf{this}, st))\}$
[Def. of $Model.model(\ldots)$]
$\mathbf{this}.total = 1;$
$\{\mathbf{this}.views \mapsto vs * \mathbf{this}.total \mapsto 1*$
$(\circledast_{v \in vs} view(v, \mathbf{this}, st))\}$
$\{model(\mathbf{this}, vs, 1)*$
$(\circledast_{v \in vs} view(v, \mathbf{this}, st))\}$
[Def. of $Model.model(\ldots)$]
$\mathbf{this}.Notify();$
$\{model(\mathbf{this}, vs, 1)*$
$(\circledast_{v \in vs} view(v, \mathbf{this}, 1))\}$
$\{MVs(\mathbf{this}, vs, 1)\}$[Def. of $Model.MVs(\ldots)$]

Proving $Model.Multiply(\mathbf{i})$:
$\{\exists j \cdot j = i \wedge MVs(\mathbf{this}, vs, st)\}$
$\{\exists j \cdot j = i \wedge model(\mathbf{this}, vs, st)*$
$(\circledast_{v \in vs} view(v, \mathbf{this}, st))\}$
[Def. of $Model.MVs(\ldots)$]
$\{\exists j \cdot j = i \wedge \mathbf{this}.views \mapsto vs*$
$\mathbf{this}.total \mapsto st*$
$(\circledast_{v \in vs} view(v, \mathbf{this}, st))\}$
[Def. of $Model.model(\ldots)$]
$\mathbf{Int} vl = \mathbf{this}.total \star i;$
$\{\exists\, j, st' \cdot j = i \wedge vl = st' \wedge st' = (st \star i)$
$\wedge \mathbf{this}.views \mapsto vs * \mathbf{this}.total \mapsto st*$
$(\circledast_{v \in vs} view(v, \mathbf{this}, st))\}$
$\{\exists\, j, st' \cdot j = i \wedge vl = st' \wedge$
$st' = product(st, i) \wedge \mathbf{this}.views \mapsto vs*$
$\mathbf{this}.total \mapsto st * (\circledast_{v \in vs} view(v, \mathbf{this}, st))\}$
[Def. of $Model.product(\ldots)$]
$\mathbf{this}.total = vl;$
$\{\exists\, j, st' \cdot j = i \wedge vl = st' \wedge$
$st' = product(st, i) \wedge \mathbf{this}.views \mapsto vs*$
$\mathbf{this}.total \mapsto st' * (\circledast_{v \in vs} view(v, \mathbf{this}, st))\}$
$\{\exists\, j, st' \cdot j = i \wedge st' = product(st, i) \wedge$
$model(\mathbf{this}, vs, st')*$
$(\circledast_{v \in vs} view(v, \mathbf{this}, st))\}$
[Def. of $Model.model(\ldots)$]
$\mathbf{this}.Notify();$
$\{\exists\, j, st' \cdot j = i \wedge st' = product(st, i) \wedge$
$MVs(\mathbf{this}, vs, st')\}$
$\{\exists\, st' \cdot st' = product(st, i) \wedge$
$MVs(\mathbf{this}, vs, st')\}$

Thus all declared methods in given implementation are correct. In conclusion, the implementation is correct for the MVC calculator design.