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 Mathematical Sciences, Peking University
{hongali,liuyijing,qzy}@math.pku.edu.cn

Abstract. Abstraction is extremely important in component-based design and implementation of systems; however, it discourages specifying and verifying behaviors of components and their interactions. In this paper we develop a framework which shows how the interfaces of components and their interactions can be specified abstractly and how the verification can be done. We show also how the abstract specification on the interface level can be used to enforce the correct implementation of the components. We take the well-known MVC architecture in here as a case study. Although our work focus on the OO based programs, some concepts and techniques developed 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 relative methodologies emphasize abstraction, interaction based on clear and abstract interfaces, separation of interfaces from components, interchangeability of components, etc. The main ideas of the techniques include 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 makes twofold benefits: On one side, the clients are designed only based on interfaces of the components which they depend on, 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 have wider design choices.

However, although the techniques are very useful and effective in supporting good component design and flexible integration, they are also obstacles to formal specification and verification of CBSs. In common practice, interface declarations provide only syntactic and typing information. For verifying behaviors of systems, we must include semantic specifications for interfaces. How and in which form the specifications are provided will become a serious problem here, due to an obvious dilemma:

- We need to protect the abstraction provided by the component interfaces, thus the specifications should not leak any unnecessary details of the implementation.
- We need the specifications to provide enough information for the behaviors of the components, to support the reasoning about their clients.

In addition, if the specifications involve real details, it will exclude 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 are expressed upon abstract predicates (named *specification predicates*) for each method, and a group of necessary *axioms* over the predicates which give constraints on the implementations and describe relations over different predicates, thus over methods and classes.

Based on our previous work [18], we build the theoretical foundation for the approach, and define how a program with these specifications is correct. We give some new rules for reasoning programs. Combining with the inference rules of a separation logic for OO programs [14], we have a more complete inference system for OO based component systems. With it, we can prove if a group of classes forms a correct implementation of a group of relative interfaces. In addition, we can also support the proof for client codes which depend only on the interfaces. As the proof involves no information from the implementation, modular verification is very-well achieved.

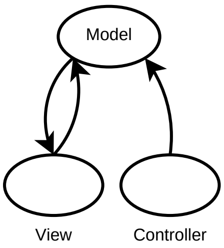
We show our ideas and approach using a simplified version of MVC architectures which is built upon closely co-related interfaces and classes. We illustrate how the components with collaborations are abstracted, how they are specified with axioms on interfaces, and how to prove their implementations and support the proof for client codes.

In the rest of the paper, we will analyze the problems 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 illustrate our approach with a simple MVC architecture in Section 5. At last, we discuss some related work and conclude in Section 6, and give the basic inference rules of the framework in Appendix A. In addition, we show the detail verification of the rest methods in given MVC implementation in Appendix B.

2 Abstractly Specify/Verify Co-related Components: Problem

To give an impression for the problem of the study, we use a simple MVC architecture with interfaces following the informal requirements and interaction structure, as depicted in **Fig. 1** (left and middle). For designing MVC, we give three interface declarations (right), where, for example, *MI* for the interface of model, and each provides some methods (here only signatures). As an example to show the problem, the model here has a simple integer state, and we have one simple update method, etc. All these aspects can be extended without affecting our discussion in the following. Here the components are closely related: views register on some model, the controller deals with user inputs and asks its model to update the state, the model then notifies all registered views for its state change, and the views need to get model's state when been notified, etc. But, how can we specify these independent of any implementations?

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 and passes it for updating the model that has transitive effects to all the related views.



```
interface MI {
  void addView( VI v);
  Int getState();
  void update(Int b);
}
interface VI {
  void paint(MI m);
}
interface CI{
  void userInput(Int b);
}
```

Fig. 1. MVC Architecture and its Component Interfaces

To specify the MVC without concrete implementation, we follow the ideas of abstract predicates [5, 13, 16] and separation logic, and introduce some predicate(s). We introduce predicate $model(m, vs, st)$ to assert that model m has a registered view list vs and a state st, $view(v, m, st)$ asserting view v has registered on m and its state is st, and $controller(c, m, st)$ asserting controller c monitors m and its state is st. We specify the MVC interfaces as given in **Fig. 2** 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 instead of more common "**requires** *pre*; **ensures** *post*;" pairs to represent pre/post conditions. 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 list of the model, and its state will be updated following the model. The specification of $getState$ means that it simply returns the state value of the model. For method $update$, things become more complex, because the method will not only affect the model, but also its related views. We introduce a predicate $MVs(m, vs, st)$ to assert the state of a bundle of one model and its related views, thus $update$ modifies this state as desired. The only method $paint$ in VI is called by a model that will flush the view's state and cause some other actions (the view painting). At last we consider

```
interface MI {
  void addView( VI v)
  ⟨model(this, vs, st) ∗ view(v, this, -)⟩
  ⟨model(this, vs ∪ {v}, st) ∗
    view(v, this, st)⟩;
  Int getState() ⟨model(this, vs, st)⟩
  ⟨model(this, vs, st) ∧ res = st⟩;
  void update(Int b)
  ⟨MVs(this, vs, -)⟩ ⟨MVs(this, vs, b)⟩;
}
```

```
interface VI {
  void paint(MI m)
  ⟨view(this, m, -) ∗ model(m, vs, st)⟩
  ⟨view(this, m, st) ∗ model(m, vs, st)⟩;
}
interface CI{
  void userInput(Int b)
  ⟨MVC(this, m, vs, -)⟩
  ⟨MVC(this, m, vs, b)⟩;
}
```

Fig. 2. Interfaces with Formal Specifications for MVC Arch.

```
void buildviews (MI m, CI c) $\langle \exists i \cdot \mathit{model}(m, \emptyset, i) * \mathit{controller}(c, m, i) \rangle$
                             $\langle \exists r_1, r_2 \cdot \mathit{MVC}(c, m, \{r_1, r_2\}, 5) \rangle$ {
    $\mathit{VI}\ v_1 = \mathbf{new}\ \mathit{View}(m);$        // add a new view to model $m$
    $c.\mathit{userInput}(5);$                // process a user input
    $\mathit{VI}\ v_2 = \mathbf{new}\ \mathit{View}_2(m);$      // add another view to model $m$
    $v_1.\mathit{paint}(m);$                  // paint one view of the model
}
```

Fig. 3. A Client Procedure Using the MVC Architecture

```
(1)    $\{\mathit{model}(m, \emptyset, i) * \mathit{controller}(c, m, i)\}$
(2)    $\mathit{VI}\ v_1 = \mathbf{new}\ \mathit{View}(m);$
(3)    $\{\mathit{model}(m, \{v_1\}, i) * \mathit{view}(v_1, m, i) * \mathit{controller}(c, m, i)\}$
               $\Downarrow$ ???
(3′)   $\{\underline{\mathit{MVC}(c,\ \overset{[1]}{\ldots})\ \overset{[2]}{\ldots}}\}$
(4)    $c.\mathit{userInput}(5);$
   ......
```

Fig. 4. Verification of the Client Code

$\mathit{userInput}$ in CI which can be called by clients of the MVC components. It brings also problems, because it will affect all components here. To specify states of this bundle of components, we introduce another predicate $\mathit{MVC}(c, m, vs, st)$ to assert that we have a bundle of a controller c, a model m, a list of views vs with internal state st. These specifications go the similar way as shown in literature, e.g. [6], and ours [18].

Having the interface declarations, we can go ahead to define classes to implement them, and thus construct concrete MVC instance(s), and write client codes to use the implementation. We postpone the implementation for a moment, and consider some client codes and their verifications in the first. Because interfaces should be a fence for hiding implementation details, on the semantic side, we should support verification of client codes with only well-specified interfaces.

Fig. 3 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 $\exists r_1, r_2 \cdot \mathit{MVC}(c, m, \{r_1, r_2\}, 5)$. View and View_2 are classes implementing VI. For the constructor of View, we assume it satisfies a specification $\{\mathit{model}(m, vs, st)\}\, \mathit{View}(m) \{\mathit{model}(m, vs \cup \{\mathbf{this}\}, st) * \mathit{view}(\mathbf{this}, m, st)\}$, here $\mathbf{this}$ refers to the new created object which is assigned to variable v by "$v = \mathbf{new}\ \ View(m);$". It is similar for $View_2$.

Now we consider its verification, and list a part of reasoning in **Fig. 4**. The constructions go well, then we meet problems in line (3). To verify the invocation $c.\mathit{userInput}(5)$, according to the specification of $\mathit{userInput}(b)$ in interface CI, we need to check whether the current program state satisfies $\mathit{MVC}(\ldots)$ with some parameters. However, with the abstract specifications, we cannot deduce out a $\mathit{MVC}(\ldots)$ assertion from an assertion building of predicate symbols $\mathit{model}(\ldots)$, $\mathit{view}(\ldots)$, $\mathit{controller}(\ldots)$. Neither can we know what are the things to fill segments $\overset{[1]}{\ldots}$ and $\overset{[2]}{\ldots}$, then we cannot go ahead. This means clearly that something is missed in our specifications.

The problem comes from that our specifications for the interface-level can only use abstract predicates, with abstract symbols as their names and relative signatures which carry no enough information. Although we have some intentions for each of them, the abstract expressions cannot reveal them in method specifications. However, on the other side, we cannot expose definitions for these predicates because the definitions should reflect something about the implementations that have not presented yet. In addition, there may be multiple implementations for a given interface.

For reasoning about a set of OO components like here, such problems are common. Because first, we want to have interfaces independent of implementations to support flexible system designs and replaceable components, then the specifications can be written on some abstract level. However, we want also to verify client codes based on the specifications that ask for more information about the components. This seems a dilemma. In the following, we will present our ideas, and develop a framework based on the concepts of *axioms* and *specification predicates* to resolve this kind of problem.

3 Axioms and Languages

In our framework, a specification predicate is abstract on the interface level, which may have definition(s) in the classes that implement the interface. On the other hand, an *axiom* is a logic statement expressed based on constants, logical variables, predicates combined by logical connections and qualifiers. It gives restrictions and/or relations over abstract predicates. Similar to the situation in first-order logic, a set of axioms defines what is its "model". Here a "model" should be a set of class definitions with specification, 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 separation logic), the inconsistency is generally determinable. We can also define non-redundancy for a set of axioms, however, that is not important and we omit it.

To conduct the interface design for a component-based system, we declare a set of interfaces to outline the system, and specify methods using predicate-based pre and post conditions. Thus we can use axioms to restrict/relate the abstract predicates, which put further restrictions on the implementations. How to choose the predicates and axioms is the matter of the designers, and their thoughts for the system informal requirements. Clearly, the axioms will be general properties of the later-coming implementations. As we said before, one important role of the axioms is to support abstract-level reasoning for the client code. In this aspect, two forms of axioms are the most important: implications and equivalences, because they support the *substitution rule* in reasoning.

Now we introduce our assertion and programming language VeriJ with brief explanations.

The assertion language of VeriJ is a separation logic revised to fit the needs of OO programs, as given in **Fig. 5**. Here we have variables (x), constants, numeric and

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

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

Fig. 5. VeriJ Assertion and Programming Language

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 logic and separation logic connectors, and qualifiers. Some OO specific assertion forms are included, where $r : T$ and $r <: T$ assert the object which r refers to is exactly of the type T or a subtype of T; and $\mathsf{obj}(r, T)$ asserts the heap contains exactly an object of type T and r refers to it. We use over-lined form to represent sequences, as in predicate application $p(\overline{r})$. We may extend it with set or other mathematical notations as needed.

VeriJ is a Java-like language with specifications, especially predicate/axiom definitions. We use C to denote a class name, I an interface name, a and m are field and method names respectively, and omit access control issues. Here are some explanations:

- $\mathbf{def}\ p(\mathbf{this}, \overline{x})$ introduces a specification predicate with signature $p(\mathbf{this}, \overline{x})$, and in class its body ψ must be given to form a definition. Here $\mathbf{this}$ will always be explicitly written as the first parameter to denote the current object. Predicates are used in method specifications and axioms to provide data abstraction. 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.
- $\mathbf{axiom}\ \psi$ introduces an axiom ψ into the global scope. No program variables are allowed in axioms, while the variables are implicitly universal-quantified, and can be instantiated in axiom application. We will give more details in Section 4.
- π is a pair of specification for constructors or methods with a brief form $\langle \mathit{pre}\rangle\langle \mathit{post}\rangle$. 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 same methods of its super-interface.
- 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 a predicate, it must define this predicate with a body or inherit a body 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, needs to be built to record useful information in G. We need also type-checking specification parts of programs, e.g., the body of a predicate definition involves only its parameters; In an axiom, each predicate must be 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. For these, we may introduce types into the specifications. The environment construction and type-checking are routine and we ignore them here. We will only consider well-typed programs below, and assume that 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; $\Phi(C.p(\mathbf{this}, \overline{x}))$ 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, unrelated predicates will always be given different names[1], thus, if several definitions for the same predicate name p appear in different classes, they are local definitions for p fitting the needs of each individual class. In addition, we assume all definitions for p have the same signature, i.e., no overloading.

4 Verifying Programs wrt Axioms and Method Specifications

In this section, we develop a framework for reasoning about VeriJ programs with specifications, especially interface-based design and axioms. With basic inference rules in our former work [14, 18] (seeing Appendix A), we extend the work here.

4.1 Verifying Implementations wrt Axioms

Because of the existence of subclass overriding and multi-implementing classes for the same interface, multiple definitions for the same predicate are common in OO programs. On the other hand, axioms are global properties/requirements over the program. To judge whether a group of classes really obey a set of axioms, we should define clearly what the predicate applications denote in axioms. We can use static environment Γ to get the predicate definitions. However, not as usual, now a predicate p may have multiple definitions, thus we must define what to use in unfolding the predicate application. 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 its application $p(r, \overline{r'})$ in axioms as a substitution:*

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

[1] This constraint can be achieved by suitable renaming.

where:

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

We substitute predicate application $p(r,\overline{r'})$ in axioms by a disjunction of formulas, while each consists of a type conjunct $(r : C_j)$ 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 and 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:

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

For a predicate set Ψ, we can define a substitution for Ψ based on all the substitutions for $p \in \Psi$. Based on **Definition 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 which mentions only types and relative predicates defined in N. We say that ψ is* well supported *by N, if we have $\Gamma_N \models \psi\delta_{\mathsf{preds}(\psi),\Gamma_N}$, where environment Γ_N provides predicate definitions, and $\delta_{\mathsf{preds}(\psi),\Gamma_N}$ is the substitution built from the set of predicates occurring in ψ according to* **Definition 2***, which is 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 $\Gamma_N \models \psi$. 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 $\psi \in \mathcal{A}$.*

Then, we define whether a program is *well-axiom-constrained* as follows:

Definition 4 (Well-Axiom-Constrained Program). *Suppose $G = (\overline{A};\ N)$ is a program, with interface/class declaration sequence N and axiom set $\mathcal{A}$. We say G is a well-axiom-constrained program if $\Gamma_N \models \mathcal{A}$.*

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 [14]. The rule set contains basic inference rules for FOL and separation logic (of course, it is not complete as the original separation logic). 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 that, 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 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. In the first, we can verify axioms one by one, or in groups:

Lemma 1. *Assume $\Gamma_N \vdash \Psi$ and $\Gamma_N \vdash \Psi'$, then we have $\Gamma_N \vdash \Psi \cup \Psi'$.*

Proof. By **Def. 3** in a program, we clearly see that each axiom can be proven independently of other axiom group according to the static environment Γ_N. Thus if N supports two axiom groups Ψ, Ψ', N also support their union set. □

Note that, Lemma 1 also tells us, because of the independence of proving disjoint axiom groups, any additional axioms indeed only bring proof obligations on themselves but nothing on already well-supported axioms of a program. 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 $N \nmid \Psi$ to mean N contains no declaration/definition of predicates in Ψ.

Lemma 2. *(1) Assume $\Gamma_N \vdash \Psi$, and N' is a sequence of fresh interface declarations. If NN' is still a well-typed declaration sequence, we have $\Gamma_{NN'} \vdash \Psi$.*

(2) If $\Gamma_N \vdash \Psi$, for a sequence of interface/class declarations N' where $N' \nmid \Psi$ and $N\,N'$ is still a well-typed declaration sequence, we have $\Gamma_{NN'} \vdash \Psi$.

(3) If $\Gamma_N \vdash \Psi$ and $\Gamma_{N'} \vdash \Psi'$, where $N \nmid \Psi'$ and $N' \nmid \Psi$, and $N\,N'$ is still a well-typed declaration sequence, we have $\Gamma_{NN'} \vdash \Psi \cup \Psi'$.

Proof. For (1), as we assume each predicate declared for a well-typed program has a distinct name, and interfaces give no definition for declared predicates, thus N' has no semantic effect on the well-supported axiom set Ψ for N. So we trivially have $\Gamma_{NN'} \vdash \Psi$ without any more proof obligation.

For (2), it is trivially true because N' provides no new semantic interpretations for axioms in Ψ thus makes no effect on their truth value.

For (3), we can simply apply the case (2) above in this lemma and Lemma 1 to conclude so. □

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 can call such classes in N' *simple clients* of N. Clearly, all the axioms in Ψ are true assertions for them and may be used in their verification. If some *client* classes use existing classes and support another independent set of axioms, they fall into the coverage of the last case in Lemma 2. As the result, the new set of classes (with their axioms) can be simply combined with the existing classes (and the existing axioms), because the verification of well-supportness does not touch 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 the *proper* but *correct* because we do not verify the methods in the classes according to their specifications yet.

The following definition tells us how to check an extended class declaration is *proper*.

Definition 5 (Proper Class Extension). *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 $\Gamma_{N\,K} \vdash \Psi_1$. □

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 $\Gamma_N \vdash \Psi$ and N' is a sequence of class/interface declarations. In addition, for any class declaration K in N' such that $N' = N'_1\,K\,N'_2$, we know K is a proper extension class with respect to $(\Psi, N\,N'_1)$, then we have $\Gamma_{N\,N'} \vdash \Psi$.

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 also have $\Gamma_{N\,K} \vdash (\Psi \setminus \Psi_1)$ by the case (2) in Lemma 2. Therefore, we easily 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 easily proven by applying the above case (1) in this lemma inductively on the position of K in sequence N'. □

In conclusion, the Lemmas 2, 3 given above 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 extending subclass may need to check supporting the specified axioms as necessary. Rather than extending a program, we may modify some of its original implementation. Then if the new implementation modifies definitions of some predicates appearing in certain axioms, those related axioms should be firstly reverified to be well-supported in proving the correctness of the new 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, while we should verify that each method satisfies its specifications, i.e., to prove the components are correctly implemented. We have developed a set of rules for the verification, which are listed in Appendix A with brief explanations.

Here are also subtleties, because of the existence of interfaces, multi-implementation, and inheritance. A class definition takes generally the form:

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

where C inherits B as its superclass and implements interfaces $I_1, \ldots, I_m$. For the m, it can be a new one, or one overriding another method m accessible in B; and it can be defined with the 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).

An available method in class C can have an explicit specification in C, thus have a definition, or inherited specifications from C's supertypes but might a definition, or an inherited definition with also inherited specification from its superclass B. 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 it fits the need of the superclass and the interfaces being implemented. We consider them in the following, and introduce some notations and definitions first.

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 use $T <: T'$ as the transitive and reflective closure of super. We use $\mathsf{super}(C)$ to get all supertypes of C, thus for example (3), $\mathsf{super}(C) = \{I_1, \ldots, I_m, B\}$.

When C implements interfaces $I_1, I_2, \ldots$, and defines method m without giving a specification, m in C may have multiple specifications if more than one of the interfaces have specifications for m. We will 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 $\Gamma, C, m \vdash \psi$ to state that ψ holds in method m of class C under Γ. Clearly, here ψ must be a state-independent formula. We use $\Gamma, C, m \vdash \{\varphi\}c\{\psi\}$ to say that command c in m of C satisfies the pair of precondition φ and postcondition ψ. We write $\Gamma \vdash \{\varphi\}C.m\{\psi\}$ (or $\Gamma \vdash \{\varphi\}C.C\{\psi\}$) to state that $C.m$ (constructor of C) is correct wrt $\langle\varphi\rangle\langle\psi\rangle$ under Γ. For methods with multiple specifications, we use $\Gamma \vdash C.m \rhd \mathit{\Pi}(C.m)$ 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 Γ, C,* iff *there exists an assertion R which is free of program variables, such that $\Gamma, C \vdash (\varphi_1 \Rightarrow \varphi_2 * R) \wedge (\psi_2 * R \Rightarrow \psi_1)$. We use $\Gamma, C \vdash \langle\varphi_1\rangle\langle\psi_1\rangle \sqsubseteq \langle\varphi_2\rangle\langle\psi_2\rangle$ to denote this fact.*

For 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 [12] defined the condition for specification refinement as $\varphi_1 \Rightarrow \varphi_2 \wedge \psi_2 \Rightarrow \psi_1$. The above definition is its extension by considering the storage extension and multiple specifications. It follows also the *nature refinement order* proposed in [11].

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 $\Gamma, I \vdash \langle\varphi'\rangle\langle\psi'\rangle \sqsubseteq \langle\varphi\rangle\langle\psi\rangle$. This verification is done only on the logic level in our framework, 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 method in a program 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 obligations. 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 ***Definition 4****.*
(2) Each class C defined in G is correct according to ***Definition 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 Experiments

Now, having the enriched specification language and verification framework with axioms of VeriJ, we will respecify and reexamine the MVC architecture discussed in Section 2, to see how the problems mentioned there can be tackled relatively naturally and also the two aspects of axiom's effects in this section.

5.1 Specifying the MVC Architecture

Following the outline given in **Fig. 1**, we have declared interfaces MI, CI, VI in **Fig. 2** to embody the system 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 interface but we omit it to save space.) In interfaces, these predicates including the parameters are abstract symbols and their concrete meaning which may be multiple will be given in later implementations of these interfaces.

However, not all definitions for these predicates are acceptable, and some predicates may have interrelations with one another. In order to reflect our anticipation in correctly specifying informal design requirements, preventing wrong implementations of MVC and providing enough information for client verifications, we need to constrain definitions of these predicates in later implementations and their correct uses in

```
class Model ▷ MI{
  def model(this, vs, st) :
    this.state ↦ st ∗ this.views ↦ vs;
  def MVs(this, vs, st) :
    model(this, vs, st) ∗ (⊛_{v∈vs} view(v, this, st));
  Int state;    List⟨VI⟩views;
  Model()⟨emp⟩⟨model(this, ∅, 0)⟩
  { this.state = 0;
    this.views = new ArrayList⟨VI⟩(); }
  Int getState(){ Int s;
    s = this.state; return s; }
  void addView(VI v){
    views.add(v); v.paint(this); }
  void update(Int b){
    this.state = b;
    for(VI v : views){v.paint(this); }
  }
}
class View ▷ VI{
  def view(this, m, st) :
    this.model ↦ m ∗ this.state ↦ st;
  MI model; Int state;
  View(MI m) ⟨model(m, vs, st)⟩
  ⟨model(m, vs ∪ {this}, st) ∗ view(this, m, st)⟩
  { this.model = m; this.state = 0;
    m.addView(this); }
  void paint(MI m){
    if (m==this.model){
      this.state = m.getState();
      System.out.println(state); }
  }
}
class View_2 ▷ VI{
  def view(this, m, st) :
    this.model ↦ m ∗ this.state ↦ st;
  MI model; Int state;
  View_2(MI m)⟨model(m, vs, st)⟩
  ⟨model(m, vs ∪ {this}, st)∗
    view(this, m, st)⟩ {...}
  void paint(MI m){ ... }
}
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 userInput(Int b){
    this.state = b; model.update(b); }
}
```

Fig. 6. An Implementation of the MVC Component Interfaces

specifications by revealing their relations or properties of individual one. Thus applying our approach in section 4, we additionally specify a set of axioms labeled as [a1-a3] as follows according to the informal requirements in **Fig.1**.

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

The axioms form a part of specifications to capture important interactions or properties of the MVC, and constrain the forthcoming implementations. Semantically, any implementation of the MVC should fulfill them, and the implementations of the methods declared in the interfaces must obey these constraints which will produce some proof obligations. In this way, although the interfaces provide no behavior definitions, their implementations have been connected formally by the predicates and axioms.

Fig. 6 gives four classes *Model*, *Controller*, *View* and View_2 which implement the interfaces and form a MVC instance. All predicates declared in the interfaces are defined with bodies here that give also specific meaning for the axioms. For example, axiom [a1] tells the whole MVC architecture can be divided into a model object, its controller object and a view-object list; [a2] means the model-views aggregate structure consists of a model object and a view-object list; and [a3] says the whole MVC can also be viewed as consisting of a model-views aggregate structure with a controller object.

Having this implementation, we consider its formal verification before concluding its correctness and using its specifications for verifying client codes. **Definition 8** gives two parts of work for the correctness of the implementation: (1) checking it supporting axioms [a1-a3]; (2) checking each declared method satisfying its specifications;

5.2 Verifying Implementations with Axioms and Method Specifications

First, we verify the well-supported property of each axiom wrt the implementation by applying the two-step procedure given in Section 4. Second, we prove each method according to its specifications. Due to limited space, we only give the detailed proofs of axioms and two methods here, and leave other proofs in Appendix B.

For axiom [a1], we construct a substitution 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 $controller(\ldots)$. Then applying this substitution on the assertion of [a1] (we simply denote it as ψ), we get the following logic formula (4) to prove,

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

To prove (4), we use definition of fix and inference rules, and get

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

and similar for $\mathsf{fix}(Controller, controller(\ldots))$, then formula (4) becomes:

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

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

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

Thus, we know that [a1] is well supported. For other axioms, we will give only the proof processes without more explanations as below.

For axiom [a2], we can construct a substitution

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

and then have two respective deductions for two direction implications of [a2] as follows:

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

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

Because (L-a2) equals to (R-a2), axiom [a2] is proven to be well supported under given Γ.

For axiom [a3], we also apply the substitution $\delta_{\{\mathit{MVs},model\},\Gamma}$ constructed for proving [a2], and have:

$$\begin{array}{ll}
\quad \mathit{MVs}(m, \emptyset, st)\delta_{\{\mathit{MVs},model\},\Gamma} & \\
\Leftrightarrow m : \mathit{Model} \wedge \mathsf{fix}(\mathit{Model}, \mathit{MVs}(m, \emptyset, st)) & \text{[Def. 2]} \\
\Leftrightarrow \mathit{Model}.\mathit{MVs}(m, \emptyset, st)\ hfill\text{[Def. of } \mathsf{fix}(D,\psi)\text{]} & \\
\Leftrightarrow \mathsf{fix}(\mathit{Model}, \Phi(\mathit{Model}.\mathit{MVs}(\mathbf{this}, vs, st)))[m, \emptyset/\mathbf{this}, vs] & \text{[Rule [EXPAND]]} \\
\Leftrightarrow \mathsf{fix}(\mathit{Model}, model(\mathbf{this}, vs, st) * (\circledast_{v \in vs} view(v, \mathbf{this}, st)))[m, \emptyset/\mathbf{this}, vs] & \\
 & \text{[Def. of } \Phi(\mathit{Model}.\mathit{MVs}(\ldots))\text{]} \\
\Leftrightarrow (\mathit{Model}.model(\mathbf{this}, vs, st) * (\circledast_{v \in vs} view(v, \mathbf{this}, st)))[m, \emptyset/\mathbf{this}, vs] & \\
 & \text{[Def. of } \mathsf{fix}(D,\psi)\text{]} \\
\Leftrightarrow \mathit{Model}.model(m, \emptyset, st) * (v = \mathsf{rnull} \wedge view(v, m, st)) & \\
\Leftrightarrow \mathit{Model}.model(m, \emptyset, st) * \mathbf{emp} & \\
\Leftrightarrow \mathit{Model}.model(m, \emptyset, st) & \text{[(L-a4)]}
\end{array}$$

$$\begin{array}{ll}
\quad model(m, \emptyset, st)\delta_{\{\mathit{MVs},model\},\Gamma} & \\
\Leftrightarrow m : \mathit{Model} \wedge \mathsf{fix}(\mathit{Model}, model(m, \emptyset, st)) & \text{[Def. 2]} \\
\Leftrightarrow \mathit{Model}.model(m, \emptyset, st) & \text{[Def. of } \mathsf{fix}(D,\psi)\text{] [(R-a4)]}
\end{array}$$

Because (L-a3) equals to (R-a3), axiom [a3] is well supported under given Γ.

In conclusion, each specified axiom can be independently well-supported by our 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 applied in verifications of the implementation and its 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 proofs of methods $View.View(m)$ and $Controller.userInput(b)$ below because they are used in the illustrating client method.

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

Proving $Controller.userInput(\mathbf{b})$:
$\{MVC(\mathbf{this}, m, vs, -)\}$
$\{\exists\, st \cdot MVC(\mathbf{this}, m, vs, st)\}$
$\{\exists\, st \cdot MVs(m, vs, st) * controller(\mathbf{this}, m, st)\}$
[Axiom [a3][$\mathbf{this}/c$] (R/L)]
$\{\exists\, st \cdot MVs(m, vs, st) * \mathbf{this}.model \mapsto m *$
$\mathbf{this}.state \mapsto st\}$[Def. of $Controller.controller(\ldots)$]
$\mathbf{this}.state = b;$
$\{\exists\, st \cdot MVs(m, vs, st) * \mathbf{this}.model \mapsto m *$
$\mathbf{this}.state \mapsto b\}$
$\{\exists\, st \cdot MVs(m, vs, st) * controller(\mathbf{this}, m, b)\}$
[Def. of $Controller.controller(\ldots)$]
$model.update(b);$
$\{MVs(m, vs, b) * controller(\mathbf{this}, m, b)\}$
$\{MVC(\mathbf{this}, m, vs, b)\}$ [Axiom [a3][$\mathbf{this}/c$] (L/R)]

In the proof, labels like "[Def. of $View.view(\ldots)$]" mean folding/unfolding the definition of $view(\ldots)$ in class $View$; and "[Axiom [a3][$\mathbf{this}/c$] (R/L)]" means using axiom [a3] from its left side (L) to right side (R) by substituting parameter c to $\mathbf{this}$. Steps without explicit labels normally use inference rules. Finally, we conclude the two methods are correct.

Thus, with proving all axioms well-supported and all methods satisfying 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. 7**, by using the extended formal specifications including axioms [a1-a3] of the MVC architecture and similar labels as in method verifications. It shows, the client can easily finish its verification now, thus makes a correct application of the MVC architecture.

6 Related Work and Conclusion

In this paper, we focus on the specification and verification of software built on components which interact with each other through clear defined interfaces. Here the components are implemented using OO techniques. We propose to integrate axioms on the specification of interfaces to constrain the implementations. The axioms relate isolated abstract predicates in specifications to support client verification, and in the same time, keep good isolation and abstraction. This work enriches our former framework VeriJ, and provides an effective approach to specify and verify interactive component based systems (CBSs). Using the technique, we successfully specify a simple MVC architecture and give it a correct implementation. The framework can well support information hiding, modularity and extensibility in specifying and verifying OO programs.

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

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

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

(3.′) $\{\mathit{MVC}(c, m, \{v_1\}, i)\}$ [Axiom [a1][$\{v_1\}, v_1/vs, v$] (L/R)]

(4.) $c.\mathit{userInput}(5);$

(5.) $\{\mathit{MVC}(c, m, \{v_1\}, 5)\}$

(5.′) $\{\mathit{model}(m, \{v_1\}, 5) * \mathit{view}(v_1, m, 5) * \mathit{controller}(c, m, 5)\}$
[Axiom [a1][$\{v_1\}, v_1, 5/vs, v, st$] (R/L)]

(6.) $\mathit{VI}\ v_2 = \mathbf{new}\ \mathit{View}_2(m);$

(7.) $\{\mathit{model}(m, \{v_1, v_2\}, 5) * \mathit{view}(v_1, m, 5) * \mathit{view}(v_2, m, 5) * \mathit{controller}(c, m, 5)\}$

(8.) $v_1.\mathit{paint}(m);$

(9.) $\{\mathit{model}(m, \{v_1, v_2\}, 5) * \mathit{view}(v_1, m, 5) * \mathit{view}(v_2, m, 5) * \mathit{controller}(c, m, 5)\}$

(9.′) $\{\mathit{MVC}(c, m, \{v_1, v_2\}, 5)\}$ [Axiom [a1][$\{v_1, v_2\}, 5/vs, st$] (L/R)]

(9″.) $\{\exists r_1, r_2 \cdot \mathit{MVC}(c, m, \{r_1, r_2\}, 5)\}$

Fig. 7. The Correct Proof of the Client Code Segment

The axioms play two important roles: (1) specifying semantic constraints for the implementation and interactions in CBSs. (2) supporting client code reasoning between objects based on interface definitions in an abstract (free of the concrete implementations hiding in predicate definitions) and modular (avoiding re-verification dynamically called implementations) way. With axioms specified for a system design, we require to verify the correctness of system implementations in two aspects: the well-supportedness of each axiom and each class is correct with its specifications. Behavioral subtyping property is ensured by checking the refinement relations of method specifications.

To our limited knowledge, there exist some works specifying and reasoning CBSs in different ways. [9] combined abstraction techniques of model variables [4] and model programs to specify interfaces of CBSs, and extended the behavioral subtyping concept for CBSs. However, the work just pointed out subtypes should obey the specifications of instance methods, no formalization details for modular reasoning was given. [2] used algebraic specification [8] 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. [17] adopted model variables and pure methods in specifying interfaces of encapsulated components too. They specified invariants expressing properties of components but a behavioral subtype relation for components was absent.

On the other hand, our thought of axioms is similar to MultiStar [19], where the *export* and *axiom* clauses are introduced under a separation logic with intuitional semantics to express properties holding for individual classes and an entire multiple inheritance hierarchies. Undoubtedly, our axiom mechanism can also specify properties of predicates defined in an individual class for global usage. Comparing to the technique of abstract predicate family (apf) [16] used in MultiStar, we simply use abstract predicate applications which uniform their two kinds of clauses into axioms. Each predicate application actually encapsulates all its polymorphic definitions in implementations and its meaning can be determined by applying our fix function and inference rules. We skillfully use this idea in proving axioms and method specifications to avoid infinite expansion of recursive predicate definitions which is not considered in MultiStar.

Further, we allow subclasses reuse predicate definitions in their superclass by inheritance. It is a nature manner as fields and methods (also method specifications) inheritance in OO programs, but MultiStar with apfs forbids so. Thus their predicate entries defined like "$x.P_C(y : a)$ as $x.P_B(y : a)$" (C is a subclass of B) which has no essential data abstractions of C indeed cause additional obligations for proving inherited axioms (relating with P) from B. Otherwise, we can conveniently inherit definitions of such $P(x, y)$ from B for C, and need not to reverify related axioms when no other predicates in them are overridden in C. Besides, the constraint effect of their export clauses is narrowed to the only class specifies them and cannot constrain subclasses. Generally their specifications seem more complex but less modular. There are similar shortcomings for the *export* clauses in jStar [6] which are used to express interactive objects from different classes and enable client verifications.

As future work, we would investigate more challenges [10] such as object invariant, frame problem in specifying and verifying OO programs. We attempt to apply our approach for more interactive systems such as design patterns [7] and other complex MVC architectures, and consider specifying and verifying CBSs with problems like adaptation [1], non-functional properties (i.e., performance, security) [20], complex upgrading and callback [3, 15], and so on. Meanwhile, we are carrying on developing an urgent tool to support our framework for convenient usage.

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. Bensalem, S., Bozga, M., Nguyen, T.H., Sifakis, J.: Compositional verification for component-based systems and application. The Institution of Engineering and Technology Software 4, 181–193 (2010)
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. Distefano, D., Parkinson, M.J.: jstar: Towards practical verification for java. In: OOPSLA'08. pp. 213–226. ACM (2008)
7. Gamma, E., Helm, R., Johnson, R., Vlissides, J.: Design Patterns, Elements of Reusable Object-Oriented Software. Addison-Wesley Reading (1994)
8. Guttag, J.V., Horowitz, E., Musser, D.R.: Abstract data types and software validation. Communications of the ACM 21(12), 1048–1064 (1978)
9. 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)
10. 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)

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

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

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

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

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

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

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

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

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

Fig. 8. Basic Inference Rules

11. 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)
12. Liskov, B., Wing, J.M.: A behavioral notion of subtyping. ACM Transactions on Programing Languages and Systems 16(6), 1811–1841 (1994)
13. 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)
14. Liu, Y., Qiu, Z.: A separation logic for OO programs. In: FACS'10. vol. 6921 of LNCS, pp. 88–105. Springer (2012)
15. McCamant, S., Emst, M.D.: Early identification of incompatibilities in multi-component upgrades. In: ECOOP'04. vol. 3086 of LNCS, pp. 440–464. Springer (2004)
16. Parkinson, M.J., Bierman, G.M.: Separation logic, abstraction and inheritance. In: POPL'08. pp. 75–86. ACM (2008)
17. Poetzsch-Heffter, A., Schäfer, J.: Modular specification of encapsulated object-oriented components. In: FMCO'05. vol. 4111 of LNCS, pp. 313–341. Springer (2006)
18. 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)
19. Van Staden, S., Calcagno, C.: Reasoning about multiple related abstractions with multistar. In: OOPSLA'10. pp. 504–519. ACM (2010)
20. Zschaler, S.: Formal specification of non-functional properties of component-based software systems. Software and Systems Modeling 9, 161–201 (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 [13, 18].

Basic inference rules are given in **Fig. 8**. We skip the explanations of many simple rules here. Rules [H-DPRE], [H-SPRE] are key to show our idea that specification

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

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

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

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

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

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

Fig. 9. Inference Rules related to Methods and Constructors

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 used in the definition of the predicates. However, these two rules are different. [H-DPRE] says if r is of the 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 (see Section 4), and provides a static explanation for ψ. In 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. 9** 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 behavioral subtyping property in a program.

[H-MTHD1] is for verifying methods with a specification (and of course a definition) in a class. It demands firstly that $C.m$'s body meets its specification, and then asks to check the refinement relations 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. Rule [H-CONSTR] is for constructors which has a similar form with [H-MTHD1]. However, a constructor cannot have multi-specifications. Here $\mathsf{raw}(\mathbf{this}, C)$ specifies that $\mathbf{this}$ 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.

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

B Other Method Verifications

In this appendix, we give details of verifying methods with their specifications in our implementation (in **Fig. 2, 6**) of a MVC architecture. Specially, verifications of two methods $\mathit{View}.\mathit{View}(m)$ and $\mathit{Controller}.\mathit{userInput}(b)$ are showed in Section 5. As the implementing detail of class View_2 which acts like class View is unimportant and omitted, verification of its correctness will also be omitted.

Since these methods in implementing classes inherit their specifications from interfaces, inference rules proposed in our former work [13, 18] for verifying programs with interfaces are useful and applied to prove them. Also, all well-supported axioms listed and well-proven in Section 5 can help their deductions. Note that, there are calls of some methods, such as $\mathit{Controller}.\mathit{Controller}(m)$ and $\mathit{View}.\mathit{paint}(m)$ calling for $\mathit{Model}.\mathit{getState}()$, $\mathit{Model}.\mathit{addView}(v)$ and $\mathit{Model}.\mathit{update}(b)$ calling for $\mathit{View}.\mathit{paint}(m)$. Therefore, we should prove the called methods ahead of the calling ones.

Firstly, we prove methods $\mathit{Model}.\mathit{Model}()$ and $\mathit{Model}.\mathit{getState}()$ as follows, both of which are basic and call no method in this implementation.

Proving $\mathit{Model}.\mathit{Model}()$:
$\{\mathbf{emp}\}$
$\mathbf{this}.\mathit{state} = 0;$
$\{\mathbf{this}.\mathit{state} \mapsto 0;\}$
$\mathbf{this}.\mathit{views} =$
  $\mathbf{new}\ \mathit{ArrayList}\langle \mathit{VI}\rangle();$
$\{\mathbf{this}.\mathit{state} \mapsto 0 *$
  $\mathbf{this}.\mathit{views} \mapsto \emptyset\}$
$\{\mathit{model}(\mathbf{this}, \emptyset, 0)\}$
[Def. of $\mathit{Model}.\mathit{model}(\ldots)$]

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

Then, using the correctness of method $\mathit{Model}.\mathit{getState}()$, we prove two methods $\mathit{Controller}.\mathit{Controller}(m)$ and $\mathit{View}.\mathit{paint}(m)$.

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

At last, we step to prove the two complicated methods $Model.addView(v)$ and $Model.update(b)$ by making use of the correctness of method $View.paint(m)$.

In conclusion, all methods declared in our implementation are correct with their specifications respectively. And any client of this implementation could call these methods with specifications to do verifications.

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

```
Proving $Model.update(\mathbf{b})$ :
$\{MVs(\mathbf{this}, vs, -)\}$
$\{\exists\, st \cdot MVs(\mathbf{this}, vs, st)\}$
$\{\exists\, st \cdot model(\mathbf{this}, vs, st) * (\circledast_{v \in vs} view(v, \mathbf{this}, st))\}$
                [Axiom [a2][$\mathbf{this}/m$] (R/L), or Def. of $Model.MVs(\ldots)$]
$\{\exists\, st \cdot \mathbf{this}.views \mapsto vs * \mathbf{this}.state \mapsto st * (\circledast_{v \in vs} view(v, \mathbf{this}, st))\}$
                                                [Def. of $Model.model(\ldots)$]
$\mathbf{this}.state = b;$
$\{\exists\, st \cdot \mathbf{this}.views \mapsto vs * \mathbf{this}.state \mapsto b * (\circledast_{v \in vs} view(v, \mathbf{this}, st))\}$
$\{model(\mathbf{this}, vs, b) * (\circledast_{v \in vs} view(v, \mathbf{this}, st))\}$        [Def. of $Model.model(\ldots)$]
$\mathbf{for}(\,VI\ v : views)\{$
  $\{\exists\, st, vs_1, v, v_1, v_2, vs_2 \cdot vs = vs_1 \cup \{v\} \cup vs_2 \wedge model(\mathbf{this}, vs, b) *$
    $(\circledast_{v_1 \in vs_1} view(v_1, \mathbf{this}, b)) * view(v, \mathbf{this}, st) * (\circledast_{v_2 \in vs_2} view(v_2, \mathbf{this}, st))\}$
  $v.paint(\mathbf{this});$
  $\{\exists\, st, vs_1, v, v_1, v_2, vs_2 \cdot vs = vs_1 \cup \{v\} \cup vs_2 \wedge model(\mathbf{this}, vs, b) *$
    $(\circledast_{v_1 \in vs_1} view(v_1, \mathbf{this}, b)) * view(v, \mathbf{this}, b) * (\circledast_{v_2 \in vs_2} view(v_2, \mathbf{this}, st))\}$
  $\{\exists\, st, vs'_1, v', v'_1, v'_2, vs'_2 \cdot vs = vs'_1 \cup \{v'\} \cup vs'_2 \wedge vs'_1 = vs \cup \{v\} \wedge vs_2 = vs'_2 \cup \{v\} \wedge$
    $model(\mathbf{this}, vs, b) * (\circledast_{v'_1 \in vs'_1} view(v'_1, \mathbf{this}, b)) * view(v', \mathbf{this}, st) *$
        $(\circledast_{v'_2 \in vs'_2} view(v'_2, \mathbf{this}, st))\}$
$\}$
$\{model(\mathbf{this}, vs, b) * (\circledast_{v \in vs} view(v, \mathbf{this}, b))\}$
$\{MVs(\mathbf{this}, vs, b)\}$        [Axiom [a2][$\mathbf{this}/m$] (L/R), or Def. of $Model.MVs(\ldots)$]
```