Specifying and Verifying MVC Architectures 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. Formal specification and verification of Object Oriented programs with many challenges has been deeply studied and some achievements have been gained. However, in real applications, we are facing large programs with design patterns or complex architectures in which modules are such as classes and others. Until now, few work focus on this kind of programs because they are difficult to specify and verify. In this work, we attempt to develop an approach to specify the popular and mature MVC architectures in practical interactive software designs. MVCs consist of three main components: model, view and controller, which are independent modules taking charge of their own functionalities then form a complete and powerful system with flexible modularity and maintainability. We will depict a general specification scheme for these components, as well as their intricate interactions as the nature design of MVCs with some additionally specified global properties named axioms in our framework VeriJ.

Keywords: Object Orientation, Abstraction, Modularity, Specification, Predicate, Axiom, Verification, Separation Logic, Model-View-Controller

1 Introduction

Formal techniques are already widely applied in software and hardware development, such as refining formal specification from informal requirements, code generation from formal specification, validation and verification, and so on. In Object Oriented field especially, many challenges are still, e.g., specifying and verifying properties related to dynamic binding, aliasing, inheritance, modularity, frames, automating the work, etc. Even so, formal methods are more preferable in many aspects to informal ones, because they are effective in checking the reliability and robustness of designs, and assuring the correctness of systems. In this area, many achievements related to specification and verification of programs with mutable data structures, inheritance hierarchies [13–21], or simple design patterns [5–11] have been made.

However, in practical OO development, more complicated designs than design patterns [4] or inheritance hierarchies are needed, such as various framework architectures of which the formal study is rare. While the design patterns are only solutions to solve local and concrete problems in the design, a framework design such as the Model-View-Controller (MVC) architecture can success in decomposing software system designs

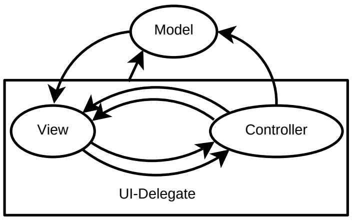


Fig. 1. MUI architecture in Java Swing

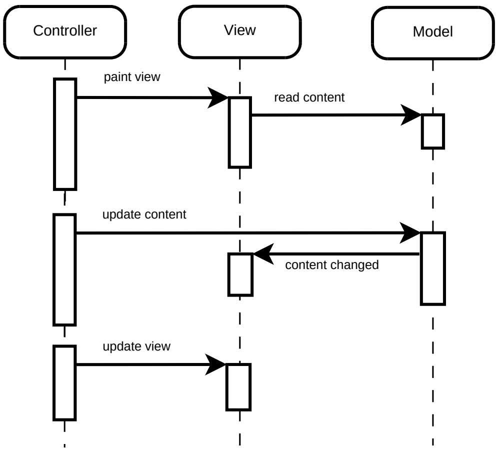


Fig. 2. Interactions among model/view/controller

into several modular and interactive components, that form an overall scale organization for a whole or a large part of the system. As an example, the observer pattern is usually integrated into a global MVC architecture as a component.

In this work we will take MCV architecture as the topic, to study in dealing with a more complicated structures, what problems we will meet, and how these problems could be tackled. MVC one of the oldest system design architectures in OO field. It was introduced by Smalltalk in 1970's and has become one of the most popular software architecture pattern, especially in applications with a Graphical User Interface (GUI). MVC allows us to separate the application logics (models) from the GUIs (views) in the design, resulting a easily modified system in which either of these two aspects can be evolved without affecting the other(s). In detail, a model maintains some data and some rules that govern accessing to and updating of the data, which can be shared among any number of view and controller objects. A view can render its model's data in its own style to users, and must update its own representation while the model data changes. A controller translates user inputs on views into actions that the model will perform. Interactions between these three components may be a little different depending on the detailed versions of MVC architecture designs.

Take Java Swing as an example [3], which renders a MVC architecture as the M-UIs (seeing **Fig. 1**), consisting of a separated model and several UI-delegates. A UI-delegate is a combination of a view and a controller. When the controller notices a user action on a view, it deals with the action and tells the model or view to update accordingly. If the data of the model needs to be updated in response to the user action, the model modifies its data and notifies all its views, then they will read this new data and repaint themselves respectively. These interactions are depicted briefly in **Fig. 2**.

Applying MVC architecture in system design has many advantages due to its separation the use of data from their representation absolutely. Data in the model are shared among several views which make can make different usage of the data. This separation also makes the model independent. In addition, the controllers bring more flexibility to satisfy user's requirements. MVC architecture allows lower coupling, better modularity and maintainability of the systems. But there are also disadvantages lying in MVC, such as the difficulty to analyze its mechanism because of the complex structures and complicated interactions between its components. It costs time to consider how to apply the MVC design into application programs. Furthermore, the strict separation of model with view brings also inconvenience to debug programs.

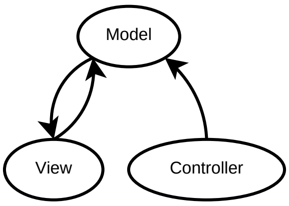
The popularity of applying MVC architecture design in practice software engineering development, together with its above advantages and disadvantages inspire and interest us to formalize it, with an aim to help understanding and correctly using MVC. It will be a meaningful but also challenging work in several aspects. One main challenge is to give a general specification which really reflects the ideas of MVC architecture, such as the interactions among the three components, the abstraction granularity of specification for modularity and maintainability, and so on.

To our limited knowledge, only few early work had been done in formalizing MVC. In 1997 [1] used Object-Z to define relationships between objects of MVC architecture in an aggregate class, and in 2006 [2] similarly applied Object-Z to describe a distributed MVC architecture for enriching media experience. In these work, the specifications creates only relation between operation names. And they talked more about comparison between MVC and another OO user-interface architecture, PAC (presentation, abstraction, controller) but no about using the the specifications to verify implementations or applications. Differently, our goal here insists to specify these relationships using abstract symbols of different components in MVC architecture which are independent of operations in implementations, and support the verification of concrete implementations and MVC based applications.

In this work, we provide an approach to specify a MVC architecture formally based on our VeriJ framework [23, 25] together with a new technique named *axiom*s. In Section 2 we give a simple MVC architecture and its formal specification, and discuss problems appeared in the relative verification. In Section 3 we extend our VeriJ verification framework with axioms. We study the fundamentals of axioms in Section 4, and then apply the extended framework to specify our MVC architecture, reexamine the verification of its implementation and client codes in Section 5. At last, we conclude, compare some related work and make future perspectives.

2 Basics

In the first, we do not want to limit our work on a formal treatment of a special and concrete MVC implementation, because that is too narrow a problem and not very useful. Our goal is to develop an approach to formalize MVC architecture, that can be used to judge if a set of modules form a "correct" MVC implementation. The first problem we must tackle is how to specify an "architecture" without mentioning its concrete implementations. In practice people use a set of interrelated interfaces to describe an abstract architecture, while no semantic restrictions can be given. We need to extend this design to the formal world and incorporate in semantical aspects in a suitable way. Our results on specification/verification of OO programs with interfaces [25] are useful here. In this section, we will present a simple MVC architecture design following the guidance of in-



i1) A complete instance set of the MVC architecture consists of one model, several views and one controller;

i2) Each component in MVC is separated and modular in fulfilling its duty to form the whole architecture, and some of them can be combined;

i3) A view can be attached on one existing model at an time and different views of the model are distinct;

i4) The controller and all views of a model must keep synchronous changes with the model's evolution.

Fig. 3. Interactions and Informal Specification of the MVC Architecture

```
interface MI {
  def model(this, vs, st);
  def MVs(this, vs, st);
  void addView(VI v)
  ⟨MVs(this, vs, st) ∗ view(v, this, -)⟩
  ⟨MVs(this, vs ∪ {v}, 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 {
  def view(this, m, st);
  void paint(MI m)
  ⟨view(this, m, -) ∗ model(m, vs, st)⟩
  ⟨view(this, m, st) ∗ model(m, vs, st)⟩;
}
interface CI{
  def controller(this, m, st);
  def MC(this, m, st);
  def MVC(this, vs, m, st);
  void userInput(Int b)
  ⟨MVC(this, vs, m, -)⟩
  ⟨MVC(this, vs, m, b)⟩;
}
```

Fig. 4. Interfaces with Formal Specifications for MVC Architecture

teraction structures and informal specifications depicted in **Fig. 3**, and formalize it using our VeriJ framework. This design supports building up the MVC-based (sub-)systems and the correct interactions between its components.

The interface declarations with formal specification are given in **Fig. 4**, where three interfaces MI, VI, CI describe duties of the basic components (corresponding to the model, view, and controller) of the MVC architecture. MI provides some methods to add a view into its view set, and to get and update its data state; VI provides a method to paint the state of its model in its specific form for users; and CI deals with the user inputs obtained. In the formal aspects, we add specifications to each method signatures here to describe their functions. The specification for a method is a pair of pre and post conditions in the form of $\langle pre\rangle\langle post\rangle$.

As designed in our former work on specification and verification of OO programs with interface-based design [25], in the interfaces, we can declare *specification predicates* which have no concrete definitions. These predicates are just abstract symbols to encapsulate the details of the interface's prospective implementations. For example, $model(\mathbf{this}, vs, st)$ in MI is introduced here to hiding the details of MI's implementation classes. Although definitions of the predicates are postponed to the implementing classes, they are useful in specify the duty of the methods here in an abstract way. For example, the specification for method $MI.update$ says that, to call it, the state must satisfy assertion $MVs(\mathbf{this}, vs, st)$ (the precondition), and after calling, the post state would make assertion $MVs(\mathbf{this}, vs, b)$ (the postcondition) hold. Things are similar for the other methods. Please note that the pre and post conditions are given based on $\mathbf{this}$ object which may be accompanied by some parameter(s).

Having the abstract definition of the MVC architecture built upon interfaces with formal specifications, now we give a simple implementation in **Fig. 5**. Four classes are defined to implement the above interfaces by embodying their declared predicates and methods with corresponding definition bodies. Methods in these classes inherit their specifications from the interfaces. We omit the detail implementing information of class $View_2$ which is similar to class $View$ to implement interface VI and define another view style for painting the model. Here the event handling mechanism used between Java Swing components is simplified, and the form of some features like the generic $List\langle VI\rangle$, **for** statements, and also system.out.println() are adjusted to suit with our VeriJ which is just a subset of sequential Java. The adjustments have no real effect on the main purpose of the work. They are only details for the implementation and would be abstracted into certain data structures such as a sequence or set for the generic typed objects. We will focus on analyzing the important aspects of formally specifying the MVC architecture design in the following.

For the predicates in class $Model$, $model(\mathbf{this}, vs, st)$ means that the current model object $\mathbf{this}$ has a view set vs and a state st; $VSet$ records recursively all the view objects of the model in set vs and each of them is disjoint with others but maintains the same painting state as the model's state; MVs specifies the aggregate structure containing the model object m and its view set. In class $View$, predicate $view(\mathbf{this}, m, st)$ says that the current view object renders model m and the state painted is st. In class $View_2$, $view(\mathbf{this}, m, st)$ is similar to the same signature one in $View$ while the concrete states abstracted in parameter st of these two kinds of views are different. In class $Controller$, predicate $controller(\mathbf{this}, m, st)$ says that the current controller object records in model m, and a state st obtained from a user input is passed to the model; MC specifies the aggregate structure of model m and its controller; and MVC specifies the whole MVC structure where each object is separated from others in the heap. Notice that the parameters st in the three basic predicates $model(m, vs, st)$, $view(v, m, st)$, and $controller(c, m, st)$ track the synchronization of responding a user input among model/view/controller objects in the MVC architecture, thus it abstractly reflect and specify the interactive mechanism between them. They should be the same after carrying out once the whole process of responding a user input.

Based on these predicate definitions, method specifications using them have their concrete meaning now. Taking $update$ in $Model$ as an example, its specification says that if the caller is a model object with its view object set vs, then after the call, its state would be updated to b and all its view objects repaint this new state accordingly. Similarly for other methods with specifications. Thus, as a first step, we have specified a simple MVC architecture design using our VeriJ framework. Since all predicates and methods have their concrete semantics in these implementing classes, a set of proof obligations would be generated during formally verifying this implementation, such as the correctness of methods with their specifications which can be referred in Appendix A. By doing so, we want to conclude that this implementation is "correct"

```
class $Model \rhd MI\{$
  def $model(\mathbf{this}, vs, st)$ :
    $\mathbf{this}.state \mapsto st \ast \mathbf{this}.views \mapsto vs;$
  def $VSet(\mathbf{this}, vs, st)$ :
    $(vs = \emptyset \wedge \mathbf{emp}) \vee (vs = vs' \cup \{v\} \wedge$
    $VSet(\mathbf{this}, vs', st) \ast view(v, \mathbf{this}, st));$
  def $MVs(\mathbf{this}, vs, st)$ :
    $model(\mathbf{this}, vs, st) \ast$
    $VSet(\mathbf{this}, vs, st);$
  Int $state;$
  $List\langle VI\rangle views;$
  $Model()\langle\mathbf{emp}\rangle\langle model(\mathbf{this}, \emptyset, 0)\rangle$
  $\{$ $\mathbf{this}.state = 0;$
    $\mathbf{this}.views =$ new $ArrayList\langle VI\rangle();$
  $\}$
  Int $getState()\{$ Int $s;$
    $s = \mathbf{this}.state;$ return $s; \}$
  void $addView(VI\ v)\{$
    $views.add(v);\ v.paint(\mathbf{this}); \}$
  void $update($Int $b)\{$
    $\mathbf{this}.state = b;$
    for$(VI\ v : views)\{v.paint(\mathbf{this}); \}$
  $\}$
$\}$
class $View \rhd VI\{$
  def $view(\mathbf{this}, m, st)$ :
    $\mathbf{this}.model \mapsto m \ast \mathbf{this}.state \mapsto st;$
  $MI\ model;$ Int $state;$
  $View(MI\ m)$
  $\langle MVs(m, vs, st)\rangle$
  $\langle MVs(m, vs \cup \{\mathbf{this}\}, st)\rangle$
  $\{$ $\mathbf{this}.model = m;$ $\mathbf{this}.state = 0;$
    $m.addView(\mathbf{this}); \}$
```

```
  void $paint(MI\ m)\{$
    if $(m{=}{=}\mathbf{this}.model)\{$
      $\mathbf{this}.state = m.getState();$
      System.out.println$(state);$ $\}$
  $\}$
$\}$
class $View_2 \rhd VI\{$
  def $view(\mathbf{this}, m, st)$ :
    $\mathbf{this}.model \mapsto m \ast \mathbf{this}.state \mapsto st;$
  $MI\ model;$ Int $state;$
  $View_2(MI\ m)$
  $\langle MVs(m, vs, st)\rangle\langle MVs(m, vs \cup \{\mathbf{this}\}, st)\rangle$
  $\{\ldots\}$
  void $paint(MI\ m)\{ \ldots \}$
$\}$
class $Controller \rhd CI\{$
  def $controller(\mathbf{this}, m, st)$ :
    $\mathbf{this}.model \mapsto m \ast \mathbf{this}.state \mapsto st;$
  def $MC(\mathbf{this}, m, st)$ :
    $model(m, vs, st) \ast controller(\mathbf{this}, m, st);$
  def $MVC(\mathbf{this}, vs, m, st)$ :
    $model(m, vs, st) \ast VSet(m, vs, st) \ast$
    $controller(\mathbf{this}, m, st);$
  $MI\ model;$
  Int $state;$
  $Controller(MI\ m)$
  $\langle model(m, vs, st)\rangle\langle MC(\mathbf{this}, m, st)\rangle$
  $\{$ $\mathbf{this}.model = m;$
    $\mathbf{this}.state = m.getState(); \}$
  void $userInput($Int $b)\{$
    $\mathbf{this}.state = b;$
    $model.update(b); \}$
$\}$
```

Fig. 5. An Implementation of the MVC Component Interfaces

according to our former definition of "*correct program*" [25]. However, some problems appear in verifying client codes which make use of the MVC's implementation.

Consider a piece of client code and its verification process given in **Fig. 6**. For modular verification of programs with interfaces [25], we demand to be able to verify client codes referring to types of the variables. Here the types are interfaces, e.g. MI, and specifications in them are based on (abstract) predicates with signatures which are just discrete symbols in the eyes of the clients. Therefore, the client could not know how to deduce the assertions with underlining, such as $MVs(m, \emptyset, 0)$ in line (5') from line (5) based on the *variables' types*, although the objects used have been specified as the MVC components, and their interactive features have been incorporated into formal specifications. Similar problems occur for the derivation of line (7') from line (7), line (9') from (9), and line (11') from (11).

$$
\begin{array}{lll}
(1.) & \{\mathbf{true}\} & \\
(2.) & \mathit{MI}\ m = \mathbf{new}\ \ \mathit{Model}(); & \text{// create a model} \\
(3.) & \{\mathit{model}(m, \emptyset, 0)\} & \\
(4.) & \mathit{CI}\ c = \mathbf{new}\ \ \mathit{Controller}(m); & \text{// create a controller of } m \\
(5.) & \{\mathit{MC}(c, m, 0)\} & \\
 & \qquad \Downarrow ??? & \\
(5'.) & \{\underline{\mathit{MVs}(m, \emptyset, 0)} * \mathit{controller}(c, m, 0)\} & \\
(6.) & \mathit{VI}\ v_1 = \mathbf{new}\ \ \mathit{View}(m); & \text{// add a new view} \\
(7.) & \{\mathit{MVs}(m, \{v_1\}, 0) * \mathit{controller}(c, m, 0)\} & \\
 & \qquad \Downarrow ??? & \\
(7'.) & \{\underline{\mathit{MVC}(c, \{v_1\}, m, 0)}\} & \\
(8.) & c.\mathit{userInput}(9); & \text{// process a user input} \\
(9.) & \{\mathit{MVC}(c, \{v_1\}, m, 9)\} & \\
 & \qquad \Downarrow ??? & \\
(9'.) & \{\underline{\mathit{MVs}(m, \{v_1\}, 9)} * \mathit{controller}(c, m, 9)\} & \\
(10.) & \mathit{VI}\ v_2 = \mathbf{new}\ \ \mathit{View}_2(m); & \text{// add another view of model} \\
(11.) & \{\mathit{MVs}(m, \{v_1, v_2\}, 9) * \mathit{controller}(c, m, 9)\} & \\
 & \qquad \Downarrow ??? & \\
(11'.) & \{\, \mathit{VSet}(m, \{v_2\}, 9) * \underline{\mathit{view}(v_1, m, 9) *} & \\
 & \quad \underline{\mathit{model}(m, \{v_1, v_2\}, 9)} * \mathit{controller}(c, m, 9)\} & \\
(12.) & v_1.\mathit{paint}(m); & \\
(13.) & \ldots\ldots &
\end{array}
$$

Fig. 6. A Client Code Segment with Verification Based on Our MVC Architecture design

Contrasting to the situation of clients, all the deductions in **Fig. 6** could be done when the details of the implementing classes are allowed to inspect. In our intention, assertion $MC(c, m, 0)$ conjoins separated heaps, $\mathit{model}(m, \emptyset, 0)$ for a model and $\mathit{controller}(c, m, 0)$ for its controller. In addition, when there is no view registering on m, assertion $MVs(m, \emptyset, 0)$ should equal to $\mathit{model}(m, \emptyset, 0)$. But clients could not intelligently deduce the migration between the views on different objects while no specification tells them so.

Some people may think that the problem roots in our specification which does not provide enough information for the clients, and suggest to specify, for example, method $\mathit{userInput}$ based on "$\mathit{model}(\dots) * \mathit{controller}(\dots) * \mathit{view}(\dots)$". This suggestion does not work because clients can only utilize specifications of the methods in their verification, while the specifications are abstract and can not specify explicitly each view object of a model as the view number of model is not limited and can not be expected in MVC architecture design. Feasibly, to adopt this suggestion, a recursive predicate $\mathit{VSet}(m, \dots)$ meaning a view set should be defined to conveniently integrate all view objects for model m and allow separating a single view object from the set. (We illustrate this suggestion in detail in Appendix B.) In addition, adopting the above suggestion will generally lead to very long specifications for the methods involving states of many other objects, especially when they have depth embedded side effects. Thus, in this paper, we prefer to introduce terse formalism to specify aggregate structures and general properties among different objects in MVC architecture, and then the instances or clients of MVC can be aware of and conveniently use them during their verifications.

In the following sections, we introduce our approach and apply it to analyze and solve the problems discussed above.

3 A Specification Language with Axioms

Our verification framework is built on a tailored separation logic (to fit OO necessaries) with user defined predicates. We separated the visibility of predicate definitions from their signatures to support information hiding, and thus support modular verification in the presence of inheritance and overriding [23]. As the next step, we extended the framework by allowing declarations of predicates in interfaces without definitions, and then each implementing class must accordingly define them in its own way that hides its implementation [25]. Our technique with this extension makes specifying and verifying OO programs with interface-based design possible while that kind of designs is thought to be growing with wide application and great perspective.

However, the framework is still not power enough to verify even more complicated OO systems. Return to the MVC example in last section, in practice, people use informal documents to describe the "semantical requirements" of the architecture behind the design, which developers should follow in order to build "correct" implementations. Then they declare interfaces to sketch the architecture as a guidance for implementing classes. Actually, a group of classes with signature-correct methods will be recognized (by compiler) as a "correct" implementation of the architecture because there is no semantic specification. To ensure semantic correctness, we include formal specifications, and require that implementing classes should additionally provide definitions for the predicates declared in the interface(s), and the code for each method should meet its specifications. However, these requirements are all local to each method or class, thus cannot carry enough information to support verifications of the MVC's client codes. For a group of three classes becoming a correct implementation of the MVC architecture, they should have correct interactions that are hard to express conveniently by local specifications in method or class level. Obviously, this problem is common in any system consisting of several or many interrelated classes.

To deal with the problem, we propose to introduce another set of formal facilities, *axioms*, designed for describing interrelations between predicates declared/defined in one or more interface(s)/class(es). On one side, axioms can enforce additional restrictions for the implementations and rule out the incorrect ones, that is one feature that we thirst for. On the other side, we let axioms visible to client codes, thus they can be used in client-level deductions. This uncovering will not break the encapsulation because axioms are described based on the abstract predicates. Now we introduce our technique formally, and integrate *axioms* into our framework VeriJ.

The assertion language used here is a version of Separation Logic [24],

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

$$
\begin{array}{rcl}
v & ::= & \textbf{this} \mid x \\
e & ::= & \textbf{true} \mid \textbf{false} \mid \textbf{null} \mid v \mid d \mid e \odot e \\
b & ::= & \textbf{true} \mid \textbf{false} \mid e < e \mid e > e \\
 & \mid & e = e \mid \neg b \mid b \vee b \mid b \wedge b \\
c & ::= & \textbf{skip} \mid x := e \mid v.a := e \mid x := v.a \\
 & \mid & x := (C)v \mid x := v.m(\overline{e}) \mid x := \textbf{new}\, C(\overline{e}) \\
 & \mid & \textbf{return}\ e \mid c; c \mid \textbf{if}\, b\, c\, \textbf{else}\, c \mid \textbf{while}\, b\, c \\
T & ::= & \textbf{Bool} \mid \textbf{Object} \mid \textbf{Int} \mid C \mid I \\
P & ::= & \textbf{def}\ [\textbf{pub}]\ p(\textbf{this}, \overline{x}) \\
A & ::= & \textbf{axiom}\ \phi
\end{array}
\qquad
\begin{array}{rcl}
\pi & ::= & \langle\varphi\rangle\langle\psi\rangle \\
M & ::= & T'\ m(\overline{T_1\ z}) \\
L & ::= & \textbf{interface}\ I\ [:\ J]\ \{\overline{P;}\ \overline{M\ [\pi];}\} \\
K & ::= & \textbf{class}\ C : B\ [\rhd\overline{I}]\ \{ \\
 & & \quad \overline{[\textbf{pub}]\ T\ a;}\ \overline{P : \psi;} \\
 & & \quad C(\overline{T_1\ z})\ [\pi]\ \{\overline{T_2\ y;}\ c\} \\
 & & \quad \overline{M\ [\pi]\ \{\overline{T_2\ y;}\ c\}}\ \} \\
N & ::= & (K \mid L) \mid (K \mid L)\ N \\
G & ::= & \overline{A;}\ N
\end{array}
$$

Fig. 7. Syntax of VeriJ with Axioms

where T is a type, v a variable or constant, r_1, r_2 references, ψ an assertion containing ρ for stack assertions, η for heap assertions and also normal forms in typical logics and SL. Notations like $r_1 = r_2$ tells r_1 and r_2 are identical; $r : T$ means r refers to an object with exact type T; $r <: T$ means that r refers to an object of T or one of its subtypes; $v = r$ asserts that 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 that the heap is exact a field a of an object (denoted by r_1) holding value r_2; $\mathsf{obj}(r, T)$ means that the heap is exact an entire object of type T, which r refers to. 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 current heap. In addition, $p(\overline{r})$ denotes a user-defined predicate p with arguments $\overline{r}$. Here we suppose that each predicate has an distinct name over the whole program. This restriction is easy to release. The connectors, e.g. $*$ and $-\!\!*$, are the same as in Separation Logic.

The syntax of our programming and specification language VeriJ with the new *axioms* feature is given in **Fig. 7**, where,

- C and I are class and interface names respectively. $\mathbf{pub}$ is used to announce that a data field, method or predicate definition body is publicly accessible. Mutation, field accessing, casting, method invocation, and object creation are all taken as special assignment commands. In this paper, we use "type" to mean either a class or an interface (sometimes the primitive types are also included).
- $p(\mathbf{this}, \overline{x}) : \psi$ is a definition of specification predicates (abbr., predicates) with the body ψ, and $\mathbf{this}$ will always be given explicitly as the first parameter to denote the current object. Predicates are defined to hide information of classes and then used in method specifications to provide abstraction.
- A denotes an axiom definition where ϕ is an assertion which does not contain program variable. Thus, ϕ is built upon predicate applications and primitive separation logic formulas based on logical variables and constants, using first-order and separation logic connectors, as well as quantifiers. The logical variables in axioms are implicitly universal-quantified, thus can be instantiated in axiom application.
- π is specification pair for a constructor or method in a type. Specifications in a supertype can be inherited or overridden in subtypes. If a non-overridden method is not explicitly specified, it takes default "$\langle\mathbf{true}\rangle\langle\mathbf{true}\rangle$".
- L declares an interface which may inherit another interface J. A class C may implement some interfaces $\overline{I}$, and inherit a class B. A sub-interface should not redeclare

the same methods of its super-interfaces. As in Java, each class has a superclass, possibly **Object**, but may implement zero or more interfaces. We assume all methods are public. For simple we omit method overloading here. A program G is a set of application axioms, a sequence of class and interface declarations.

We will consider only well-typed and well-specified programs from now on. For program G, we need a static environment to support our framework. Its construction has been presented in our previous work [22, 26] where readers can refer to for details. Now we introduce a new environment component $\mathcal{A}_G$ which is the set of all axioms specified program G. More details for axioms will be given in the next section.

4 The Fundamentals of Axioms

Now we investigate the semantic foundations for the axioms according to its two aspects of effects in our specification and verification framework.

By giving axioms for a system design, we can define relations crossing several types to relate their implementations, or constrain implementation for a specific type. We have the following definition to connect axioms with interface and class declarations.

Definition 1 (Well-Supported Axiom). *Suppose N is a well-typed and well-specified sequence of interface/class declarations, and ϕ is an axiom. We say that ϕ is* well supported *by N, or N is* a model *of ϕ, written as $N \models \phi$, if all predicates appearing in ϕ are defined in N, and with any definition for the predicates in ϕ given by N, ϕ holds always. If Φ is a set of axioms, we write $N \models \Phi$ if $N \models \phi$ for every $\phi \in \Phi$. In this case, we say that Φ is* well supported *by N, or N is* a model *of Φ.*

Please notes that, because of the existences of the subclass overriding and multiple implementing classes for an interface, a predicate may have several definitions in N.

By the above definition of a well-supported axiom by a program N, we clearly see that each axiom can be proven independently of other axiom group, thus we have:

Lemma 1. *If $N \models \Phi$ and $N \models \Phi'$, then $N \models \Phi \cup \Phi'$.*

Because interfaces give no definition for existing predicates, and each predicate declared in a well-typed program has a distinct name as we assumed, so we have:

Lemma 2. *If $N \models \Phi$ and N' is a sequence of interface declarations, and $N\,N'$ is still well-typed and well-specified, we will have $N\,N' \models \Phi$.*

For a sequence of interface/class declarations N and a set of axioms Φ, we use $N \nmid \Phi$ to mean that N contains no declaration/definition for any predicate in Φ.

Lemma 3. *If $N \models \Phi$, for a sequence of interface/class declarations N' where $N' \nmid \Phi$ and $N\,N'$ is still well-typed and well-specified, we have $N\,N' \models \Phi$.*

This lemma is trivially true because N' provides no new interpretations for axioms in Φ thus makes no effect on their truth value. A special case is that N' consists of some class declarations without any predicate declaration/definition and use classes in N to implement their behaviors. We call classes in N' *simple clients* of N. Clearly, all the axioms in Φ are true assertions for them and may be used in their verification.

Farther, we have,

Lemma 4. *If $N \models \Phi$ and $N' \models \Phi'$, where $N \nmid \Phi'$ and $N' \nmid \Phi$, and $N\,N'$ is still well-typed and well-specified, we will have $N\,N' \models \Phi \cup \Phi'$.*

If a group of *client classes* which uses some existing classes and support another independent set of axioms, they fall into the coverage of Lemma 4. As the consequence, the new set of classes (with their axioms) can be simply combined with the existing classes (and the existing axioms).

When we add a new class declaration onto an existing class/interface sequence with some axioms, we need to check if it is proper with respect to the existing program components. We have a definition as follows.

Definition 2 (Proper Class Declaration). *If $N \models \Phi$, and K is a class declaration which declares a subclass of some class in N or define a new implementing class of some interface in N, and $N\,K$ is still a well-typed and well-specified program. We say that K defines a* proper class *with respect to (Φ, N), if*

(1) $K \nmid \Phi$, or
(2) K provides definition(s) for one (or more) predicate(s) appearing in some axioms in Φ, and with these definitions the related axioms keep true.

Obviously, the proper class extension can be generalized to a sequence of classes,

Lemma 5. *If $N \models \Phi$ and N' is a sequence of class/interface declarations. In addition, for any class declaration K such that $N' = N'_1\,K\,N'_2$ we know that K defines a proper class with respect to $(\Phi, N\,N'_1)$, then we have $N\,N' \models \Phi$.*

In fact, the Lemmas 2, 3, 4 and 5 given above reflect some cases of program extension where the *well-supported* property of axioms can be modularly maintained. Of course, situations could become more complicated, and at that time, we might need to go back to the basic definition.

Some forms of axioms are extremely useful, which are: conjunctions of assertions, implications, equivalence. A conjunctive axiom can be divided into a group of axioms, and each of the conjuncts can be used as a true assertion independently. For the axioms of an implication in form, we can use its consequent to replace its antecedent in deductions, and for the axioms in the form of an equivalence, we can replace its left-hand side by its right-hand side and vise verso freely in the deductions.

We have written $\Gamma_N \vdash \phi$ to denote that assertion ϕ can be proved based on N using the inference rules of VeriJ, where N is a well-typed and well-specified sequence of interface/class declarations, and Γ_N is the static environment created from N. Because we have proved that the VeriJ verification framework is sound, thus we have:

Theorem 1. *Suppose N is a well-typed and well-specified sequence of interface/class declarations and ϕ is an axiom, if $\Gamma_N \vdash \phi$, then $N \models \phi$.*

This theorem tell us that, we can use our inference rules to prove the well-supporting relation for axioms, although the framework is in general not complete. Because the axioms are state-independent (due to that they do not contain program variables), in the inference to prove them, we need at most the predicate definitions syntactically given in the classes in N, the code and their specifications will not be involved.

We have the following definitions for programs:

Definition 3 (Well-Axiom-Constrained Program). *Suppose $G = (\overline{A};\ N)$ is a well-typed and well-specified program, with interface/class declaration sequence N, its axiom set is $\mathcal{A}_G = \{\phi \mid \phi$ is an axiom given in section $\overline{A};\}$. We say that G is a well-axiom-constrained program if and only if $N \models \mathcal{A}_G$.*

If an axiom ϕ in program G is will supported, then we can use it as a true assertion in deductions of verification for the program code, make any substitution for its parameters as necessary. We introduce following inference rule:

$$\text{[H-AXM]}\ \frac{G = (\overline{A};\ N), \quad \phi \in \mathcal{A}_G, \quad N \vdash \phi, \quad \text{any } p(\overline{x}) \text{ in } \phi, \quad \overline{y} <: \overline{x}}{\phi[\overline{p(\overline{x})/p(\overline{y})})]}$$

Here we use $\overline{y} <: \overline{x}$ to mean that each variable in $\overline{y}$ is type compatible to the corresponding parameter in $\overline{x}$. Although we have not introduce types for the parameters of predicate explicitly, the meaning of this restriction is clear.

In our previous work, we have a formal definition for a well-formed *correct class* C, where two aspects of verification are required:

(1) For each $\{\varphi\}\{\psi\} \in \mathit{\Pi}(C.m)$ *(*and $\mathit{\Pi}(C.C) = \{\varphi\}\{\psi\}$*)*, $\mathit{\Gamma} \vdash \{\varphi\}\ C.m\{\psi\}$ *(*and $\mathit{\Gamma} \vdash \{\varphi\}\ C.C\{\psi\}$*)* holds. Here $\mathit{\Pi}(C.m)$ denotes the specification for method m in class C, where $\mathit{\Pi}$ is the specification environment recording the specifications for methods in the program. Generally, we may have one or more pre and post condition pair(s) $\{\varphi\}\{\psi\}$ for $C.m$, and we need to prove that method $C.m$ is correct with respect to each of them. In addition, $C.C$ is the constructor of class C.

(2) If C is a subclass of class D, C *is a behavioral subtype of* D.

The details of the definition are given in our previous work [25].

Because we have extended the framework with new ingredient, i.e., axioms, we have to extend our definition for the correctness of a program.

Definition 4 (Correct Program). *Program G is* correct*, if*

(1) G is a well-axiom-constrained.

(2) Each class C defined in G is correct.

5 Experiments

Now, having the enriched VeriJ language and framework with the axiom mechanism, we will 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 Extended Specification of the MVC

In Section 2, following the guidance of given informal specification of the referred MVC architecture in **Fig. 3**, we declare interfaces *MI*, *CI*, *VI* in **Fig. 4** to embody the architecture, and introduce some specification predicates (abstract symbols) to form a foundation for formal specification of the architecture. Predicates like $\mathit{model}(m, vs, st)$, $\mathit{controller}(c, m, st)$, $\mathit{view}(v, m, st)$ and $\mathit{VSet}(m, vs, st)$, are used to specify the components (i.e., model, controller, view) and the view set of the model respectively. And also predicates are used to specify some combinations of the component, as $MVs(m, vs, st)$, $MC(c, m, st)$ for two aggregate structures and $MVC(c, vs, m, st)$ for the whole system. Logical variables (parameters, which are also symbols) are used with some specific intentions, like m for a model, v for a view, c a controller, st a state, vs the view set. To reflect the synchronous evolution of the components in a MVC architecture when they respond user inputs, we use abstract parameter st in each component.

The specification predicates with signatures are declared in the interfaces respectively with their first parameters modified to $\mathbf{this}$, because we require this parameter to refer to the current object. For example, $\mathit{model}(\mathbf{this}, vs, st)$ is declared to hide the implementing information of the model object, a component. In logical inference on this level, predicates are only abstract symbols. In addition, as said before, these predicates do not have concrete meaning until some implementations of the architecture are given, while multiple implementations are allowed.

However, not all definitions for these predicates are acceptable, and some predicates have interrelations with one another. For constraining definitions of the predicates in implementations to reflect our anticipation and prevent wrong implementations, we specify a set of axioms as follows which are labeled as [a1-a5]. The axioms form a part of specifications for capturing the important interactions and the intentional properties of the whole MVC architecture. Semantically, any implementation of the MVC architecture should fulfill these axioms.

$$
\begin{aligned}
&\mathbf{axiom}\ MVC(c, vs, m, st) \Leftrightarrow MC(c, m, st) * \mathit{VSet}(m, vs, st) && \text{[a1]}\\
&\qquad \Leftrightarrow \mathit{model}(m, vs, st) * \mathit{VSet}(m, vs, st) * \mathit{controller}(c, m, st) &&\\
&\qquad \Leftrightarrow MVs(m, vs, st) * \mathit{controller}(c, m, st); &&\\
&\mathbf{axiom}\ MVs(m, vs \cup \{v\}, st) \Leftrightarrow MVs(m, vs, st) * \mathit{view}(v, m, st) && \text{[a2]}\\
&\qquad \Leftrightarrow \mathit{model}(m, vs \cup \{v\}, st) * \mathit{VSet}(m, vs \cup \{v\}, st); &&\\
&\mathbf{axiom}\ MVs(m, \emptyset, st) \Leftrightarrow \mathit{model}(m, \emptyset, st); && \text{[a3]}\\
&\mathbf{axiom}\ \mathit{VSet}(m, vs \cup \{v\}, st) \Leftrightarrow \mathit{VSet}(m, vs, st) * \mathit{view}(v, m, st); && \text{[a4]}\\
&\mathbf{axiom}\ MC(c, m, st) \Leftrightarrow \mathit{model}(m, vs, st) * \mathit{controller}(c, m, st); && \text{[a5]}
\end{aligned}
$$

Now, although the interfaces do not provide defined behaviors, the predicates are still abstract, and different interfaces and predicates have been connected formally by these axioms. It happens that, the properties expressing by the axioms turn to constraints on the three type's forthcoming implementations. That is, implementations of the methods which are declared in these interfaces and specified based on the abstract predicates must obey these constraints, that will produce some proof obligations.

Then in **Fig. 5** we define four classes Model, $\mathit{Controller}$, View and View_2 to implement the interfaces, that form an instance of MVC. All predicates declared in the interfaces are defined with bodies to hide implementing details, that give also specific meaning for the axioms. For example, axiom [a1] means that the whole MVC structure can either be divided into a model object, its controller object and view object set, or a model-views aggregate structure with a controller object, or a model-controller aggregate structure with a view object set; axiom [a2] means that the model-views aggregate structure consists of a model object and its view object set, in another way, it contains a separated view object and the combination of the model with its other view set; [a3] says that when no view attaching on a model object, the model-views aggregate structure is just the model object; [a4] says that all views in the model's view set are disjoined; [a5] says that the model-controller aggregate structure is composed of a model object and its controller object. Importantly, parameters st in each axiom must keep the same and reflect the synchronous change among three components to respond user inputs.

Now, we have encoded an implementation of our MVC architecture while its code part can be compiled and debugged in Java system and indeed acts like a MVC design. However, it is neither enough nor safe to conclude before a formal verification for this implementation has been done as required in our new definition of *correct program*, and can not apply it in client code verification. Therefore, by our **Definition. 4**, two parts of the work should be done: (1) checking this implementation supporting axioms [a1-a5] to assure the MVC architecture's requirements are satisfied; (2) checking each declared method satisfying its specifications;

5.2 The Implementations Supporting Axioms

In this subsection we verify the well-supporting property of the axioms with respect to the given MVC implementation according to the **Definition. 1**.

For axiom [a1], we unfold definitions of $MVC(c, vs, m, st)$, $MC(c, m, st)$ in class $Controller$ and $MVs(m, vs, st)$ in class $Model$ during the proofs:

$$
\begin{array}{ll}
\quad MVC(c, m, vs, st) & \\
\Leftrightarrow \mathit{model}(m, vs, st) * \mathit{VSet}(m, vs, st) * \mathit{controller}(c, m, st) & \text{[Def. of } MVC(c, vs, m, st)\text{–(1)]} \\
\Leftrightarrow MVs(m, vs, st) * \mathit{controller}(c, m, st) & \text{[Def. of } MVs(m, vs, st)\text{–(2), from (1)]} \\
\Leftrightarrow MC(c, m, st) * \mathit{VSet}(m, vs, st) & \text{[Def. of } MC(c, m, st)\text{, or Axiom [a5]–(3), from (1)]}
\end{array}
$$

Thus, we can conclude that, axiom [a1] is well supported by our simple implementation.

In the following, we give only the proof processes without more explanations.

For axiom [a2], we have deduction as follows:

$$
\begin{array}{ll}
\quad MVs(m, vs \cup \{v\}, st) & \\
\Leftrightarrow \mathit{model}(m, vs \cup \{v\}, st) * \mathit{VSet}(m, vs \cup \{v\}, st) & \text{[Def. of } MVs(m, vs, st)\text{]} \\
\Leftrightarrow \mathit{model}(m, vs, st) * (\mathit{VSet}(m, vs, st) * \mathit{view}(v, m, st)) & \text{[Def. of } VSet(m, vs, st)\text{, or Axiom [a4]]} \\
\Leftrightarrow (\mathit{model}(m, vs, st) * \mathit{VSet}(m, vs, st)) * \mathit{view}(v, m, st) & \text{[Inf. rule in SL } (p * q) * r \Leftrightarrow p * (q * r)\text{]} \\
\Leftrightarrow MVs(m, vs, st) * \mathit{view}(v, m, st) & \text{[Def. of } MVs(m, vs, st)\text{]}
\end{array}
$$

For axiom [a3], we have:

$$
\begin{array}{ll}
\quad MVs(m, \emptyset, st) & \\
\Leftrightarrow \mathit{model}(m, \emptyset, st) * \mathit{VSet}(m, \emptyset, st) & \text{[Def. of } MVs(m, vs, st)\text{]} \\
\Leftrightarrow \mathit{model}(m, \emptyset, st) * \mathbf{emp} & \text{[Def. of } VSet(m, vs, st)\text{, or Axiom [a4]]} \\
\Leftrightarrow \mathit{model}(m, \emptyset, st) &
\end{array}
$$

For axiom [a4]:

$$VSet(m, vs \cup \{v\}, st) \Leftrightarrow VSet(m, vs, st) * view(v, m, st) \qquad \text{[Def. of } VSet(m, vs, st)\text{]}$$

For axiom [a5]:

$$MC(c, m, st) \Leftrightarrow model(m, vs, st) * controller(c, m, st) \qquad \text{[Def. of } MC(c, m, st)\text{]}$$

In conclusion, the axioms we specified for the MVC architecture are all well-supported by this implementation. Therefore, they can be directly applied in verifications of the implementation and its clients code from now on.

Then we need to verify that each method is correct wrt. its specifications. Because this is not the main focus here, we leave the verification details of most methods in our report [**?**] but give the proofs for methods $View.View(m)$ and $CI.userInput(b)$ as following which are used in the illustrating client code. Moreover, axioms that proven to be supported can also be used to help the deduction.

Proving $View.View(\mathbf{m})$:	Proving $CI.userInput(\mathbf{b})$:
$\{MVs(m, vs, st)\}$	$\{MVC(\mathbf{this}, vs, m, -)\}$
$\mathbf{this}.model = m;$	$\{MVs(m, vs, -) * controller(\mathbf{this}, m, -)\}$ [Axiom [a1]]
$\mathbf{this}.state = 0;$	$\{MVs(m, vs, -) * \mathbf{this}.model \mapsto m*$
$\{MVs(m, vs, st) *$	$\mathbf{this}.state \mapsto -\}$ [Def. of $controller(c, m, st)$]
$\mathbf{this}.model \mapsto m *$	$\mathbf{this}.state = b;$
$\mathbf{this}.state \mapsto 0\}$	$\{MVs(m, vs, -) * \mathbf{this}.model \mapsto m * \mathbf{this}.state \mapsto b\}$
$\{MVs(m, vs, st)*$	$\{MVs(m, vs, -) * controller(\mathbf{this}, m, b)\}$
$view(\mathbf{this}, m, 0)\}$	[Def. of $controller(c, m, st)$]
[Def. of $view(v, m, st)$]	$model.update(b);$
$m.addView(\mathbf{this});$	$\{MVs(m, vs, b) * controller(\mathbf{this}, m, b)\}$
$\{MVs(m, vs \cup \{\mathbf{this}\}, st)\}$	$\{MVC(\mathbf{this}, vs, m, b)\}$ [Axiom [a1]]

Thus, with all axioms well-supported and all methods satisfying their specifications, we can conclude that the classes form a correct implementation for the MVC architecture. Formal specification and verification of more variants of MVC architecture in VeriJ are left in our future work.

5.3 Verifying Clients Using Axioms

At last, we show how to solve the problems mentioned during verifying the client code segment in **Fig. 6**, by using the extended formal specifications including the axioms of our correct implementation of the describing MVC. The related deduction is given in **Fig. 8**. Now it is easy for the client to finish the verification of this code segment with the help of the axioms and thus we can conclude that it makes a correct application of our implementation of MVC architecture.

6 Conclusion and Related Work

In this work, we provide our approach to formally specify a sort of MVC architecture using our global axiom mechanism and also formally verify one implementation

......

(5.) $\{MC(c, m, 0)\}$

(5″.) $\{model(m, \emptyset, 0) * controller(c, m, 0)\}$ [Axiom [a5]]

(5′.) $\{\underline{MVs(m, \emptyset, 0)} * controller(c, m, 0)\}$ [Axiom [a3]]

(6.) $VI\ v_1 = \mathbf{new}\ \ View(m);$ // add a new view

(7.) $\{MVs(m, \{v_1\}, 0) * controller(c, m, 0)\}$

(7′.) $\{\underline{MVC(c, \{v_1\}, m, 0)}\}$ [Axiom [a1]]

(8.) $c.userInput(9);$ // process a user input

(9.) $\{MVC(c, \{v_1\}, m, 9)\}$

(9′.) $\{\underline{MVs(m, \{v_1\}, 9)} * controller(c, m, 9)\}$ [Axiom [a1]]

(10.) $VI\ v_2 = \mathbf{new}\ \ View_2(m);$ // add another view of model

(11.) $\{MVs(m, \{v_1, v_2\}, 9) * controller(c, m, 9)\}$

(11′.) $\{\underline{VSet(m, \{v_1, v_2\}, 9) * model(m, \{v_1, v_2\}, 9) *}$
$controller(c, m, 9)\}$ [Axiom [a2]]

(11″.) $\{VSet(m, \{v_2\}, 9) * \underline{view(v_1, m, 9) *}$
$\underline{model(m, \{v_1, v_2\}, 9)} * controller(c, m, 9)\}$ [Axiom [a4]]

(12.) $v_1.paint(m);$

......

Fig. 8. The Correct Proof of the Client Code Segment

of the MVC. The contributions of this work are: (1) enrich our specification language with *application axiom*s; (2) define the semantics of our axioms reflecting two effects: 1) an axiom formally and globally specifying a system's design requirement (maybe interactions as in MVC architecture) constrain any implementation of the system by relating isolated predicate signatures in different implementing types. 2) an axiom well-supported by an implementation is trivially a $true$ formula for verifying the implementation and its client's code. Thus it can help deducing between different objects without caring about their detail information hiding in predicate definitions; (3) extend our formal definition of a correct program; (4) specify a simple MVC architecture and give it a correct implementation by using our extended framework. Also our approach for specifying and verifying programs (including interactive ones) are modular and extensible.

Our axioms have some similar aspects to the axioms in algebraic specification technique [12], such as both: (1) are extracted from informal requirements of design and well-specified in an abstract hierarchy before concrete implementations; (2) specify some constraints on later implementations. However, they are incompletely the same. (1) Axioms in an algebraic specification mainly describe relationships among some behaviors defined in abstract data types, whereas ours specify relations of/among some individual/interactive objects of an architecture design; (2) The members appearing in the former would be implemented to instance methods in a program, while ours are embodied to be abstract predicates for concrete types or data structures; (3) Axioms in algebraic specifications are of equations defined in some types' interfaces, while ours have no restricted form which may be such as equalities, bidirectional or unidirectional implications defined outside all declared types. Therefore, our axioms specified for the whole system is more abstract than those in algebraic specifications.

Comparing to the Object-Z specification technique of MVC architecture in [1] and [2], we described relations among interactive components using abstract symbols which are later declared as specification predicates to hide concrete implementing information rather than operations. And we freely place these relationships in a global axiom block which is beyond any declared type hierarchy to constrain the whole system's implementations rather than in some special aggregate class. Fatherly, we propose an approach of proving the correctness of implementations together with axioms which is not concerned by these early work. However, our approach can not describe concurrency relations in intuitive expressions like "$a \parallel b$" in Object-Z. We adopt recording all the components' states in abstract symbols (predicate signatures) in axioms to track the synchronization of several interactive objects in MVC.

Additionally, MultiStar [13] proposed *export* and *axiom* clauses under the intuitional SL semantics to correspondingly express properties holding for individual classes and the entire hierarchies in Eiffel. Similarly, axioms of MultiStar and us both offer general approach to constrain the implementation of abstractions in subclasses. However, since multiple inheritance is not allowed in Java, our axioms can not specify this kind of programs as MultiStar did. Programs of interactive systems like MVC architecture are not considered in MultiStar while we can succeed in modularly specifying and verifying this kind of systems and also programs with inheritance hierarchies.

As future work, we want to apply our extended framework with global application axioms in more aspects, including studying and specifying more complex interactive systems, deeply researching both the behavioral subtyping and modularity properties of programs with axioms, and also developing a tool to support our framework for convenient usage in large systems.

References

1. A. Hussey, and D. Carrington. Comparing the MVC and PAC architectures: a formal perspective *IEE Proc. Software Engineering*, 144(4):224-236, 1997.
2. Jun Hu. *Design of a Distributed Architecture for Enriching Media Experience in Home Theaters*. PhD thesis, Technische Universiteit Eindhoven, 2006
3. Cay S. Horstmann and Gary Cornell. *Core Java. Volume 1, Foundamentals, Eight edition*. Sun Microsystems, Inc., 2008.
4. E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns, Elements of Reusable Object-Oriented Software*. Addlison Wesley, 1994.
5. T. Mikkonen. Formalizing Design Patterns. *Proceedings of the International Conference on Software Engineering*, 19-25, 1998.
6. A. H. Eden. Precise Specification of Design Patterns and Tool Support in Their Application. PhD dissertation, Department of Computer Science, Tel Aviv University, 2000.
7. Dae-Kyoo Kim and Charbel El Khawand. An approach to precisely specifying the problem domain of design patterns. *Journal of Visual Languages & Computing*, 18:560-591, 2007.
8. Jonathan Nicholson, Epameinondas Gasparis, Amnon H. Eden and Rick Kazman. Automated Verification of Design Patterns with LePUS3. Proceedings of the First NASA Formal Methods Symposium, 76-85, 2009.
9. Neelakantan R. Krishnaswami, Jonathan Aldrich, Lars Birkedal, Kasper Svendsen, and Alexandre Buisse. Design Patterns in Separation Logic. *TLDI*, 105-115, 2009.

10. Neelam Soundarajan and Jason O. Hallstrom. Responsibilities and Rewards: Specifying Design Patterns. Proceedings of the 26th International Conference on Software Engineering, 1-10, 2004.
11. Toufik Taibi and David Chek Ling Ngo. Formal Specification of Design Patterns-A Balanced Approach. *JOURNAL OF OBJECT TECHNOLOGY*, 2(4):127-140, 2003.
12. J. V. Guttag, E. Horowitz, and D. R. Musser. Abstract data types and software validation. *Commun. ACM*, 21:1048C1064, 1978.
13. Stephan van Staden and Cristiano Calcagno. Reasoning about Multiple Related Abstractions with MultiStar. *ACM SIGPLAN Notices-OOPSLA/SPLASH*, 45:504-519, 2010.
14. M. Barnett, M. Fähndrich, K.R.M. Leino, P. Müller, W. Schulte, and H. Venter. Specification and verification: the spec# experience. *Communications of the ACM*, 54(6):81–91, 2011.
15. Wei-Ngan Chin, Cristina David, Huu Hai Nguyen, and Shengchao Qin. Enhancing modular OO verification with separation logic. In *POPL '08*, pages 87–99, New York, NY, USA, 2008. ACM.
16. D. Distefano and M.J. Parkinson J. jstar: Towards practical verification for java. *ACM SIGPLAN Notices*, 43(10):213–226, 2008.
17. G. Leavens. JML's rich, inherited specifications for behavioral subtypes. *Formal Methods and Software Engineering*, pages 2–34, 2006.
18. Barbara Liskov and Jeannette M. Wing. A behavioral notion of subtyping. *ACM Transactions on Programing Languages and Systems*, 16(6):1811–1841, 1994.
19. Matthew J. Parkinson and Gavin M. Bierman. Separation logic, abstraction and inheritance. In *POPL '08*, pages 75-86, 2008. ACM.
20. A. Poetzsch-Heffter. Specification and verification of object-oriented programs. *Technische Universität München*, 1997.
21. Jan Smans, Bart Jacobs, and Frank Piessens. Implicit dynamic frames: Combining dynamic frames and separation logic. In Sophia Drossopoulou, editor, *ECOOP 2009*, volume 5653 of *LNCS*, pages 148–172. Springer, 2009.
22. Qiu Zongyan, Wang Shuling and Long Quan. Sequential μJava: Formal Foundations. Avaliable at http://www.mathinst.pku.edu.cn/index.php?styleid=2, Preprints, number 2007-35, 2007.
23. Liu Yijing, Hong Ali, and Qiu Zongyan. Inheritance and modularity in specification and verification of OO programs. In *TASE 2011*, pages 19–26. IEEE Computer Society, 2011.
24. Liu Yijing and Qiu Zongyan. A separation logic for OO programs. In *FACS 2010*, volume 6921 of *LNCS*, pages 88–105. Springer, 2011.
25. Qiu Zongyan, Hong Ali, and Liu Yijing. Modular verification of OO programs with interface types. In *ICFEM 2012*, 7635:151-166, 2012
26. Qiu Zongyan, Hong Ali, and Liu Yijing. Modular verification of OO programs with interface types. Technical report, School of Math., Peking Univ., 2012. Avaliable at http://www.mathinst.pku.edu.cn/download.php?classid=22.

A Other Method Verifications

In this appendix, we give details of verifying methods with their specifications in our implementation (in Fig. 4, 5) of a MVC architecture. Specially, verifications of two methods $View.View(m)$ and $CI.userInput(b)$ 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 inherit their specifications from interfaces, inference rules proposed in our former work [23, 25] for verifying programs with interfaces are applied to prove them. Also, all well-supported axioms listed and well-proven in Section 5 can help our deduction. Note that, there are calls of some methods, such as $Controller.Controller(m)$ and $View.paint(m)$ calling for $Model.getState()$, $Model.addView(v)$ and $Model.update(b)$ calling for $View.paint(m)$. Therefore, we should prove the called methods ahead of the calling ones.

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

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

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

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

```
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 $controller(c, m, st)$]
$\{MC(\mathbf{this}, m, st)\}$
                    [Def. of $MC(m, vs, st)$ or Axiom [a5]]
```

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

Proving $Model.addView(\mathbf{v})$:

$\{MVs(\mathbf{this}, vs, st) * view(v, \mathbf{this}, -)\}$

$\{model(\mathbf{this}, vs, st)*$
$\quad VSet(\mathbf{this}, vs, st) * view(v, \mathbf{this}, -)\}$

[Def. of $MVs(m, vs, st)$ or Axiom[a2]]

$\{\mathbf{this}.views \mapsto vs * \mathbf{this}.state \mapsto st*$
$\quad VSet(\mathbf{this}, vs, st) * view(v, \mathbf{this}, -)\}$

[Def. of $model(m, vs, st)$]

$views.add(v);$

$\{\mathbf{this}.views \mapsto (vs \cup \{v\})*$
$\quad \mathbf{this}.state \mapsto st * VSet(\mathbf{this}, vs, st)*$
$\quad view(v, \mathbf{this}, -)\}$

$\{model(\mathbf{this}, vs \cup \{v\}, st)*$
$\quad VSet(\mathbf{this}, vs, st) * view(v, \mathbf{this}, -)\}$

[Def. of $model(m, vs, st)$]

$v.paint(\mathbf{this});$

$\{model(\mathbf{this}, vs \cup \{v\}, st)*$
$\quad view(v, \mathbf{this}, st) * VSet(\mathbf{this}, vs, st)\}$

$\{model(\mathbf{this}, vs \cup \{v\}, st)*$
$\quad VSet(\mathbf{this}, vs \cup \{v\}, st)\}$

[Def. of $VSet(m, vs, st)$] or Axiom [a4]

$\{MVs(\mathbf{this}, vs \cup \{v\}, st)\}$

[Def. of $MVs(m, vs, st)$] or Axiom [a2]

Proving $Model.update(\mathbf{b})$:

$\{MVs(\mathbf{this}, vs, -)\}$

$\{model(\mathbf{this}, vs, -) * VSet(\mathbf{this}, vs, -)\}$

[Def. of $MVs(m, vs, st)$ or Axiom [a2]]

$\{\mathbf{this}.views \mapsto vs * \mathbf{this}.state \mapsto -*$
$\quad VSet(\mathbf{this}, vs, -)\}$

[Def. of $model(m, vs, st)$]

$\mathbf{this}.state = b;$

$\{\mathbf{this}.views \mapsto vs * \mathbf{this}.state \mapsto b*$
$\quad VSet(\mathbf{this}, vs, -)\}$

$\{model(\mathbf{this}, vs, b) * VSet(\mathbf{this}, vs, -)\}$

[Def. of $model(m, vs, st)$]

$\mathbf{for}(\,VI\ v : views)$
$\quad \{\exists vs_1, v, vs_2 \cdot vs = vs_1 \cup \{v\} \cup vs_2 \wedge$
$\qquad model(\mathbf{this}, vs, b) * VSet(\mathbf{this}, vs_1, b)*$
$\qquad view(v, \mathbf{this}, -) * VSet(\mathbf{this}, vs_2, -)\}$
$\{\quad v.paint(\mathbf{this});$
$\quad \{\exists vs_1, v, vs_2 \cdot vs = vs_1 \cup \{v\} \cup vs_2 \wedge$
$\qquad model(\mathbf{this}, vs, b) * VSet(\mathbf{this}, vs_1, b)*$
$\qquad view(v, \mathbf{this}, b) * VSet(\mathbf{this}, vs_2, -)\}$
$\quad \{\exists vs'_1, v', vs'_2 \cdot vs = vs'_1 \cup \{v'\} \cup vs'_2 \wedge$
$\qquad vs'_1 = vs \cup \{v\} \wedge vs_2 = vs'_2 \cup \{v\} \wedge$
$\qquad model(\mathbf{this}, vs, b) * VSet(\mathbf{this}, vs'_1, b)*$
$\qquad view(v', \mathbf{this}, -) * VSet(\mathbf{this}, vs'_2, -)\}$
$\}$

$\{model(\mathbf{this}, vs, b) * VSet(\mathbf{this}, vs, b)\}$

$\{MVs(\mathbf{this}, vs, b)\}$

[Def. of $MVs(m, vs, st)$ or Axiom [a2]]

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.

B Another Suggestion to Write Specifications

In this appendix, we illustrate another approach to specify the MVC architecture as talked in Section 2. The basic idea is, rather than specifying the members of aggregate structures among the MVC architecture totally in method specifications, we specify them separately and explicitly using $*$ operator in OOSL as below. Thus, predicates like MVs, MC and MVC are removed while $VSet$ is kept as need of recursively depicting the view set of a model.

The specifications to change are lying in the three interfaces and also the constructors of implementing classes, while other part such as reserved predicates' declarations and also definitions remains the same as before. Therefore, we simply give new specifications in **Fig. 9** and implementing information is omitted here. New specifications also use declared abstract predicate signatures like $model(\ldots)$, $view(\ldots)$, $controller(\ldots)$ and $VSet(\ldots)$ to hide information. For example, the precondition for method $CI.userInput(b)$ changes to $model(m, vs, -) * VSet(m, vs, -) * controller(\mathbf{this}, m, -)$ from $MVC(\mathbf{this}, vs, m, -)$ in Section 2.

```
$\textbf{interface}\ \mathit{MI}\ \{$
  $\textbf{def}\ \mathit{model}(\textbf{this}, vs, st);$
  $\ldots$
  $\textbf{void}\ \mathit{addView}(\mathit{VI}\ v)$
  $\langle v \notin vs \wedge \mathit{model}(\textbf{this}, vs, st) *$
    $\mathit{VSet}(\textbf{this}, vs, st) * \mathit{view}(v, \textbf{this}, -)\rangle$
  $\langle \mathit{model}(\textbf{this}, vs \cup \{v\}, st) *$
    $\mathit{VSet}(\textbf{this}, vs \cup \{v\}, st)\rangle;$
  $\textbf{Int}\ \mathit{getState}()\ \langle \mathit{model}(\textbf{this}, vs, st)\rangle$
  $\langle \mathit{model}(\textbf{this}, vs, st) \wedge \mathsf{res} = st\rangle;$
  $\textbf{void}\ \mathit{update}(\textbf{Int}\ b)$
  $\langle \mathit{model}(\textbf{this}, vs, -) * \mathit{VSet}(\textbf{this}, vs, -)\rangle$
  $\langle \mathit{model}(\textbf{this}, vs, b) * \mathit{VSet}(\textbf{this}, vs, b)\rangle;$
$\}$
$\textbf{interface}\ \mathit{VI}\ \{$
  $\ldots$
  $\textbf{void}\ \mathit{paint}(\mathit{MI}\ m)$
  $\langle \mathit{view}(\textbf{this}, m, -) * \mathit{model}(m, vs, st)\rangle$
  $\langle \mathit{view}(\textbf{this}, m, st) * \mathit{model}(m, vs, st)\rangle;$
$\}$
$\textbf{interface}\ \mathit{CI}\{$
  $\ldots$
  $\textbf{void}\ \mathit{userInput}(\textbf{Int}\ b)$
  $\langle \mathit{model}(m, vs, -) * \mathit{VSet}(m, vs, -) *$
    $\mathit{controller}(\textbf{this}, m, -)\rangle$
  $\langle \mathit{model}(m, vs, b) * \mathit{VSet}(m, vs, b) *$
    $\mathit{controller}(\textbf{this}, m, b)\rangle;$
$\}$
```

```
$\textbf{class}\ \mathit{Model}\ \triangleright \mathit{MI}\{$
  $\ldots$
  $\mathit{Model}()\langle \textbf{emp}\rangle\langle \mathit{model}(\textbf{this}, \emptyset, 0)\rangle$
  $\ldots$
$\}$
$\textbf{class}\ \mathit{View}\ \triangleright \mathit{VI}\{$
  $\ldots$
  $\mathit{View}(\mathit{MI}\ m)$
  $\langle \mathit{model}(m, vs, st) * \mathit{VSet}(m, vs, st)\rangle$
  $\langle \mathit{model}(m, vs \cup \{\textbf{this}\}, st) *$
        $\mathit{VSet}(m, vs \cup \{\textbf{this}\}, st)\rangle$
  $\ldots$
$\}$
$\textbf{class}\ \mathit{View}_2\ \triangleright \mathit{VI}\{$
  $\ldots$
  $\mathit{View}_2(\mathit{MI}\ m)$
  $\langle \mathit{model}(m, vs, st) * \mathit{VSet}(m, vs, st)\rangle$
  $\langle \mathit{model}(m, vs \cup \{\textbf{this}\}, st) *$
        $\mathit{VSet}(m, vs \cup \{\textbf{this}\}, st)\rangle$
  $\ldots$
$\}$
$\textbf{class}\ \mathit{Controller}\ \triangleright \mathit{CI}\{$
  $\ldots$
  $\mathit{Controller}(\mathit{MI}\ m)$
  $\langle \mathit{model}(m, vs, st)\rangle\langle \mathit{model}(m, vs, st) *$
    $\mathit{controller}(\textbf{this}, m, st)\rangle$
  $\ldots$
$\}$
```

Fig. 9. Various Specifications for our MVC Architecture Implementation

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

```
Proving $\mathit{CI}.\mathit{userInput}(\mathbf{b})$ :
$\{\mathit{model}(m, vs, -) * \mathit{VSet}(m, vs, -) *$
  $\mathit{controller}(\textbf{this}, m, -)\}$
$\{\mathit{model}(m, vs, -) * \mathit{VSet}(m, vs, -) *$
  $\textbf{this}.\mathit{model} \mapsto m * \textbf{this}.\mathit{state} \mapsto -\}$
                                        [Def. of $controller(c, m, st)$]
$\textbf{this}.\mathit{state} = b;$
$\{\mathit{model}(m, vs, -) * \mathit{VSet}(m, vs, -) *$
  $\textbf{this}.\mathit{model} \mapsto m * \textbf{this}.\mathit{state} \mapsto b\}$
$\{\mathit{model}(m, vs, -) * \mathit{VSet}(m, vs, -) *$
  $\mathit{controller}(\textbf{this}, m, b)\}$
                                        [Def. of $controller(c, m, st)$]
$\mathit{model}.\mathit{update}(b);$
$\{\mathit{model}(m, vs, b) * \mathit{VSet}(m, vs, b) *$
  $\mathit{controller}(\textbf{this}, m, b)\}$
```

It is easy to prove all methods with their various specifications to be correct as we do in Appendix A, because the deduction is actually similar to anther group specifications.

However, the proving process would be longer than before since each component object which could be combined with others into an abstract aggregate structure is suggested to specify explicitly in heap assertions. We just illustrate and compare the proofs of two methods $View.View(m)$ and $Controller.userInput(b)$ as above.

Now, with this various specification, we reverify the same client code as in Section 2 and find that the problem in line (11') from (11) (Seeing Fig. 10) could neither be well solved. Because we define abstract predicate $VSet$ to recursively record all views of the model object, without more information specified publicly, clients could not deduce on $VSet$'s signature in abstract method specification to get the required view object.

```
(1.)    $\{\mathbf{true}\}$
(2.)    $MI\ m = \mathbf{new}\ \ Model();$                                  // create a model
(3.)    $\{model(m, \emptyset, 0)\}$
(4.)    $CI\ c = \mathbf{new}\ \ Controller(m);$                            // create a controller of $m$
(5.)    $\{model(m, \emptyset, 0) * controller(c, m, 0)\}$
(6.)    $VI\ v_1 = \mathbf{new}\ \ View(m);$                                // add a new view
(7.)    $\{model(m, \{v_1\}, 0) * VSet(m, \{v_1\}, 0) * controller(c, m, 0)\}$
(8.)    $c.userInput(9);$                                                   // process a user input
(9.)    $\{model(m, \{v_1\}, 9) * VSet(m, \{v_1\}, 9)*$
            $controller(c, m, 9)\}$
(10.)   $VI\ v_2 = \mathbf{new}\ \ View_2(m);$                              // add another view
(11.)   $\{model(m, \{v_1, v_2\}, 9) * VSet(m, \{v_1, v_2\}, 9)*$
            $controller(c, m, 9)\}$
              $\Downarrow$ ???
(11'.)  $\{model(m, \{v_1, v_2\}, 9) * \underline{VSet(m, \{v_2\}, 9)*}$
            $\underline{view(v_1, m, 9)} * controller(c, m, 9)\}$
(12.)   $v_1.paint(m);$                                                     // paint the view $v_1$
(13.) ...
```

Fig. 10. Verification of the Client Code Segment with Various Specifications

Therefore, similar to the approach in Section 5 we propose to solve the problems in Section 2, we specify an additional axiom same as the axiom [a4] in Section 5, to depict the property of model's view set for clients.

$$\mathbf{axiom}\ \ VSet(m, vs \cup \{v\}, st) \Leftrightarrow VSet(m, vs, st) * view(v, m, st)$$

Because the definition of predicate $VSet(m, vs, st)$ remains as before, checking the well-supported property of this axiom in the implementation is the same as we do in Section 5 for axiom [a4]. Also afterwards, this axiom can be applied in verifications.

To conclude, the implementation with its various specifications and axioms is also correct. And by using this additional axiom, deducing (11') from (11) in Fig. 10 can also be directly done, and we omit the details. Thus, this client code is also a correct application of the MVC architecture.