Comparison between Model Fields and Abstract Predicates

Ke Zhang, Zongyan Qiu
LMAM and Department of Informatics
School of Math, Peking University
{zksms,zyqiu}@pku.edu.cn

***Abstract*—To modularly specify and verify object oriented programs on some abstract level, we need abstraction techniques to hide the implementation details. *Model fields* and *abstract predicates* are two most important approaches to address the requirements. In this paper, we compare them about their expressiveness and other aspects. We develop a set of rules and translation algorithms, which can translate a program with model fields based specification to one with abstract predicates based specification. We prove that the translation is correct, and the resulting specifications are well-encapsulated and well-formed. We prove that the abstract predicates technique is more expressive. On the other hand, model fields are easier to use and more useful in automatic verification. In addition, we discuss the characteristics of the two approaches with respect to framing, inheritance, and recursion.**

***Index Terms*—OO, Verification, Abstraction, Model Field, Abstract Predicate.**

I. Introduction

Object Orientation (abbr. OO) is widely used in software development. Due to the increasing demand on the reliability and correctness, techniques for specifying/verifying OO programs attract more attention recently.

However, to specify/verify OO programs, we have to face the challenges brought by encapsulation, inheritance and polymorphism, which are the core features of OO. Abstraction is the key idea to address these issues. Introducing suitable abstraction mechanisms into the formal notations is helpful to enhance the modularity of OO specification and verification. In addition, these mechanisms are also useful for information hiding and specification re-use. All of these characteristics are crucial in verifying large-scale OO programs.

Researchers have paid much attention to abstraction mechanisms in the OO verification for years. *Model fields* [8] and *abstract predicates* [17] are two important techniques here. Model fields are abstract fields whose values are determined by concrete fields of the object. Abstract predicates are user-defined logic abstractions. Both techniques allow us to specify the program abstractly, thus support information hiding.

Model fields and abstract predicates have been studied for years, and both techniques address the similar problems. However, no systematic and formal study has been published on their relationship and advantages/disadvantages. We try to make such a comparison in this paper. We prove that the abstract predicates are more expressive than model fields, by showing: (1) the model fields based specification can be translated into abstract predicates based specification; (2) there exists programs which can only be specified by abstract predicates.

The rest of the paper is organized as follows: Section II introduces the model fields and abstract predicates techniques and compares them briefly; Section III presents the translation rules and algorithms, together with some discussion; Section IV summarizes some related work and Section V concludes.

II. Background

In this section, we introduce the main ideas of *model fields* and *abstract predicates* techniques and briefly compare them.

A. Model Fields

The *model fields* (or *abstract fields*) technique was introduced in 1990's [11] to deal with data abstraction and data refinement. Syntactically, model fields are a kind of fields in class declarations which can only be used in the specifications, but cannot be accessed by real code.

We take JML [8] as an example, which uses model fields for specification abstraction[1]:

Definition 1 (Model fields)**.** *A model field declaration in class C takes the form:*

$$\textbf{model}\ T\ f\ \textbf{such\_that}\ R_C(f, \overline{f}, \overline{g});$$

Here T is a type, f is the field name, and the **such_that** *clause introduces a constraint $R_C(f, \overline{f}, \overline{g})$ which must hold in every state. The constraint could mention the model fields $\overline{f}$ and concrete fields $\overline{g}$ of class C.*

Subclass D inherits all the model fields and constraints from its superclass. It can add a new constraint $R_D(\ldots)$ for an inherited model field. Both the inherited constraints and the new constraint $R_D(\ldots)$ must hold in every state.

The official references of JML lack formal semantics, but [2] gives formal semantics for model fields in JML as follows:

Definition 2 (Model field valuator)**.** *Let $\mathcal{S}$ be the set of states, $\mathcal{U}$ be the set of objects, $\mathbb{I}$ be the set of concrete and model fields in classes. A function $M : \mathcal{S} \times \mathcal{U} \times \mathbb{I} \to \mathcal{U}$ is called a* model field valuator *iff:*

[1]Here we omit the privacy modifiers and exceptions. We assume that all methods are public, and all concrete fields are private. For model fields, all model fields are public, but their constraints are private. Besides, we do not allow throwing exceptions when evaluating the constraints of model fields.

For every state $s \in \mathcal{S}$ and every model field $o.f$ with constraint ψ (ψ denotes the conjunction of all the constraints which $o.f$ must conform to):

- *The type of object $M(s,o,x)$ is a subtype of the declaration type of field $o.x$,*
- *$\psi[\overline{M(s,o,f)/o.f}] = \mathbf{true}$ in state s.*

Here $[M(s,o,f)/o.f]$ means replacing $o.f$ with $M(s,o,f)$. Taking model fields into account, specifications are always meant to hold for every possible valuator[2].

For example, a model fields based method specification (pre and post-conditions) $\langle\phi\rangle\langle\psi\rangle$ specifies $C.m()$, iff for every possible valuator M, the concrete fields based specification $\langle\phi[M(s,o,f)/o.f]\rangle\langle\psi[M(s,o,f)/o.f]\rangle$ specifies $C.m()$.

When the method specifications in a class only mention model fields but not concrete fields, an abstraction boundary is formed which separates the real code (and concrete fields) from the specification and hides the implementation details. And the constraints of model fields serve as a bridge to connect the specification and the implementation. To express this idea, we have the following definition:

Definition 3 (Well-encapsulated)**.** *If the method specifications (pre and post-conditions) in a class do not mention any concrete field, we call it a* well-encapsulated specification *or an* abstract specification *of the class.*

Notice that we do not forbid the invariants to mention concrete fields in Definition 3, because the invariants are always seen as conditions to constrain the valid object states. In the view of modular programming, those invariants should not be concerned by other objects. In fact, the example in **Fig. 1**, which comes from literature on model fields [13], uses concrete fields in its invariant. Thus we allow the invariants to mention concrete fields in Definition 3.

Fig. 1 shows a class Rec specified by model fields technique. Rec uses the coordinates of two opposite corners to represent a rectangle, and declares two model fields $width$ and $height$ as the abstraction of the concrete fields.

In the class, the method $ScaleH$ scales the rectangle horizontally, which is specified using model field $width$. In addition, the **modifies** clause lists all the (concrete and model) fields modified by the method, which is useful when reasoning about frame properties of the method.

B. Abstract Predicates

Abstract predicates are proposed when researchers apply Separation Logic [20] into OO area [17]. Abstract predicates are basically user-defined predicates in class declarations. The predicates with same name (but in different classes) form an *abstract predicate family*.

Definition 4 (Abstract predicates)**.** *An abstract predicate definition in class C takes the following form:*

$$\mathbf{define}\ p_C(\mathbf{this}, \overline{v}) : \psi;$$

[2]Here we require the constraints of model fields to be satisfiable, i.e. there exists at least one possible valuator.

```
class Rec : Object{
  int x1, y1, x2, y2;
  invariant x1 ≤ x2 ∧ y1 ≤ y2;

  model int width such_that width = x2 − x1;
  model int height such_that height = y2 − y1;

  Rec() ⟨⟩⟨width = 1 ∧ height = 1⟩
  {x1 := 0; y1 := 0; x2 := 1; y2 := 1;}

  void ScaleH(int factor)
  ⟨0 ≤ factor⟩⟨width = old(width) × factor/100⟩
  modifies width, x2;
  { x2 := (x2 − x1) × factor/100 + x1; }
}
```

Fig. 1. *Rec* Written Using Model Fields

The predicate has a name p, parameters $(\mathbf{this}, \overline{v})$, and a definition ψ which is a separation logic assertion. The definition could mention any object o which is accessible from class C, and also the predicates of o (i.e. the predicates whose first parameter is o).

Subclass inherits all the abstract predicates from its superclass, and it can also override the definition of inherited predicates.

Paper [18] defines the semantics as follows. Briefly speaking, when a predicate is used in the specification, its definition is chosen by the dynamic type of its first parameter.

Definition 5 (Assumption context)**.** *We define an assumption set Δ, which includes the following formulas:*

For all class C, predicate p, parameters o and $\overline{v}$,

(1) $o : C \Rightarrow (p(o, \overline{v}) \Leftrightarrow p_C(o, \overline{v}))$

(2) $$\frac{\mathbf{define}\ p_C(\mathbf{this}, \overline{v}) : \psi;}{p_C(o, \overline{v}) \Leftrightarrow \psi[o/\mathbf{this}]}$$

(3) $p(\overline{v}) \Leftrightarrow \exists w \cdot p(\overline{v}, w)$

(4) $p() \Leftrightarrow \mathbf{true}$

When the type of o is C, denoted as $o : C$, we can translate the family to a specific entry for class C. In addition, the entry for class C is equivalent to its definition in class C. The last two rules allow us to change the arity of predicates.

Taking abstract predicates into account, specifications are always meant to hold within the assumption context Δ.

For example, if the proof obligation of a concrete specification ψ is

$$\Gamma \vdash \psi$$

where Γ denotes the static environment, then the proof obligation of an abstract predicates based specification ψ' is

$$\Gamma, \Delta \vdash \psi'.$$

We can use the abstract predicates to write another abstract specification of Rec, as shown in **Fig. 2**.

C. Preliminary Comparison

Both model fields and abstract predicates are commonly used techniques in OO verification. Here we briefly compare their ideas and applicabilities, and leave the comparison of expressiveness to the rest of the paper.

```
class Rec : Object{
  //concrete fields and method bodies are the same as Fig. 1
  define rec_Rec(this, w, h) : w = x2 − x1 ∧ h = y2 − y1;
  Rec() ⟨⟩⟨rec(this, 1, 1)⟩
  void ScaleH(int factor)
  ⟨(0 ≤ factor) ∧ rec(this, w, h)⟩
  ⟨rec(this, w × factor/100, h)⟩
}
```

Fig. 2. *Rec* Written Using Abstract Predicates

In common, the assertions written based on model fields or abstraction predicates can both be easily translated into the assertions on the concrete fields of the objects. We can either unfold the model fields references by their constraints, or unfold the predicates invocations by the predicates' definitions. The real code must satisfy the unfolded assertions. On the other hand, outside the encapsulation, we only use the abstract specification for reasoning.

The model fields are just names referring to some values derived from concrete fields, thus it is a relative simple and easy-to-use mechanism. In addition, because the values of model fields are constrained to concrete fields, their values can be really implemented and automatically updated in the verification or static/dynamic analysis. It makes the model fields useful in the automatic verification or analysis. As another evidence for its usefulness, JML [8] and Spec# [1] integrate model fields into their specification languages to express the abstract specifications of classes and interfaces.

On the other hand, since the model fields have a close connection to the concrete fields, which also limits the way to define them and the flexibility of the specifications, especially for subclasses and overridden methods.

The power of abstract predicates comes from its structure. Firstly, predicates may have arbitrary parameters, which can be instantiated in invocations, or be used to connect different parts of specifications. Furthermore, recursively defined predicates can be naturally constructed,which makes it easy to handle recursive data structures. Furthermore, abstract predicates decouple the specification from implementation details more thoroughly, because the definitions of predicates are about the properties but not the values of the program states. The definitions of the predicates are almost arbitrary even in the presence of inheritance, which provide the expressiveness for the specifications. Due to these advantages, some verification frameworks, like VeriFast [7], implement abstract predicates as its abstraction mechanism.

However, the developers must have more logic training before working with the abstract predicates. In many cases, they must carefully design appropriate predicates to ensure the correctness of the specifications, especially when inheritance and overridden methods are involved. In addition, the specifications based on abstract predicates cannot be automatically verified.

From the discussion, we conclude that both abstraction techniques has their own merit, thus none of them is superior to the other.

III. Expressiveness

Considering the importance of the abstraction techniques in OO verification, we are interested in comparing model fields and abstract predicates about their expressiveness. As an example, we have shown in Section II that *Rec* can be specified by both techniques respectively. Although it is just a special case, it gives us some clues for the study.

Firstly we notice that each model field holds a value, and a constraint binds it to some concrete fields (directly, or indirectly via some other model fields). Correspondingly, an abstract predicate is also able to describe a value by adding a parameter, and the definition of the predicate can describe any constraints between this value and the concrete fields. Thus, loosely speaking, it seems that model fields are probably special cases of abstract predicates. But we also need to consider their scope rules and their different ways to deal with frame properties.

Following this basic idea, we develop a set of rules and two translation algorithms, which can translate a program with model fields based specification into one with abstract predicates based specification. It indicates that the abstract predicates are not less expressive than model fields. We prove the correctness and termination of the translation algorithms, and we also show programs which can only be specified/verified by abstract predicates.

A. Translation Rules

In this subsection, we introduce our translation rules and an algorithm. We take a sequential subset of Java. The abstract predicates based specification is written using separation logic [17], and model fields based specification uses a subset of separation logic (without separating conjunction/implication). Before translation, the program is specified by model fields only. After translation, we will have a program specified by abstract predicates only. However, during the translation, the specification may contain both kinds of notations. In such cases, we define the semantics as follows:

Definition 6. *Suppose ψ is an assertion with both model fields and abstract predicates, and s is a state. ψ holds in s* iff *for each possible model field valuator M, the abstract predicate based assertion $\psi[M(s,o,f)/o.f]$ holds in s. Here $M(s,o,f)$ is the value of $o.f$ under the valuator M.*

We introduce some notations at first:

- $spec_1 \Longrightarrow_t spec_2$ expresses that we translate specification $spec_1$ into $spec_2$;
- B and C denote classes, φ and ψ assertions;
- $\mathsf{model}(C, \overline{f_i}), \mathsf{concrete}(C, \overline{g_j})$ state that $\overline{f_i}, \overline{g_j}$ are the model fields and concrete fields of class C, respectively. We assume the set $\{\overline{C.f_i}\}$ unchanged even when their definitions are translated into abstract predicates.
- $\mathsf{fresh}(\overline{r_j})$ means that $\overline{r_j}$ are fresh variables, and $\mathsf{inh}(C, f_i)$ denotes the model field f_i is also defined in the superclass of C.

The rules are listed in **Fig. 3**. Here are some explanations.

$$\text{[H-PPRE]}\ \frac{\mathsf{concrete}(C, \overline{g_j}),\quad \mathsf{fresh}(\overline{r_j})}{\Longrightarrow_t \mathbf{define}\ pri_C(\mathbf{this}, \overline{r_j}) :\ \bigwedge_j \mathbf{this}.g_j = r_j}$$

$$\text{[H-MPRE]}\ \frac{\mathsf{model}(C, \overline{f_i}),\ \mathsf{concrete}(C, \overline{g_j}),\ \mathsf{fresh}(v, \overline{r_j}),\ \neg\mathsf{inh}(C, f_k),\quad \mathbf{model}\ T\ f_k\ \mathbf{such\_that}\ R^k_C(f_k, \overline{f_i}, \overline{g_j}) \in C}{\begin{array}{l}\mathbf{model}\ T\ f_k\ \mathbf{such\_that}\ R^k_C(f_k, \overline{f_i}, \overline{g_j}) \Longrightarrow_t \\ \quad \mathbf{define}\ p^k_C(\mathbf{this}, v) :\ \exists \overline{r_j} \cdot pri_C(\mathbf{this}, \overline{r_j}) \wedge R^k_C(v, \overline{f_i}, \overline{r_j})\end{array}}$$

$$\text{[H-MINH]}\ \frac{\mathsf{model}(C, \overline{f_i}),\ \mathsf{concrete}(C, \overline{g_j}),\ \mathsf{fresh}(v, \overline{r_j}),\ \mathsf{inh}(C, f_k),\ \mathsf{super}(C, B),\quad \mathbf{model}\ T\ f_k\ \mathbf{such\_that}\ R^k_C(f_k, \overline{f_i}, \overline{g_j}) \in C}{\begin{array}{l}\mathbf{model}\ T\ f_k\ \mathbf{such\_that}\ R^k_C(f_k, \overline{f_i}, \overline{g_j}) \Longrightarrow_t \\ \quad \mathbf{define}\ p^k_C(\mathbf{this}, v) : \\ \quad\quad (\exists \overline{r_j} \cdot pri_C(\mathbf{this}, \overline{r_j}) \wedge R^k_C(v, \overline{f_i}, \overline{r_j})) \wedge p^k_B(\mathbf{this}, v)\end{array}}$$

$$\text{[H-MOLD]}\ \frac{\langle\varphi\rangle\ m(\ldots)\langle\psi\rangle,\quad \mathbf{old}(o.f_k)\ \text{appears in}\ \psi,\quad \mathsf{fresh}(d_k)}{\langle\varphi\rangle\langle\psi\rangle \Longrightarrow_t \langle\varphi \wedge p^k(o, d_k)\rangle\langle\psi[d_k/\mathbf{old}(o.f_k)]\rangle}$$

$$\text{[H-TRANS]}\ \frac{\psi\ \text{is not an invariant},\quad \mathsf{fresh}(\overline{v_i}, \overline{r_j}),\quad \overline{o_i.f_i}, \overline{o_j.g_j}\ \text{are all the model/concrete fields in}\ \psi}{\psi \Longrightarrow_t \exists \overline{v_i}, \overline{r_j} \cdot \psi\overline{[v_i/o_i.f_i][r_j/o_j.g_j]} \wedge \bigwedge_i p^i(o_i, v_i) \wedge \bigwedge_j pri(o_j, =, r_j, =)}$$

$$\text{[H-INV]}\ \frac{\psi\ \text{is an invariant in class}\ C,\quad \mathsf{model}(C, \overline{f_i}),\quad \mathsf{fresh}(\overline{v_i}),\quad B\ \text{ranges in the superclasses of}\ C\ \text{(inclusive)}}{\psi \Longrightarrow_t \exists \overline{v_i} \cdot \psi\overline{[v_i/f_i]} \wedge \bigwedge_k \bigwedge_B R^k_B(v_k, \overline{v_i}, \overline{g_j})}$$

Fig. 3. Translation Rules

Rule [H-PPRE] says that, we introduce a predicate $pri_C(\mathbf{this}, r_1, \ldots, r_n)$ for each class C, to assert all the concrete fields of the class. This predicate serves as a safeguard to prevent these fields from being exposed in the abstract specification.

Here pri is not a good predicate since its parameters give clues about the implementation details. However, our goal is only to show that the model fields can be translated into abstract predicates. We have a theorem below (**Theorem 2**) that says: if the model fields based specification is well-encapsulated, then we do not need [H-PPRE] any more.

[H-MPRE] shows how we translate a model field definition into an abstract predicate definition. The last premise indicates the existence of the definition for model field f_k in class C. We translate the model field definition into a predicate definition for p^k_C with the same number k. If $p^k_C(\mathbf{this}, v)$ holds, then the value of v equals a possible value of model field f_k.

In [H-MPRE], we do not handle those $\overline{f_i}$ in the constraint. Those model field appearances will be translated by the rule [H-TRANS] below.

Since subclasses are allowed to strengthen the constraints of the inherited model fields [8], [13], then the inherited model field should satisfy not only the constraint defined in C, but also the ones inherited from the superclasses of C. Rule [H-MINH] translates the inherited model fields, where $\mathsf{super}(C, B)$ says that B is the immediate superclass of C, and p^k_B is predicate in class B which p^k_C inherits. Clearly, p^k_C should also conform to the definition of p^k_B.

[H-MOLD] deals with the $\mathbf{old}(o.f_k)$ in method specifications, where $o.f_k$ is a model field. Here $\langle\varphi\rangle\ m(\ldots)\langle\psi\rangle$ means that φ and ψ are the pre and post-conditions of method m. In programs with model fields based specification, $\mathbf{old}(o.f_k)$ can be written in the post-condition, to denote the value of $o.f_k$ in the pre-state. Our rule declares d_k explicitly for the value of $o.f_k$ in the pre-state, then we can use d_k to substitute all the $\mathbf{old}(o.f_k)$ in the post-condition. Furthermore, the pre-condition after translation is equivalent to the original one. Since the constraints of model fields must be satisfiable, i.e. there surely exists a value d_k which equals to $o.f_k$ in the pre-state, [H-MOLD] does not strengthen the pre-condition.

[H-TRANS] translates assertions in the program except the invariants (i.e. the pre and post-conditions of methods and the definition of predicates). Here the predicate $p^i(o_i, v_i)$ asserts that v_i has the same value as model field $o_i.f_i$, then the substitution of all the $o_i.f_i$ to v_i does not modify the semantics of the assertions. The $pri(o_j, =, r_j, =)$ means that the parameter corresponding to $o_j.g_j$ is r_j, and we do not care the value of other parameters.

[H-INV] translates a model fields based invariant into a concrete fields based invariant. Without ownership [5] technique, the invariant can only mention the model and concrete fields defined in the same class. Thus we can just unfold the constraints and substitute the model field appearances to fresh variables within the scope of the existential quantifiers. We will consider the ownership technique in Section III-E.

Based on the rules in **Fig. 3**, we present an algorithm to translate a program with the model fields based specification to the same program with the corresponding abstract predicates based specification:

Algorithm 1. *We translate a program by 4 steps:*

- *For each class, use rule* [H-PPRE] *to encapsulate all the private fields;*
- *For each model field definition, use* [H-MPRE] *or* [H-MINH] *to translate it into a predicate definition;*
- *For each* $\mathbf{old}(\cdot)$ *appearance in assertions, use rule* [H-MOLD]*;*
- *Use* [H-INV] *to translate all the invariants, and use* [H-TRANS] *to translate all the other assertions.*

We can use this algorithm to translate the specification of all the classes and client code. In this and next subsection, we will prove several good properties of the algorithm.

Theorem 1. *For a program S with model fields based specification, using* **Algorithm 1***, we can obtain a program T which is specified based on abstract predicates. In addition, the specification of T is well-encapsulated.*

Proof. The second step will remove all the model field definitions, and the fourth step will remove all the model field appearances in assertions. Thus T has an abstract predicates based specification (and we will prove that it is a well-formed specification in the next subsection). Besides, we can see that all the concrete field appearances in the method specifications are substituted by predicate pri in the fourth step. Thus, the resulting specification is well-encapsulated. □

```
class Rec : Object{
  int x1, y1, x2, y2;
  invariant x1 ≤ x2 ∧ y1 ≤ y2;
  define width_Rec(this, v) :  v = x2 − x1;
  define height_Rec(this, v) :  v = y2 − y1;
  Rec() ⟨⟩
  ⟨∃v, v' · v = 1 ∧ v' = 1 ∧ width(this, v) ∧ height(this, v')⟩
  void ScaleH(int factor) ⟨(0 ≤ factor) ∧ width(this, d)⟩
  ⟨∃v · width(this, v) ∧ v = d × factor/100⟩
}
```

Fig. 4. *Rec* Translated into Abstract Predicates

Theorem 2. *If a program S with model fields based specification is well-encapsulated, we can modify the rules to simplify the translation algorithm:*

- *Delete the rule* [H-PPRE] *and delete the first step in* **Algorithm 1***;*
- *In other rules, unfold predicate $pri(\ldots)$ or $pri_C(\ldots)$ to its definition.*

If we obtain a program T from S using the simplified algorithm, then the specification of T will still be well-encapsulated.

Proof. We can prove it by induction. If the method specifications in S do not mention concrete fields, no matter which rule we use, the method specifications in T would not mention concrete fields. □

Example 1. The specification of *Rec* in **Fig. 1** is well-encapsulated. According to **Theorem 2**, we can translate it using the simplified algorithm. The result is shown in **Fig. 4**.

B. Well-Formedness of Translation Result

The frameworks based on either model fields or abstract predicates provide some scope rules about: 1) the locations in specifications where a model field or an abstract predicate can be mentioned; 2) the concrete fields that a model field or an abstract predicate can depend on. These rules can be found in the related work [10], [18].

Most of the scope rules are stated in Definition 1 and 4. Besides, model fields can be mentioned in the invariant of the same class, but abstract predicates cannot be mentioned in any invariant. Based on the scope rules, we define:

Definition 7 (Well-formed)**.** *If the model fields or abstract predicates based specification for a program does not violate the corresponding scope rules, we say that it is a* well-formed specification*. Programs with well-formed specifications are called* well-formed programs*.*

Theorem 3. *If a program S with model fields based specification is well-formed, we translate S using* **Algorithm 1** *and obtain T, then T is well-formed.*

Proof. We prove the two requirements of the scope respectively.

(1) An abstract predicate can be used at anywhere (syntactically), except the invariants. Due to [H-INV], the invariants in T do not mention any predicate. Thus T meets the first requirement.

(2) Since S is well-formed, the model fields in S only depend on the (model and concrete) fields in the same class. Thus the predicates in T only depend on the fields in the same class, which is allowed by Definition 4. Therefore T meets the second requirement.

From (1) and (2), we can conclude that T is well-formed. □

```
class SRec : Object{
  int x1, y1, x2, y2, scale;
  model int width such_that width = (x2 − x1) × scale;
  model int height such_that height = (y2 − y1) × scale;
  void ScaleH(int factor)
  ⟨0 ≤ factor⟩⟨width = old(width) × factor/100⟩
  modifies width, x2;
}
```

Fig. 5. Another Implementation of *Rec* with Model Fields Based Specification

C. Framing

Framing is an important problem in verification. It requires us to specify the boundary of memory locations modifiable/modified by the methods. For example, we must be able to prove that $ScaleH$ does not modify $height$ in **Fig. 1**; and $ScaleH$ does not modify the validity of $height(\mathbf{this}, v)$ in Example 1.

Several solutions are proposed to deal with the frame problem. Literature on model fields often uses a **modifies** clause to declare the fields possibly modified by the method, while the other fields will not be modified. Thus in **Fig. 1**, we are sure that $ScaleH$ does not modify $height$.

Specifications based on abstract predicates use separation logic to describe the frame properties. In Example 1, since predicates $width_{Rec}$ and $height_{Rec}$ are separated, we know that $ScaleH$ does not modify $height$.

Since the model fields and abstract predicates techniques use different strategies to deal with the frame problem, we need to present another algorithm to translate the **modifies** clauses into separation logic assertions.

The example in **Fig. 5** illustrates the main difficulty of the translation. Class $SRec$ includes a concrete field $scale$ as the scale factor of the rectangle, and both model fields $width$ and $height$ depend on this field.

Due to the **modifies** clause of $ScaleH$, we can still prove that it does not modify $height$. But after we translate this program using **Algorithm 1**, we cannot use frame rule to prove that $height_{SRec}$ is not modified by $ScaleH$, since the predicates $width_{SRec}$ and $height_{SRec}$ are not separated.

This example suggests us to merge the predicates which are not separated. Firstly we introduce some notations:

- In order to distinguish the new rules from those rules in **Fig. 3**, here we use $spec_1 \Longrightarrow_f spec_2$ to denote that $spec_1$ is translated into $spec_2$.
- α denotes a sequence of numbers, and $\alpha :: k$ is the sequence which appends k at the end of sequence α. $\alpha[i]$ denotes the i-th number in the sequence.

$$\text{[P-MRG]}\ \frac{\begin{array}{c}p_C^\alpha, p_C^k \text{ are not separated},\\ \textbf{define } p_C^\alpha(\textbf{this}, v_1, \ldots, v_n) : \psi_\alpha,\\ \textbf{define } p_C^k(\textbf{this}, v) : \psi_k\end{array}}{\begin{array}{c}\textbf{define } p_C^\alpha \ldots;\ \textbf{define } p_C^k \ldots \Longrightarrow_f\\ \textbf{define } p_C^{\alpha::k}(\textbf{this}, v_1, \ldots, v_n, v_{n+1}) : \psi_\alpha \wedge \psi_k[v_{n+1}/v]\end{array}}$$

$$\text{[P-SBST]}\ \frac{\begin{array}{c}\textbf{define } p_C^\alpha(\textbf{this}, v_1, \ldots, v_n) : -,\\ \mathsf{type}(o) = C, \quad k = \alpha[i], \quad p^k(o, v) \text{ appears in } \psi\end{array}}{\psi \Longrightarrow_f \psi[p^\alpha(o, =, v, =)/p^k(o, v)]}$$

$$\text{[P-MTHD]}\ \frac{\begin{array}{c}\langle\varphi\rangle\, C.m(\ldots)\langle\psi\rangle,\ m \text{ is not a constructor},\ \mathsf{fresh}(v),\\ \textbf{define } p_C^\alpha(\textbf{this}, v_1, \ldots, v_n) : -,\\ \mathsf{type}(o) = C,\ k = \alpha[i],\ p^\alpha(o, \ldots) \text{ appears in } \langle\varphi\rangle\langle\psi\rangle,\\ o.f_k \text{ does not appear in the } \textbf{modifies} \text{ clause of } m\end{array}}{\langle\varphi\rangle\langle\psi\rangle \Longrightarrow_f \langle\varphi \wedge p^\alpha(o, =, v, =)\rangle\langle\psi \wedge p^\alpha(o, =, v, =)\rangle}$$

$$\text{[P-MDF]}\ \frac{}{\textbf{modifies } \ldots \Longrightarrow_f}$$

Fig. 6. Translation Rules about **modifies** Clauses

We begin from a simple case without inheritance. All the new rules are shown in **Fig. 6**.

[P-MRG] merges two predicates into a "big" predicate. The superscript α records that the "big" predicate is merged from which predicates, the order of elements in α is the same as the order of parameters.

[P-SBST] uses the "big" predicates to substitute for the predicates in specifications. If p_C^α is merged from p_C^k and other predicates, then all the $p^k(o, v)$ in specifications can be translated to $p^\alpha(o, =, v, =)$, which means the first parameter is o and the $(i+1)$-th parameter is v, and we do not care about other parameters.

[P-MTHD] deals with method specifications. If a "big" predicate $p^\alpha(o, \ldots)$ appears in the pre or post-condition, but model field $o.f_k$ does not appear in the **modifies** clause ($k \in \alpha$), then we need to explicitly declare that the corresponding parameter of the "big" predicate is not modified.

The resulting specification of [P-MTHD] can usually be simplified, e.g.:

$$p^\alpha(o, a, -) \wedge p^\alpha(o, -, b)$$

can be reduced to

$$p^\alpha(o, a, b)$$

due to the definition of p^α. Since we mainly concern about the expressiveness but not the specification simplification, thus we omit the simplification step in the translation algorithm.

The last rule [P-MDF] simply deletes the **modifies** clause. Based on these rules, we define the translation algorithm as follows.

Algorithm 2. *For a result of* **Algorithm 1***, i.e. a program with abstract predicates based specification and* **modifies** *clauses, we translate the* **modifies** *clauses by 4 steps:*

- *For each class, use rule* [P-MRG] *to merge its predicates until all the "big" predicates separate from each other*[3]*;*
- *Use* [P-SBST] *to translate all the assertions, until there are only "big" predicates in assertions;*
- *For each method, for each "big" predicate $p^\alpha(o, \ldots)$ appears in the pre or post-condition, for each $k \in \alpha$, if $o.f_k$ does not appear in the* **modifies** *clause, use* [P-MTHD] *to declare that it is not modified;*
- *Use* [P-MDF] *to delete all the* **modifies** *clauses.*

[3]If we cannot judge whether two predicates are separated, we just merge them.

Theorem 4. *Suppose program S has model fields based specification and* **modifies** *clauses. We use* **Algorithm 1** *and* **2** *to translate S, and obtain a program T, then the reasoning ability of T about frame properties is not weaker than that of S.*

Proof. In program S, suppose a model field $o.f_k$ is not modified by a method $C.m$. We will prove the corresponding frame property in program T:

(1) If $k \in \alpha$ and the predicate $p^\alpha(o, \ldots)$ appears in the pre or post-condition of $C.m$, then [P-MTHD] would declare that the corresponding parameter of the "big" predicate remains the same in the pre and post-condition.

(2) If $k \in \alpha$ but the predicate $p^\alpha(o, \ldots)$ does not appear in the pre or post-condition of $C.m$, since $p^\alpha(o, \ldots)$ is separated from the other "big" predicates, we can use frame rule to prove that the parameters of p^α are not modified.

So we can conclude that all the frame properties provable in S are also provable in T. □

Example 2. After using **Algorithm 1** and **2** to translate the class *SRec* in **Fig. 5** (with some simplification), the result is:

$$\begin{array}{l}\textbf{define } \mathit{widthHeight}_{SRec}(\textbf{this}, v1, v2) :\\ \quad v1 = (y2 - y1) \times \mathit{scale} \wedge v2 = (y2 - y1) \times \mathit{scale};\\ \mathit{SRec}()\ \langle\rangle\langle \mathit{widthHeight}(\textbf{this}, 1, 1)\rangle\\ \textbf{void } \mathit{ScaleH}(\textbf{int } \mathit{factor})\\ \langle 0 \le \mathit{factor} \wedge \mathit{widthHeight}(\textbf{this}, d, v')\rangle\\ \langle \mathit{widthHeight}(\textbf{this}, d \times \mathit{factor}/100, v')\rangle\end{array}$$

Now we consider the inheritance. Since a subclass can add a new constraint to the inherited model fields, then even two predicates translated from model fields are separated in the superclass, they are possibly not separated in the subclass. To keep the behavioural subtype relation, we also need to merge them in the superclass. Thus, the requirement of [P-MRG] that

$$p_C^\alpha, p_C^k \text{ are not separated}$$

should be modified to

$$\exists D <: C \cdot p_D^\alpha, p_D^k \text{ are not separated.}$$

That is to say, if two predicates are not separated in one of C's subclasses, we need to merge them in C. This rule need to check all the subclasses of C, thus it is not modular. The other rules remain the same.

We prove the correctness and termination of both translation algorithms. The proof can be found in our technical report [**?**].

Theorem 5 (Correctness)**.** *If $S \Longrightarrow_t T$ or $S \Longrightarrow_f T$, then program S satisfies its specification* iff *program T satisfies its specification.*

D. Further Investigation

Previously, we have proved that the abstract predicates are not less expressive than model fields. Now we show two examples which can be specified by abstract predicates, but cannot be specified by model fields, therefore, abstract predicates are strictly more expressive than model fields.

When dealing with subtyping, abstract predicates based frameworks are more flexible, because they allow a predicate in subclass to have completely different definition from the predicate in the superclass. The behavioral subtype relation holds if the abstract specification of subclass refines the abstract specification of superclass, no matter how the predicates are overridden. On the other hand, an inherited model field is not allowed to have arbitrary constraint. Some papers [15], [4] do not allow any modification, and some others [8], [13] only allow the inherited model field to strengthen the constraint. Either is much more limited than the abstract predicates.

For example, D is a subclass of C, and method $D.m()$ $\{\mathbf{return}\ 2\times\mathbf{this}.x;\}$ overrides $C.m()$ $\{\mathbf{return}\ \mathbf{this}.x;\}$. It is impossible to accurately specify their return values using model fields, since the behavioural subtype relation will be violated in that case. However, both methods can be accurately specified using overridden predicates [18].

As another example, when dealing with recursion data structures, abstract predicates are more expressive than model fields. For the length of linked lists, we can define a recursive abstract predicate:

$$\begin{aligned}\mathbf{define}\ len_{Node}(\mathbf{this}, v):\ &(\mathbf{this}.next = \mathbf{null} \Rightarrow v = 1)\\ &\wedge(\mathbf{this}.next \neq \mathbf{null} \Rightarrow len(\mathbf{this}.next, v-1));\end{aligned}$$

However, using model fields (without ownership mechanism), we cannot define the length of linked lists. Since the constraints of model fields are invariant-like properties (they must hold in all program states), the model fields cannot depend on the fields of other objects. Thus, methods like "append a new node to a linked list" cannot be specified by model fields.

Even in the presence of the ownership mechanism, model fields have more restrictions on recursion than abstract predicates. In a typical setting [13], the constraint of model field $o.f$ could only mention the fields of o, and the fields of the representation fields of o. Thus, to allow the model field len depending on $next.len$, each node must own the next node. But this design will prevent us from traversal the list by loop[4].

Now we can conclude that, abstract predicates are strictly more expressive than model fields.

E. Ownership

Since a model field can only depend on the fields of the same object, it can hardly describe the properties of aggregate objects. To solve this problem, [15], [8], [13] introduce the ownership [5] mechanism into model field approaches. With this mechanism, a model field declaration in class C takes the form:

$$\mathbf{model}\ T\ f\ \mathbf{such\_that}\ R_C(f, \overline{o.f}, \overline{o.g});$$

Here o ranges in all the objects which are (transitive) owned by $\mathbf{this}$ object, including $\mathbf{this}$ itself.

On the other hand, since the predicate can already mention other objects in its definition, [16] states that abstract predicates do not need the ownership mechanism.

When we take the ownership mechanism into account, the specification based on model fields, except the class invariants, can still be translated. However, the translation rules need to be revised for a little bit, since the constraints of model fields can depend on other objects now. Take [H-MPRE] as an example, we modify the premise "$\overline{f_i}$ are model fields of $\mathbf{this}$" to "$\overline{o_i.f_i}$ are model fields of $\mathbf{this}$ and the objects $\mathbf{this}$ transitive owns", and replace "$\overline{f_i}$" with "$\overline{o_i.f_i}$" in the rule. Other rules can be revised similarly.

But the invariants cannot be directly translated, since abstract predicates based specification does not allow multi-object invariants. To solve the problem, we translate the invariants into method specifications, as suggested by [16].

We define a predicate for each class:

$$\text{[H-INVD]}\ \frac{\overline{\mathbf{invariant}\ \psi_i};\ \text{are all the invariants of class } C}{\overline{\mathbf{invariant}\ \psi_i} \Longrightarrow_t \mathbf{define}\ inv_C(\mathbf{this}):\ \bigwedge_i \psi_i;}$$

And its definition needs to be translated by [H-TRANS]. Then, suppose the predicate $inv(o)$ mentions some other objects $\overline{o_i}$ owned by o, we assert that $inv(o)$ holds on the entry and exit of every public method of $\overline{o_i}$:

$$\text{[H-INVM]}\ \frac{\begin{array}{l}o : C,\ \overline{o_i}\ \text{are all the objects owned by } o\text{, including } o \text{ itself}\end{array}}{\begin{array}{l}\langle\varphi\rangle\ C.C(\ldots)\langle\psi\rangle \Longrightarrow_t \langle\varphi\rangle\ C.C(\ldots)\langle\psi \wedge inv(\mathsf{ret})\rangle;\\ \langle\varphi\rangle\ o_i.m(\ldots)\langle\psi\rangle \Longrightarrow_t \langle\varphi \wedge inv(o)\rangle\ o_i.m(\ldots)\langle\psi \wedge inv(o)\rangle\end{array}}$$

Here ret denotes the return value. When reasoning about the aggregate object (o together with $\overline{o_i}$), the multi-object invariant can be proved using [H-INVM]; when reasoning about the other objects, the multi-object invariant can be proved using frame rule.

Notice that if the behavioural subtype relation holds in the original specification, then it holds in the translating result, since we just add a same predicate to all the pre and post-conditions of non-construction methods.

In conclusion, considering the ownership mechanism, the model fields based specification can still be translated into abstract predicates based one. However, the invariants need to be translated into method specifications.

IV. Related Work

The concept of *model fields* was proposed by Leino in his PhD thesis [11], where it was called *abstract fields*. In the thesis, these fields are used to specify/verify data refinement. However, the work almost ignores information hiding, and requires the abstract fields to be uniquely determined by the constraint.

[4]In literature on ownership, the nodes of a list do not own each other, instead, they have a same owner. If we define that each node owns the next node in Spec#, we cannot traverse a list by loop since the owner of a variable must keep unchanged in a loop.

Müller *et al.* [15] presented a modular technique for specifying/verifying frame properties, using model fields and modifies clauses. It also uses the ownership technique to deal with aggregate objects. However, the work does not fully support the inheritance.

Leino and Müller presented a modular verification methodology for model fields [13]. In this work, model fields of an object are updated only when the object is in valid states. The approach is tightly combined with the ownership technique, thus it is able to modularly deal with the frame properties of aggregate objects, and fully supports inheritance. However, as we discussed in Section III-D, the expressiveness of the work has some limitations.

There are some applications of model fields. For instance, they are used to express the abstract specifications of classes and interfaces in JML [8], to write and check *design by contract* assertions [4], and to achieve information hiding in the specifications for interfaces [9].

The concept of *abstract predicates* was proposed by Parkinson and Bierman [17] in developing an OO program verification system based on separation logic. Lately they take inheritance into account [18], and introduce static/dynamic specifications for each method, to avoid restricting subtypes' behaviors or re-verifying inherited methods.

Liu *et al.* proposed an OO verification framework in [14], whose abstraction technique is similar to Parkinson's. It can deal with the inheritance by using simpler notations, where each method only need one specification and re-verification is avoided as well. Qiu *et al.*extend the framework for interfaces [19], and give a solution to the problems brought by multiple inheritance.

There are some other abstraction techniques for OO verification, e.g., pure methods [6] and data groups [12]. The *pure methods* are side-effect free methods, thus they can be used in specifications to hide implementation details. *Data groups* are mainly for framing and information hiding. Each data group represents a set of variables, and it can be used in the modifies clauses to announce the modifiability of its members.

Although many papers focus on the model fields or abstract predicates based techniques, a systematic comparison between the two techniques is lacking. Burgman [3] proposed a comparison between JML and Separation Logic in verifying multi-threaded Java programs. But the work does not focus on the abstraction mechanisms.

V. Conclusion

In this work, we made a deep investigation on two important abstraction techniques for specifying/verifying OO programs, the *model fields* and the *abstract predicates*.

By developing some translation rules and algorithms, we demonstrate that any program with model fields based specification can be translated into one with abstract predicates based specification. The translation is still correct in the presence of inheritance, recursion, or even the ownership technique.

We prove that the resulting specification obtained from the translation is well-encapsulated, well-formed and correct. A program satisfies its model fields based specification iff it satisfies the resulting abstract predicates based specification. In addition, we demonstrate the existence of programs which can be specified by abstract predicates but not model fields, which shows that the abstract predicates technique is more expressive.

Although abstract predicates are more expressive, model fields have its own advantages like user friendliness and supporting automatic verification. Our conclusion is, the seeking for the powerful and easy-to-use abstraction techniques for OO specification and verification has not come to the end. As a future work, we plan to go further to investigate other abstraction techniques and compare them, and try to develop better specification techniques or specification patterns for OO programs.

References

[1] M. Barnett, K. R. M. Leino, and W. Schulte. The Spec# programming system: An overview. In *LNCS 3362*, pages 49–69. Springer, 2005.

[2] D. Bruns. Formal semantics for the Java Modeling Language. Diplomarbeit, Universität Karlsruhe, June 2009.

[3] R. Burgman. Specifying multi-threaded Java programs. Available at http://referaat.cs.utwente.nl/conference/12/paper, 2010.

[4] Y. Cheon, G. Leavens, M. Sitaraman, and S. Edwards. Model variables: Cleanly supporting abstraction in design by contract: Research articles. *Software: Practice and Experience*, 35(6):583–599, May 2005.

[5] D. G. Clarke, J. M. Potter, and J. Noble. Ownership types for flexible alias protection. In *OOPSLA'98*, pages 48–64. ACM, 1998.

[6] Á. Darvas and K. R. M. Leino. Practical reasoning about invocations and implementations of pure methods. In *FASE'07*, pages 336–351. Springer, 2007.

[7] B. Jacobs and F. Piessens. The VeriFast program verifier. *CW Reports*, 2008.

[8] G. T. Leavens, A. L. Baker, and C. Ruby. Preliminary design of JML: A behavioral interface specification language for Java. *SIGSOFT Software Engineering Notes*, 31(3):1–38, May 2006.

[9] G. T. Leavens and P. Müller. Information hiding and visibility in interface specifications. In *ICSE'07*, pages 385–395. IEEE CS, 2007.

[10] G. T. Leavens, E. Poll, C. Clifton, Y. Cheon, C. Ruby, D. Cok, P. Müller, J. Kiniry, and P. Chalin. *JML Reference Manual*, 2008.

[11] K. R. Leino. *Toward Reliable Modular Programs*. PhD thesis, California Institute of Technology, 1995.

[12] K. R. M. Leino. Data groups: Specifying the modification of extended state. In *OOPSLA'98*, pages 144–153. ACM, 1998.

[13] K. R. M. Leino and P. Müller. A verification methodology for model fields. In P. Sestoft, editor, *LNCS 3924*, pages 115–130. Springer, 2006.

[14] Y. Liu, A. Hong, and Z. Qiu. Inheritance and modularity in specification and verification of OO programs. In *TASE'11*, pages 19–26. IEEE CS, 2011.

[15] P. Müller, A. Poetzsch-Heffter, and G. T. Leavens. Modular specification of frame properties in JML. *Concurrency and Computation: Practice and Experience*, 15(2):117–154, 2003.

[16] M. Parkinson. Class invariants: The end of the road. In *International Workshop on Aliasing, Confinement and Ownership*, volume 23, 2007.

[17] M. Parkinson and G. Bierman. Separation logic and abstraction. In *POPL'05*, pages 247–258. ACM, 2005.

[18] M. J. Parkinson and G. M. Bierman. Separation logic, abstraction and inheritance. In *POPL'08*, pages 75–86. ACM, 2008.

[19] Z. Qiu, A. Hong, and Y. Liu. Modular verification of OO programs with interfaces. In *LNCS 7635*, pages 151–166. Springer, 2012.

[20] J. C. Reynolds. Separation logic: A logic for shared mutable data structures. In *LICS'02*, pages 55–74. IEEE CS, 2002.

APPENDIX

We include the proof of correctness and termination here as an easy access for the reviewers. These materials will be removed later and left in our technical report [?].

Theorem 6. *Given the definition of the predicates introduced by other rules,* [H-TRANS] *does not change the semantics of assertions. More precisely, for every state s and model field valuator M,*

$$\begin{aligned}&\overline{o_i : C_i}, \overline{o_j : C_j}, \Delta \vdash (\forall M \cdot \psi[\overline{M(s, o_i, f_i)/o_i.f_i}]) \Leftrightarrow \\ &\quad (\exists \overline{v_i}, \overline{r_j} \cdot \psi[\overline{v_i/o_i.f_i}][r_j/o_j.g_j] \wedge \textstyle\bigwedge_i p^i(o_i, v_i) \\ &\quad\quad \wedge \textstyle\bigwedge_j private(o_j, =, r_j, =)).\end{aligned}$$

Here Δ is the assumption context introduced in Section II-B, *$\overline{v_i}$ and $\overline{r_j}$ are fresh variables.*

Proof. Suppose B_i ranges in the superclasses of C_i (inclusive). According to Definition 2, we have

$$\begin{aligned}&\overline{o_i : C_i}, \overline{o_j : C_j}, \Delta \vdash \\ &(\forall M \cdot \psi[\overline{M(s, o_i, f_i)/o_i.f_i}]) \\ \Leftrightarrow\ &(\forall M \cdot (\psi[\overline{M(s, o_i, f_i)/o_i.f_i}] \\ &\quad \wedge \textstyle\bigwedge_i \bigwedge_{B_i} R^i_{B_i}(M(s, o_i, f_i), \overline{M(s, o_i, f_i)}, o_i.\overline{g_{ij}}))) \\ \Leftrightarrow\ &(\forall M \cdot (\exists \overline{v_i} \cdot \psi[\overline{v_i/o_i.f_i}] \wedge \textstyle\bigwedge_i \bigwedge_{B_i} R^i_{B_i}(v_i, o_i.\overline{f_i}, o_i.\overline{g_{ij}}) \\ &\quad \wedge \textstyle\bigwedge_i v_i = M(s, o_i, f_i))) \\ \Leftrightarrow\ &(\exists \overline{v_i} \cdot \psi[\overline{v_i/o_i.f_i}] \wedge \textstyle\bigwedge_i \bigwedge_{B_i} R^i_{B_i}(v_i, o_i.\overline{f_i}, o_i.\overline{g_{ij}})) \\ \Leftrightarrow\ &(\exists \overline{v_i} \cdot \psi[\overline{v_i/o_i.f_i}] \wedge \textstyle\bigwedge_i \bigwedge_{B_i} \exists \overline{r_{ij}} \cdot (R^i_{B_i}(v_i, o_i.\overline{f_i}, \overline{r_{ij}}) \\ &\quad \wedge \textstyle\bigwedge_j o_i.g_{ij} = r_{ij})) \\ \Leftrightarrow\ &(\exists \overline{v_i} \cdot \psi[\overline{v_i/o_i.f_i}] \wedge \textstyle\bigwedge_i \bigwedge_{B_i} \exists \overline{r_{ij}} \cdot (R^i_{B_i}(v_i, o_i.\overline{f_i}, \overline{r_{ij}}) \\ &\quad \wedge private_{B_i}(o_i, \overline{r_{ij}}))) \\ \Leftrightarrow\ &(\exists \overline{v_i} \cdot \psi[\overline{v_i/o_i.f_i}] \wedge \textstyle\bigwedge_i p^i_{C_i}(o_i, v_i)) \\ \Leftrightarrow\ &(\exists \overline{v_i}, \overline{r_j} \cdot \psi[\overline{v_i/o_i.f_i}] \wedge \textstyle\bigwedge_i p^i_{C_i}(o_i, v_i) \wedge \textstyle\bigwedge_j o_j.g_j = r_j) \\ \Leftrightarrow\ &(\exists \overline{v_i}, \overline{r_j} \cdot \psi[\overline{v_i/o_i.f_i}][r_j/o_j.g_j] \wedge \textstyle\bigwedge_i p^i_{C_i}(o_i, v_i) \\ &\quad \wedge \textstyle\bigwedge_j o_j.g_j = r_j) \\ \Leftrightarrow\ &(\exists \overline{v_i}, \overline{r_j} \cdot \psi[\overline{v_i/o_i.f_i}][r_j/o_j.g_j] \wedge \textstyle\bigwedge_i p^i_{C_i}(o_i, v_i) \\ &\quad \wedge \textstyle\bigwedge_j private_{C_j}(o_j, =, r_j, =)) \\ \Leftrightarrow\ &(\exists \overline{v_i}, \overline{r_j} \cdot \psi[\overline{v_i/o_i.f_i}][r_j/o_j.g_j] \wedge \textstyle\bigwedge_i p^i(o_i, v_i) \\ &\quad \wedge \textstyle\bigwedge_j private(o_j, =, r_j, =)).\end{aligned}$$

□

We can prove that the other rules does not change the semantics similarly:

Theorem 7. *Given the definition of the predicates introduced by other rules,* [H-MOLD] *and* [H-INV] *does not change the semantics of assertions.*

Proof. Suppose B ranges in the superclasses of C (inclusive). For [H-INV]:

$$\begin{aligned}&(\forall M \cdot \psi[\overline{M(s, \mathbf{this}, f_i)/f_i}]) \\ \Leftrightarrow\ &(\forall M \cdot (\psi[\overline{M(s, \mathbf{this}, f_i)/f_i}] \\ &\quad \wedge \textstyle\bigwedge_k \bigwedge_B R^i_B(M(s, \mathbf{this}, f_k), \overline{M(s, \mathbf{this}, f_i)}, \overline{g_j}))) \\ \Leftrightarrow\ &(\forall M \cdot (\exists \overline{v_i} \cdot \psi[\overline{v_i/f_i}] \wedge \textstyle\bigwedge_k \bigwedge_B R^k_B(v_k, \overline{v_i}, \overline{g_j}) \\ &\quad \wedge \textstyle\bigwedge_i v_i = M(s, \mathbf{this}, f_i))) \\ \Leftrightarrow\ &(\exists \overline{v_i} \cdot \psi[\overline{v_i/f_i}] \wedge \textstyle\bigwedge_k \bigwedge_B R^k_B(v_k, \overline{v_i}, \overline{g_j})).\end{aligned}$$

For [H-MOLD]:

$$\frac{\dfrac{\langle\varphi\rangle\ -\langle\psi\rangle}{\langle\varphi \wedge p^k(o, d_k)\rangle\ -\langle\psi\rangle}\ \text{constraints are satisfiable}}{\dfrac{\langle\varphi \wedge p^k(o, d_k)\rangle\ -\langle\psi \wedge d_k = \mathbf{old}(o.f_k)\rangle}{\langle\varphi \wedge p^k(o, d_k)\rangle\ -\langle\psi[d_k/\mathbf{old}(o.f_k)]\rangle}\ \text{semantics of } \mathbf{old}}$$

and vice versa. □

Now we can conclude that:

Theorem 8 (Correctness of $\Longrightarrow_t$)**.** *Suppose $S \Longrightarrow_t T$, then program S satisfies its specification* iff *program T satisfies its specification.*

Proof. Only [H-TRANS], [H-INV] and [H-MOLD] are involved in specification translation. According to **Theorem 6** and **7**, given the definitions of predicates, all these rules do not change the semantics of specification. □

For the translation rules in Section III-C, we have:

Theorem 9 (Correctness of $\Longrightarrow_f$)**.** *Suppose $S \Longrightarrow_f T$, then program S satisfies its specification* iff *program T satisfies its specification.*

Proof. Only [P-SBST] and [P-MTHD] are involved in specification translation. Suppose the assumption context Δ contains all the formulas about predicates p^k and p^α. Then for [P-SBST]:

$$\begin{aligned}&o : C, \Delta \vdash \\ &\psi \\ \Leftrightarrow\ &(\psi \wedge p^k_C(o, v) = p^k_C(o, v)) \\ \Leftrightarrow\ &(\psi \wedge p^k_C(o, v) = p^\alpha_C(o, =, v, =)) \\ \Leftrightarrow\ &(\psi[p^\alpha_C(o, =, v, =)/p^k_C(o, v)] \wedge p^k_C(o, v) = p^\alpha_C(o, =, v, =)) \\ \Leftrightarrow\ &(\psi[p^\alpha_C(o, =, v, =)/p^k_C(o, v)]) \\ \Leftrightarrow\ &(\psi[p^\alpha(o, =, v, =)/p^k(o, v)]).\end{aligned}$$

For [P-MTHD]:

$$\frac{\dfrac{\langle\varphi\rangle\ -\langle\psi\rangle}{\langle\varphi \wedge p^\alpha_C(o, =, v, =)\rangle\ -\langle\psi\rangle}\ \text{constraints are satisfiable}}{\langle\varphi \wedge p^\alpha(o, =, v, =)\rangle\ -\langle\psi \wedge p^\alpha(o, =, v, =)\rangle}\ \text{the } \mathbf{modifies} \text{ clause}$$

and vice versa. Thus we can conclude that, the original program satisfies its specification, iff the resulting program satisfies its specification. □

Theorem 8 and **9** show the correctness of the translation algorithms. Besides that, the termination of **Algorithm 1** is obvious. Suppose there are n classes, and m model field annotations totally. Rule [H-PPRE] will execute no more than n times, and all the other rules will decrease the number of model field annotations in the program, therefore, the time complexity of **Algorithm 1** is $O(n + m)$.

For **Algorithm 2**, if there are n methods, and m annotations of predicates totally. Rule [P-MTHD] will execute no more than mn times, and other rules will execute no more than n times. Thus the time complexity of **Algorithm 2** is $O(mn)$.