Comparison between Model Fields and Abstract Predicates

Zhang Ke and Qiu Zongyan

LMAM & Depart. of Informatics, School of Math., Peking University, Beijing, China
{zksms@pku.edu.cn,qzy@math.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 of the classes. *Model fields* and *abstract predicates* are two most important approaches to address the requirements. In this paper, we compare these two approaches 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 algorithms are correct, and the resulting specifications are well-encapsulated and well-formed. This shows that the abstract predicates technique is more expressive in some sense. On the other hand, the model fields based specifications are easier to use and more user friendly. In addition, we discuss the different characteristics of the two approaches in framing, inheritance, and recursion.

1 Introduction

Object Orientation (abbr. OO) is widely used in the practices of 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 specific 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 OO area for years [1,2]. *Model fields* [3,4] and *abstract predicates* [5,6] are two important techniques proposed for abstraction in the OO verification. Model fields are abstract fields whose values are determined by concrete fields of the object. Abstract predicates are user-defined logic abstractions, whose definition is determined by the type of parameters. Both techniques allow us to specify the program more abstractly, thus provide better information hiding.

Model fields and abstract predicates have been proposed years ago, and both techniques address the similar problems in OO verification. However, no systematic and formal study has been published on their relationship and advantages/disadvantages in comparison of them with each other. We try to make such a comparison in this paper. We prove that the abstract predicates technique is more expressive than model fields, by giving a set of rules and algorithms which can translate a program with model fields based specification into one with abstract predicates based specification. We prove that the translation is correct, i.e. a program satisfies its model fields based specification, if and only if it satisfies the abstract predicates based specification obtained by our algorithm. On the other hand, although model fields are less expressive, they are more intuitive and user friendly.

The rest of the paper is organized as follows: Section 2 introduces the main idea of model fields and abstract predicates techniques, and also a preliminary comparison between the two techniques; Section 3 presents the translation rules and algorithms, together with some discussion and the correctness proof of the rules; Section 4 summarizes some related work and Section 5 concludes.

2 Background

In this section, we discuss the necessity and advantages of abstraction mechanisms in the specification/verification of OO programs, and introduce the main ideas of *model fields* and *abstract predicates* specification techniques briefly.

2.1 Abstraction Techniques in OO Specification and Verification

Object Orientation is important in programming and software construction, mainly because it provides a set of useful abstraction features. The *classes* provide the basic data abstraction for encapsulating related data and operations into a concrete module; the *inheritance* supports the incremental construction of related data abstractions; and the *dynamic binding* allows customized operation details in common behaviour patterns while keeping the abstraction boundary. These features provided by OO languages build up a flexible framework which has been proved extremely powerful in dealing with various construction problems of complex software systems. However, all these abstraction features bring challenges to the formal specification and verification.

For a specification/verification framework targeting at OO programs, its basic abilities should keep pace with the features and characteristics of OO programs. Ideally, the specifications should keep the abstraction boundary of the programs, and support inheritance at the same time, as well as provide a static and modular verification procedure even in the presence of dynamic binding.

Specifying OO programs based on some suitable abstraction utilities, but not directly on the concrete code, would be helpful for addressing these challenges. This kind of *abstract specifications* may provide many benefits, e.g.: 1) It would support the information hiding. For example, the method specifications visible to client code could be expressed in an implementation-independent way, thus the encapsulation boundary is protected. In addition, it allows the programmers to modify the real implementation but keep the abstract specification unchanged. 2) The specification allows developers to reason modularly on some abstract level. 3) The specification may be more friendly to the specification re-use, since the subclasses may share the abstract specification in the superclass, even when they have different implementations. 4) The abstraction would also improve the expressiveness of the specification. For example, since we can define the abstract specification of an interface without implementation, we could possibly prove the behavioral subtype relation for some more cases.

Due to these advantages, most OO verification frameworks introduce some abstraction mechanisms for specifying/verifying OO programs. *Model fields* and *abstract predicates* are two widely discussed abstraction techniques in this area. We give some brief introduction to them in the following subsections.

2.2 Model Fields

The *model fields* (or *abstract fields*) technique was introduced in 1990s [3] to deal with data abstraction and data refinement. Syntactically, model fields are a kind of fields in class declarations of the OO programs with specifications. Unlike concrete fields, model fields can only be used in the specifications, but cannot be accessed by real code.

We take JML [4] 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, and it can define 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 [7] 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 $M : \mathcal{S} \times \mathcal{U} \times \mathbb{I} \rightarrow \mathcal{U}$ is called a* model field valuator *iff:*

For every state $s \in \mathcal{S}$ and every model field $o.f$ with constraint ψ:[2]

– *The type of object $M(s, o, x)$ is a subtype of field x's type,*

[1] Here we omit some details about 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.

[2] ψ denotes the conjunction of all the constraints which $o.f$ must conform to.

– $\psi\overline{[M(s,o,f)/o.f]} = \mathbf{true}$ *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*[3]*.*

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 $\langle\phi\overline{[M(s,o,f)/o.f]}\rangle\langle\psi\overline{[M(s,o,f)/o.f]}\rangle$ 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 (of single object) 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 [8], uses concrete fields in its invariant. Thus we allow the invariants to mention concrete fields in Definition 3.

The specification visible to client code should be well-encapsulated, to get the benefits of data abstraction. Since the model field constraints are invisible outside the class, the concrete implementations are hidden. In addition, abstraction allows programmers to keep the specification unchanged when they modify the implementation. In this case, some modifications of the constraints would be enough, and the verification of the client code is still valid.

Fig. 1 shows an example to illustrate the idea and form of model fields, which comes from [8]. Here a class *Rec* uses the coordinates of two opposite corners to store information of a rectangle, and $x1, y1, x2, y2$ are the concrete fields. Two model fields *width* and *height* are introduced as an abstraction of the concrete fields, where the formulas in the **such_that** clauses express how the model fields depend on the concrete fields (and possibly other model fields).

Two methods are defined in the class, one is the constructor, and the other *ScaleH* scales the rectangle horizontally by a given percentage parameter. The post-condition describes this property by using model field *width*. Here we use the form $\langle\cdot\rangle\langle\cdot\rangle$ to denote the pre and post-conditions of methods. In addition, the **modifies** clause lists all the fields modified by the method, including concrete and model fields. This is useful when reasoning about frame properties of the method. Here method *ScaleH* modifies $x2$ and *width*.

[3] 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 $x1 \leq x2 \wedge y1 \leq y2$;
   model int $width$ such_that $width = x2 - x1$;
   model int $height$ such_that $height = y2 - y1$;
   $Rec()$ $\langle\rangle\langle width = 1 \wedge height = 1\rangle$
   $\{x1 := 0; y1 := 0; x2 := 1; y2 := 1; \}$
   void $ScaleH$(int $factor$)
   $\langle 0 \leq factor\rangle\langle width = \mathbf{old}(width) \times factor/100\rangle$
   modifies $width, x2$;
   $\{\ x2 := (x2 - x1) \times factor/100 + x1;\ \}$
}
```

Fig. 1. *Rec* Written Using Model Fields

2.3 Abstract Predicates

Abstract predicates are proposed when researchers apply Separation Logic [9] into OO area [5]. Basically, they are user-defined predicates in class declarations.

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

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

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.

The predicates with same name (but in different classes) form an *abstract predicate family*. Paper [10] defines its semantics as follows. Briefly speaking, when a predicate family 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 context Δ, which is a set including 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) $\dfrac{\mathbf{define}\ p_C(\mathbf{this}, \overline{v}) : \psi;}{p_C(o, \overline{v}) \Leftrightarrow \psi[o/\mathbf{this}]}$

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

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

```
class $Rec$ : Object{
    int $x1, y1, x2, y2$;
    invariant $x1 \leq x2 \wedge y1 \leq y2$;
    define $rec_{Rec}(\mathbf{this}, w, h)$ :  $w = x2 - x1 \wedge h = y2 - y1$;
    $Rec()$
    $\langle\rangle\langle rec(\mathbf{this}, 1, 1)\rangle$
    $\{x1 := 0; y1 := 0; x2 := 1; y2 := 1;\}$
    void $ScaleH$(int $factor$)
    $\langle(0 \leq factor) \wedge rec(\mathbf{this}, w, h)\rangle\langle rec(\mathbf{this}, w \times factor/100, h)\rangle$
    $\{\ x2 := (x2 - x1) \times factor/100 + x1;\ \}$
}
```

Fig. 2. *Rec* Written Using Abstract Predicates

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 also use the abstract predicates, with the help of separation logic, to write abstract specifications, as shown in **Fig. 2**. The **define** clause defines an abstract predicate entry of the class. When verifying the body of the methods of the class, e.g., *ScaleH*, we can unfold the predicate *rec* in the reasoning using the predicate definition. On the other hand, when verifying the client code which invokes *ScaleH*, we should treat $rec(\mathbf{this}, w, h)$ in the method specification as an atomic assertion and reason in an abstract level.

2.4 Preliminary Comparison

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

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 mechanism. The model fields of a class form an abstract boundary for hiding the concrete fields. In the literature on model fields, the notations for the constraints are simply formulas built from (concrete or model) fields and operators of the programming language. The simple notations limit the expressiveness of the model fields.

However, this design reflects the basic idea of the researchers, because they want a simple and easy-to-use abstract notation, without introducing complex syntax into the language. The meaning of model fields is simply a kind of dependent values. These features make the model fields rather friendly to the developers. In addition, because the values of model fields are constrained to concrete fields, in the verification or static/dynamic analysis, their values can be really implemented and automatically updated. This makes the model fields useful in the automatic verification or analysis. As another evidence for its usefulness, several OO specification languages, like JML [4] and Spec# [11], has integrated model fields into their specification languages for the users 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. In addition, theoretical works do not impose any limitation on the definition, thus useful mathematical concepts (e.g. sets, sequences) are widely used in predicate definitions in the literature.

In addition, recursively defined predicates can be naturally constructed, which makes it easy to handle recursive properties of programs, especially for 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 [12], 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.

3 Expressiveness

Considering the importance of the abstraction techniques in OO verification, we are interested in comparing model fields and abstract predicates about their expressiveness. We want to know whether the two techniques have equivalent specification power, in other words, is there any program which can only be specified using one technique, but cannot be specified by the other. As an example, we have shown in Section 2 that *Rec* can be specified by both model fields and abstract predicates 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 which 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 another 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 a particular case of abstract predicates. But we also need to consider their scope rules (e.g. predicates cannot be mentioned in invariants), and their different ways to deal with frame properties.

Following this basic idea, we have developed 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. This indicates that the abstract predicates are not less expressive than model fields. We give the rules and algorithms here, and then prove that the resulting specifications are well-formed and well-encapsulated. We also show a program which can be specified/verified by abstract predicates, but cannot be specified by model fields. At last, we prove the correctness and termination of the translation algorithms.

3.1 Translation Rules

In this subsection, we introduce our translation rules and algorithm. The translation takes two steps: to translate the model field definitions into corresponding abstract predicate definitions, and to translate the specification written based on the model fields into the corresponding specification based on those abstract predicates.

We take a sequential subset of Java. The abstract predicates based specification is written using separation logic [5], and model fields based specification uses a subset of separation logic (without separating conjunction/implication).

Before translation, the program is only specified by model fields. After translation, we will have a program only specified by abstract predicates. 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.*

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

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

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

$$\textsf{[H-IPRE]}\ \frac{\textbf{model}\ T\ f_k \in I,\quad \mathsf{fresh}(v)}{\textbf{model}\ T\ f_k \Longrightarrow_t \textbf{define}\ p^k_I(\textbf{this},v)}$$

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

$$\textsf{[H-TRANS]}\ \frac{\begin{array}{c}\psi\ \text{is not an invariant},\quad \mathsf{fresh}(\overline{v_i}),\quad \mathsf{fresh}(\overline{r_j}),\\ \overline{o_i.f_i},\overline{o_j.g_j}\ \text{are all the model field and concrete field appearances in}\ \psi\end{array}}{\psi \Longrightarrow_t \exists\overline{v_i},\overline{r_j}\cdot \psi\overline{[v_i/o_i.f_i][r_j/o_j.g_j]} \wedge \bigwedge_i p^i(o_i,v_i) \wedge \bigwedge_j private(o_j,\overline{\_},r_j,\overline{\_})}$$

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

Fig. 3. Translation Rules

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, I an interface, 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 $\mathsf{inh}(C,f_k)$ denotes the model field f_k is also defined in the superclass of C.

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

Rule [H-PPRE] says that, we introduce a predicate $private_C(\textbf{this},r_1,\ldots,r_n)$ 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. Even if the model fields based specification is not well-encapsulated, i.e. it mentions concrete fields, we can still guarantee that after the translation, the resulting specification is well-encapsulated.

The predicate introduced by [H-PPRE] is not good since its parameters give clues about the implementation details. However, our goal in the moment 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 rule [H-PPRE] any more.

Rule [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_C^k with the same number k. If $p_C^k(\mathbf{this}, v)$ holds, then the value of v equals a possible value of model field f_k.

In rule [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 [4,8], 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_B^k is predicate in class B which p_C^k inherits. Clearly, p_C^k should also conform to the definition of p_B^k.

In literature [4], a model field in an interface does not have a constraint, since there is no concrete field, nor implementation in interfaces. For such a model field, rule [H-IPRE] translates it into an abstract predicate without a definition. When verifying the interfaces, we can only reason at the level of predicates (without unfolding them).

[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, our translation 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 $private(o_j, \bar{\_}, r_j, \bar{\_})$ 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 [13] 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

```
class $Rec$ : **Object**{
  **int** $x1, y1, x2, y2$;
  **invariant** $x1 \leq x2 \wedge y1 \leq y2$;
  **define** $private_{Rec}(\mathbf{this}, r1, r2, r3, r4)$ :  $x1 = r1 \wedge y1 = r2 \wedge x2 = r3 \wedge y2 = r4$;
  **define** $width_{Rec}(\mathbf{this}, v)$ :  $\exists r1, r3 \cdot private_{Rec}(\mathbf{this}, r1, \text{-}, r3, \text{-}) \wedge v = r3 - r1$;
  **define** $height_{Rec}(\mathbf{this}, v)$ :  $\exists r2, r4 \cdot private_{Rec}(\mathbf{this}, \text{-}, r2, \text{-}, r4) \wedge v = r4 - r2$;
  $Rec()$
  $\langle\rangle\langle \exists v1, v2 \cdot v1 = 1 \wedge v2 = 1 \wedge width(\mathbf{this}, v1) \wedge height(\mathbf{this}, v2)\rangle$
  **void** $ScaleH(\mathbf{int}\ factor)$
  $\langle 0 \leq factor \wedge width(\mathbf{this}, d)\rangle\langle \exists v \cdot width(\mathbf{this}, v) \wedge v = d \times factor/100\rangle$
}
```

Fig. 4. *Rec* Translated into Abstract Predicates

the scope of the existential quantifiers. We will consider about the ownership technique in Section 3.6.

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] *or* [H-IPRE] *to translate it into a predicate definition;*
- *For each* **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, interfaces, 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 *private* in the fourth step. Thus, the resulting specification is well-encapsulated. □

Now we use the example of class *Rec* given in **Fig. 1** to show how does our translation algorithm work.

Example 1. **Fig. 4** is the result of our translation, where the method bodies are omitted. We use rule [H-PPRE] to define the predicate $private_{Rec}$ which specifies the values of the private fields (line 4); then use [H-MPRE] to translate the definition of model field *width* and *height* into predicates (line 5-6); use [H-MOLD] to translate the expression **old**(*width*) of post-condition into predicates based specification (line 11); and use [H-TRANS] to translate the method specifications (line 8 and 11) at last. In this example, we do not need to use [H-INV] since the invariant does not mention any model field.

Furthermore, we can prove the following theorem.

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 $private(\ldots)$ or $private_C(\ldots)$ 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. $\Box$

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

```
class $Rec$ : Object{
    int $x1, y1, x2, y2$;
    invariant $x1 \leq x2 \wedge y1 \leq y2$;
    define $width_{Rec}(\textbf{this}, v)$ :  $v = x2 - x1$;
    define $height_{Rec}(\textbf{this}, v)$ :  $v = y2 - y1$;
    . . . //omitted, the same as Fig. 4
}
```

The definitions of predicates become simpler. Here $private_{Rec}$ disappears, and the definitions of $width_{Rec}$ and $height_{Rec}$ take a simpler form. However, the remaining part of the class, including the method specifications, are the same as in **Fig. 4**. This example also shows that abstraction is a good way to decouple the abstract specification from the implementation details. Although the definition of abstract predicates changes, the method specifications remain the same.

3.2 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 [14,10].

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*.*

We can prove the following theorem:

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) The model fields can depend on all the (model and concrete) fields in the same class. Since S is well-formed, thus after our translation, the abstract predicates in T will only depend on fields of this class. As in Definition 4, the abstract predicates are allowed to depend on those fields. Therefore T meets the second requirement.

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

3.3 Framing

Framing is an important problem in verification. It requires the specification to specify the boundary of memory locations modifiable/modified by the methods. For example, for the program given in **Fig. 1**, we must be able to prove that method *ScaleH* does not modify *height*; and in Example 2, if $height(\textbf{this}, v')$ holds in the pre-state of method *ScaleH*, we must be able to conclude that this assertion also holds in the corresponding post-state.

Several solutions are proposed to deal with the frame problem. Literature on model fields often uses a **modifies** clause to denote the locations that the method is allowed to modify, as shown in **Fig. 1**. If a (model or concrete) field is not mentioned in the **modifies** clause of a method, we know that it is not modified by the method. Thus, we are sure that *ScaleH* does not modify *height*. More details about verifying the **modifies** clauses can be found in [15].

Specifications based on abstract predicates use separation logic notations in assertions, to explicitly describe how the heap is composed of separated parts which are independent of each other. The specification in **Fig. 2** uses only one predicate, thus we can directly infer that argument h is not modified by *ScaleH*. But even if we use more predicates to define *width* and *height* (as in Example 2), as long as they are separated from each other, we can still prove that *ScaleH* does not modify *height* by using frame rule.

```
class $SRec$ : Object{
    int $x1, y1, x2, y2, scale$;
    invariant $x1 \leq x2 \wedge y1 \leq y2 \wedge scale \geq 0$;
    model int $width$ such_that $width = (x2 - x1) \times scale$;
    model int $height$ such_that $height = (y2 - y1) \times scale$;
    $SRec()$
    $\langle\rangle\langle width = 1 \wedge height = 1\rangle$
    void $ScaleH$(int $factor$)
    $\langle 0 \leq factor\rangle\langle width = \mathbf{old}(width) \times factor/100\rangle$
    modifies $width, x2$;
}
```

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

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.

Fig. 6 shows all the new rules. Here we provide some explanations.

[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. Obviously, using this rule does not change the semantics of specification.

[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.

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

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

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

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

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

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

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

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*[4]*;*
- *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;*

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

– *Use* [P-MDF] *to delete all the* **modifies** *clauses.*

We can prove the following theorem about frame properties:

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 3. Now we consider the class *SRec* in **Fig. 5**. After using **Algorithm 1** and **2**, the resulting specification is:

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

After some simplification, the method specifications will become:

$$\begin{array}{l}
\mathit{SRec}() \\
\langle\rangle\langle \mathit{widthHeight}(\textbf{this}, 1, 1)\rangle \\
\textbf{void } \mathit{ScaleH}(\textbf{int } \mathit{factor}) \\
\langle 0 \leq \mathit{factor} \wedge \mathit{widthHeight}(\textbf{this}, d, v')\rangle\langle \mathit{widthHeight}(\textbf{this}, d \times \mathit{factor}/100, v')\rangle
\end{array}$$

Thus we are able to prove that the second parameter of *widthHeight* is not modified by *ScaleH*.

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, we cannot modularly translate the specification now. The requirement of [P-MRG] that

$$p^\alpha_C, p^k_C \text{ are not separated}$$

should be modified to

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

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.

3.4 Further Investigation

Previously, we have proved that the abstract predicates are not less expressive than model fields. Now we show a program which can be specified by abstract predicates, but cannot be specified by model fields. i.e. 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 between the subclass and the superclass can be proved if the abstract specification of subclass refines the abstract specification of superclass, no matter how the subclass overrides the predicates. On the other hand, an inherited model field cannot modify its constraint arbitrarily. Some literature [15,16] do not allow any modification, and some other literature [4,8] only allow the inherited model field to strengthen the constraint. Either is much more limited than what can do with the abstract predicates.

For example, let us consider a method $C.m()\{\textbf{return this}.x;\}$, and an overridden method $D.m()\{\textbf{return } 2 \times \textbf{this}.x;\}$, where D is a subclass of C. It is impossible to accurately specify their return values using model fields, since the behavioural subtype relation will be violated in that case. However, we can use overridden predicates to accurately specify both methods, as shown in [10].

Furthermore, when dealing with recursion data structures, abstract predicates are more expressive than model fields. For example, if we need to define the length of a linked list, we can define a recursive abstract predicate:

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

However, if we use model fields (without ownership mechanism), we cannot define the length of the linked list. Since the constraints of model fields are invariant-like properties (it must hold in all program states), the model fields cannot depend on the fields of other objects. That is to say, 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 [8], 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

```
class Square : Rec{
  invariant $x2 - x1 = y2 - y1$;
  define $rec_{Square}(\mathbf{this}, w, h) :\ w = x2 - x1 \wedge w = y2 - y1$;
  void ScaleH(int factor)
  $\langle (0 \le factor) \wedge rec(\mathbf{this}, w, h) \rangle \langle rec(\mathbf{this}, w \times factor/100, h) \rangle$
  //the abstract specification is still the same as Rec.ScaleH
  { ··· }
}
```

Fig. 7. Using Abstract Predicates to Specify Class *Square*

on *next.len*, each node must own the next node. But this design will prevent us from looping through the list[5].

Here we give another example, which can be specified naturally using abstract predicates, but it is at least difficult to specify using model fields.

Example 4. We define a class *Square* as a subclass of *Rec*, as shown in **Fig. 7**. It introduces a new invariant $x2 - x1 = y2 - y1$, thus, method *Square.ScaleH* modifies not only $x2$, but also $y2$. This is a valid OO program and we can specify it using abstract predicates. Class *Square* overrides the predicate *rec*, but it does not modify the abstract specification of *ScaleH*. Thus *Square* is a behavioral subtype of *Rec*.

However, we cannot find a way to specify this example using model fields and **modifies** clauses. The **modifies** clause in *Rec* (**Fig. 1**) allows *Rec.ScaleH* to modify only *width* and $x2$, but *Square.ScaleH* needs to modify $y2$ now. In this case, we cannot prove the behavioral subtype relation.

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

3.5 Correctness

In this subsection, we prove the correctness of the translation rules. We will prove that: a program satisfies the model fields based specification, iff the program satisfies the predicates based specification generated by our rules.

Theorem 5. *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,*

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

[5] 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 JML, we cannot loop through the list and modify its elements, since it violates the *owner-as-modifier* discipline; in Spec#, we cannot loop through the list since the owner of a variable must keep unchanged in a loop.

Here Δ is the assumption context introduced in Section 2.3, $\overline{v_i}$ *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]} \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}}) \wedge \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}}) \wedge \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}}) \wedge \mathit{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 \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) \wedge \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) \wedge \bigwedge_j \mathit{private}_{C_j}(o_j, \overline{\_}, r_j, \overline{\_})) \\
\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) \wedge \bigwedge_j \mathit{private}(o_j, \overline{\_}, r_j, \overline{\_})).
\end{aligned}$$

□

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

Theorem 6. *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]:

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

For [H-MOLD]:

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

and vice versa. □

Now we can conclude that:

Theorem 7 (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 5** and **6**, given the definitions of predicates, all these rules do not change the semantics of specification. □

For the translation rules in Section 3.3, we have:

Theorem 8 (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 $\mathit{\Delta}$ contains all the formulas about predicates p^k and p^α. Then for [P-SBST]:

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

For [P-MTHD]:

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

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

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 $O(mn)$.

3.6 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, papers [15,4,8] introduce the ownership [13] mechanism into model field approaches. With this mechanism, a model field declaration in class C takes the form:

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

Here o ranges in all the objects which are (transitive) owned by **this** object, including **this** itself.

On the other hand, since the predicate can already mention other objects in its definition, the literature on abstract predicates does not take the ownership mechanism into account [17].

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 **this**" to "$\overline{o_i.f_i}$ are model fields of **this** and the objects **this** transitive owns", and replace "$\overline{f_i}$" with "$\overline{o_i.f_i}$" in this 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. Instead, we can define a predicate for each class:

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

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 $\overline{o_i}$:

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

Here ret denotes the return value. When reasoning about the aggregate object (o together with $\overline{o_i}$), the multi-object invariant can be inferred from [H-INVM]; when reasoning about the other objects, the multi-object invariant can be inferred from 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, as suggested by [17].

4 Related Work

The concept of *model fields* was proposed by Leino in his PhD thesis [3], where it was called *abstract fields*. In the thesis, these fields are used to specify/verify data refinement. However, the work has some serious limitations, e.g. it almost ignores information hiding, and requires the abstract fields to be uniquely determined by the constraint.

Müller *et al.* in [15] presented a modular technique for specifying/verifying frame properties, by 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.

In [8] Leino and Müller presented a modular verification methodology for model fields. In this work, updating concrete fields does not automatically modify the related model fields. Instead, model fields of an object are updated only when the object is valid, i.e. in the states that its invariants are known to hold. 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 3.4, the expressiveness of the work has some limitations.

There are also some applications of model fields. For instance, model fields are used to express the abstract specifications of classes and interfaces in specification language JML [4], to write and check *design by contract* assertions [16], and to achieve information hiding in the specifications for interfaces [18].

The concept of *abstract predicates* was proposed by Parkinson and Bierman [5] in developing an OO program verification system based on separation logic. Lately they take inheritance into account [10], and introduce static/dynamic specifications for each method, to avoid restricting subtypes' behaviors or re-verifying inherited methods. A later work [19] considers the interfaces, and allows user to describe both static and dynamic specification in one statement. But the paper does not analyze multiple inheritance brought by interfaces, nor the behavioral subtyping between multiple specifications, etc.

Liu *et al.* proposed an OO verification framework in [6], whose abstraction technique is similar to Parkinson's. This work can deal with the inheritance by using simpler notations. It requires only one specification for each method and avoids re-verification as well. Qiu *et al.* extend the framework for interfaces [20], and give a solution to the problems brought by multiple inheritance.

Researchers have also proposed some other abstraction techniques for OO verification, e.g., pure methods [21] and data groups [22]. 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 all the members of the group. By this technique, we can avoid listing the private fields explicitly.

Although many papers focus on the model fields or abstract predicates based techniques, a systematic comparison between the two techniques is lacking. Burgman [23] proposed a comparison between JML and Separation Logic in verifying multi-threaded Java programs. But the work does not focus on the abstraction mechanisms. The motivation of this paper is to deeply understand the two abstraction techniques and their relationship, and to understand what is really needed for the abstract mechanisms in OO verification.

5 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, if and only if it satisfies the resulting abstract predicates based specification. Thus, the abstract predicates are not less expressive than model fields. 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. Leino, K.R.M., Nelson, G.: Data abstraction and information hiding. ACM Trans. Program. Lang. Syst. **24**(5) (September 2002) 491–553
2. Müller, P.: Modular Specification and Verification of Object-oriented Programs. Springer, Berlin, Heidelberg (2002)
3. Leino, K.R.: Toward Reliable Modular Programs. PhD thesis, California Institute of Technology, Pasadena, CA, USA (1995) UMI Order No. GAX95-26835.
4. Leavens, G.T., Baker, A.L., Ruby, C.: Preliminary design of JML: A behavioral interface specification language for Java. SIGSOFT Software Engineering Notes **31**(3) (May 2006) 1–38
5. Parkinson, M., Bierman, G.: Separation logic and abstraction. In: Proceedings of the 32Nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. POPL '05, New York, NY, USA, ACM (2005) 247–258
6. Liu, Y., Hong, A., Qiu, Z.: Inheritance and modularity in specification and verification of OO programs. In: TASE'11, IEEE Computer Society (2011) 19–26
7. Bruns, D.: Formal semantics for the Java Modeling Language. Diplomarbeit, Universität Karlsruhe (June 2009)
8. Leino, K.R.M., Müller, P.: A verification methodology for model fields. In Sestoft, P., ed.: Programming Languages and Systems. Volume 3924 of Lecture Notes in Computer Science. Springer (2006) 115–130
9. Reynolds, J.C.: Separation logic: A logic for shared mutable data structures. In: Proceedings of the 17th Annual IEEE Symposium on Logic in Computer Science, Washington DC, USA, IEEE Computer Society (2002) 55–74

10. Parkinson, M.J., Bierman, G.M.: Separation logic, abstraction and inheritance. In: Proceedings of the 35th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. POPL '08, New York, NY, USA, ACM (2008) 75–86
11. Barnett, M., Leino, K.R.M., Schulte, W.: The spec# programming system: An overview. In: Proceedings of the 2004 International Conference on Construction and Analysis of Safe, Secure, and Interoperable Smart Devices. Volume 3362 of Lecture Notes in Computer Science., Marseille, France, Springer (2005) 49–69
12. Jacobs, B., Piessens, F.: The verifast program verifier. CW Reports (2008)
13. Clarke, D.G., Potter, J.M., Noble, J.: Ownership types for flexible alias protection. In: Proceedings of the 13th ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications, ACM (1998) 48–64
14. Leavens, G.T., Poll, E., Clifton, C., Cheon, Y., Ruby, C., Cok, D., Müller, P., Kiniry, J., Chalin, P.: JML Reference Manual. (2008)
15. Müller, P., Poetzsch-Heffter, A., Leavens, G.T.: Modular specification of frame properties in jml. Concurrency and Computation: Practice and Experience **15**(2) (2003) 117–154
16. Cheon, Y., Leavens, G., Sitaraman, M., Edwards, S.: Model variables: Cleanly supporting abstraction in design by contract: Research articles. Software: Practice and Experience **35**(6) (May 2005) 583–599
17. Parkinson, M.: Class invariants: The end of the road. In: International Workshop on Aliasing, Confinement and Ownership. Volume 23. (2007)
18. Leavens, G.T., Müller, P.: Information hiding and visibility in interface specifications. In: Proceedings of the 29th International Conference on Software Engineering. ICSE '07, Washington, DC, USA, IEEE Computer Society (2007) 385–395
19. Parkinson, M., Bierman, G.: Separation logic for object-oriented programming. In: Aliasing in Object-Oriented Programming. Types, Analysis and Verification. Volume 7850 of Lecture Notes in Computer Science. Springer (2013) 366–406
20. Qiu, Z., Hong, A., Liu, Y.: Modular verification of OO programs with interfaces. In: Proceedings of the 14th International Conference on Formal Engineering Methods. Volume 7635 of Lecture Notes in Computer Science., Springer (2012) 151–166
21. Darvas, Á., Leino, K.R.M.: Practical reasoning about invocations and implementations of pure methods. In: Fundamental Approaches to Software Engineering. Springer (2007) 336–351
22. Leino, K.R.M.: Data groups: Specifying the modification of extended state. In: Proceedings of the 13th ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications, New York, NY, USA, ACM (1998) 144–153
23. Burgman, R.: Specifying multi-threaded java programs. 12th Twente Student Conference on IT, available at `http://referaat.cs.utwente.nl/conference/12/paper` (2010)