Formal Semantics of Model Fields in Annotation-based Specifications* **

Bernhard Beckert and Daniel Bruns

Karlsruhe Institute of Technology (KIT), Department of Informatics

Abstract. It is widely recognized that abstraction and modularization are indispensable for specification of real-world programs. In source-code level program specification and verification, *model fields* are a common means for those goals. However, it remains a challenge to provide a well-founded formal semantics for the general case in which the abstraction relation defining a model field is non-functional.
In this paper, we discuss and compare several possibilities for defining model field semantics, and we give a complete formal semantics for the general case. Our analysis and the proposed semantics is based on a generalization of Hilbert's ε terms.

1 Introduction

Annotation-based specification. Recently, formal specification of programs at the source-code level has become increasingly popular with a wider community, where such specifications are used both in safety and security contexts. This is mainly due to the rise of a specification methodology where the boundaries between program proper and specification become blurred: specifications are written as annotations into program source files. The specifications languages extend the programming language's syntax, and their semantics is defined on top of the programming language's semantics. Particular examples include the Java Modeling Language (JML) [12] for Java, the ANSI/ISO C Specification Language (ACSL) [2] for C, and Spec# [1] for C#. This approach bears the clear advantage of being intuitively understandable to users who are not familiar with logic. At the same time, however, it has become far more laborious to come up with a sound semantical foundation.

Model fields. Even if programs are specified at the source-code level, abstraction and modularization are indispensable for handling real-word programs. For that purpose, the concept of model fields [9, 14] is widely used. Model fields

* This work was supported by the German National Science Foundation (DFG) under project "Program-level Specification and Deductive Verification of Security Properties" within priority programme 1496 "Reliably Secure Software Systems – RS^3".

are abstractions of the program's memory state given in a syntactically convenient form (as fields in a class). The relation between the concrete state and the model fields, i.e., the abstraction relation, is specified by so-called *represents clauses*. In general, abstraction relations may be non-functional, and they may refer to entities which are not present in the concrete program (e.g., other model fields). Model fields are also commonly used to give implementation-independent specifications of abstract data types. In that case, the requirement specification only refers to model fields while the abstraction relation is part of the (hidden) implementation details.

Topic of this paper. There is yet no common understanding of what the semantics of model fields is in the general case. The semantics used by verification and runtime checking tools as well as the semantics defined in the literature is restricted to functional represents clauses, to model fields of a primitive type, or by restricting the syntax of represents clauses. The general case, however, raises several questions:

- On which memory locations does the value of a model field depend?
- What value is chosen if the represents clause is non-functional?
- Is there aliasing between model fields of a reference type?
- What does an unsatisfiable represents clause mean? Does it lead to an inconsistent axiom set?
- At what points in time does the value of a memory field change?
- In which cases are represents clauses well-defined? What about recursive represents clauses?

In this paper, we answer these questions by presenting a well-founded semantics for model fields in the general case, which is inspired by a generalization of ε terms. Although we primarily report on the situation in the Java Modeling Language (JML)—this is the specification language the authors know best—we expect the principles outlined here to apply to other specification languages which make use of model fields as well. In particular, the semantics is independent of any reasoning system. It is parameterized so that it can be instantiated in different ways in cases where different reasonable answers to the above questions exist.

Hilbert's ε terms. The concept of ε terms was first introduced by Hilbert in 1939 as an extension to classical first-order predicate logic [8]. An ε term $\varepsilon x.\varphi(x)$, where x is a variable and φ is a formula, stands for 'some domain element such that φ holds (if such exists)'. In general, this informal understanding is the only restriction on the value of an ε term; however, we will later see that further useful restrictions may be added. These terms can, for example, be used to represent instantiations of existentially quantified variables without assigning a concrete value (i.e., skolemization). The quantifier in $\exists x.\varphi(x)$ can be eliminated by replacing the formula $\exists x.\varphi(x)$ by $\varphi(\varepsilon x.\varphi(x))$.

In the course of this paper, we will review the discussion on restrictions to the valuation of ε terms, with a look for analogies in the discussion on model field semantics.

Structure of this paper. In Section 2, we give a short introduction to JML and, in particular, model fields in JML. We however expect the results presented here to apply to other specification languages as well. In Section 3, we give a formal semantics for JML expressions *without model fields*. Before extending this semantics to model fields, we review the discussion on semantics of ε terms and develop an n-ary generalization of ε terms (Section 4). Then, in Section 5, we present a first approach to model field semantics inspired by ε terms. In Section 5.1, we discuss some of the desired properties of this approach. We discuss one deficiency in Section 5.2 and propose a solution in a second, extended approach based on our generalized ε terms from Section 4.1.

2 The Java Modeling Language

The Java Modeling Language (JML) [12] was conceived to be both easily accessible to the 'common programmer'—who might *not* be skilled in formal modeling—and "capable of being given a rigorous, formal semantics" [11]. JML specifications, i.e., primarily method contracts (including frame conditions) and class invariants, are written in a Java-like expression language as comments in source files. Due to this advantage, JML has quickly become one of the most popular specification languages, which is employed in numerous program verification and runtime checking methods and tools. Despite good intuitive understanding of JML, there is yet no canonized formal semantics. Some have been presented before [4–6, 10], including one of the authors' own. But, semantics of model fields, in particular, are subject to an ongoing debate.

In addition to plain Java expressions, JML expressions are enriched with quantification and special constructs, such as `\result` to access the return value of a method invocation, or `\old` to refer to pre-state values of expressions (in a method's post-state) as well as quantification over primitive and reference types.

Model fields in JML are declared similarly to regular Java fields, but within specifications. They may occur as either non-static (i.e., instance) or static fields. A model field is declared with the additional modifier `model`. This does not yet give any information on the value of the field but only type information. To impose a constraint on the possible values of a model field, a (separate) `represents` clause is provided. It comes in two variations: a functional form, in which a model field points to exactly one value depending on the memory state, and a relational form, in which it is constrained to values satisfying a boolean expression. Obviously, the functional form is a special case of the relational form. The functional form is indicated by the assignment operator `=`, while a relational definition is indicated by the keyword `\such_that`; see Fig. 1 for an example. Since they are not describing the relationship between two states, represents clauses are not allowed to contain `\old` or `\result`. Otherwise, their syntax is not restricted. They may, for example, contain references to other model fields. We will call such references the *dependencies* of the model field.

```
public class List {                                                    1
    private int[] theList;                                             2
    /*@ model int bound;                                               3
      @ model int idxMax;                                              4
      @ represents bound \such_that                                    5
      @     (\forall int i; 0 <= i &                                   6
      @         i < theList.length;                                    7
      @         bound >= theList[i]);                                  8
      @ represents idxMax \such_that                                   9
      @     theList[idxMax] == bound;                                  10
      @*/ }                                                            11
```

Fig. 1. Two model fields with non-functional represents clauses. While there are always multiple possible values for `bound`, the relation for `idxMax` may be empty.

3 Semantics of JML Expressions

In this section, we summarize the framework given in [4] to evaluate JML expressions that do *not* contain references to model fields. This semantics is then extended to the case of model fields in Section 5.

We assume that a closed Java program is given that is annotated with JML specifications. This program provides a type hierarchy, i.e., a partially ordered set $(\mathcal{T}, \sqsubseteq)$ of types, and a *universe* $\mathcal{U}$. The universe $\mathcal{U}$ consists of all semantical objects which may be referenced in the program. In particular, it includes the mathematical integers and truth values tt (true) and ff (false). For each reference type $T \sqsubseteq \texttt{Object}$ there is a countably-infinite subset $V_T \subset \mathcal{U}$ which serves as a reservoir and contains the elements of that type and a special element *null*; $T' \sqsubset T$ implies $V_{T'} \subset V_T$. When needed, we denote the set of *direct instances* by $V_T^{\mathfrak{d}}$, with the property $V_T^{\mathfrak{d}} \cap V_{T'}^{\mathfrak{d}} = \emptyset$ if $T \neq T'$. For a primitive type T, V_T and $V_T^{\mathfrak{d}}$ are identical and map directly to the corresponding mathematical entities, e.g., $V_{\texttt{int}} = \{z \in \mathbb{Z} \mid -2^{31} \leq z < 2^{31}\}$. We also require that $\mathcal{T} \subset \mathcal{U}$ in order to allow types (i.e., classes and interfaces) to act as the receiver of a static field or method. The declared type of a field identified by `x` is denoted by *typeof*(`x`). Let *Id* denote the set of valid Java identifiers and *Expr* the set of syntactically well-formed JML expressions. See [12, Appendix A] for the complete syntax.

A *(system) state* s is the union of functions η and σ, which represent the heap and the stack memory, respectively. $\eta : \mathcal{U} \times Id \to \mathcal{U}$ maps pairs of *receiver objects* and field identifiers (these pairs are also called *locations*) to a value. $\sigma : Id \cup \{\texttt{this}\} \to \mathcal{U}$ evaluates local variables. For now, we assume both η and σ to be total functions.[1] For a total function f and a partial function p, we use the notation $f \oplus p$ to indicate that p overrides f, i.e., $(f \oplus p)(x) = p(x)$ if $x \in \mathrm{Id}(p)$ and $f(x)$ otherwise. We omit $\oplus$ where the notion is clear.

We define the evaluation function $val : \mathcal{S}^2 \times Expr \to \mathcal{U}$ that, given a pair of system states, maps expressions to elements of the universe. The two state

[1] This can be achieved through underspecification of otherwise undefined values.

parameters represent the state referred to by the `\old` operator (pre-state) and the current state (post-state), respectively. Table 1 shows the valuation function for some exemplary parameters. For a comprehensive definition, the reader is referred to [4, Appendix A]. For the sake of simplicity, we disregard the fact here that JML even allows pure methods to have certain side-effects.

field access:	$val(s_0, s_1, a.\texttt{x}) = \eta(val(s_0, s_1, a), \texttt{x})$	where $s_1 = (\eta, \_)$
variable:	$val(s_0, s_1, \texttt{x}) = \sigma(\texttt{x})$	where $s_1 = (\_, \sigma)$
constant:	$val(s_0, s_1, \texttt{true}) = tt$	
identity:	$val(s_0, s_1, a \texttt{ == } b) = \begin{cases} tt & val(s_0, s_1, a) = val(s_0, s_1, b) \\ ff & \text{otherwise} \end{cases}$	
logical and:	$val(s_0, s_1, a \texttt{ \&\& } b) = \begin{cases} tt & val(s_0, s_1, a) = val(s_0, s_1, b) = tt \\ ff & \text{otherwise} \end{cases}$	
implication:	$val(s_0, s_1, a \texttt{ ==> } b) = \begin{cases} tt & val(s_0, s_1, a) = ff \vee val(s_0, s_1, b) = tt \\ ff & \text{otherwise} \end{cases}$	
quantification:	$val(s_0, s_1, \texttt{(\forall T x; } a\texttt{; } b\texttt{)}) = \begin{cases} tt & \forall y \in V_T \setminus null : val(s_0\{\texttt{x} \mapsto y\}, s_1\{\texttt{x} \mapsto y\}, a \texttt{ ==> } b) = tt \\ ff & \text{otherwise} \end{cases}$	
old state:	$val(s_0, s_1, \texttt{\old(}a\texttt{)}) = val(s_0, s_0, a)$	

Table 1. Valuation function *val* for some representative expressions where a and b are expressions and `x` is an identifier.

4 Semantics of ε Terms

When Hilbert first introduced ε terms, he provided only a vague informal understanding of their semantics. This has led to some interesting discussions on a formalization. In this section, we mainly reprise the account given in [7].

A *pre-structure* of first-order logic with ε terms is a triple $\mathfrak{S} = (\mathcal{U}, \mathcal{I}, \mathcal{A})$ consisting of a domain $\mathcal{U}$, an interpretation $\mathcal{I}$ of predicates and functions as in classical logic, and additionally an *ε-valuation function* $\mathcal{A}$. The function $\mathcal{A}$ maps a term $\varepsilon x.\varphi$ and a variable assignment β to a value $\mathcal{A}(\varepsilon x.\varphi, \beta) \in \mathcal{U}$. To gain a 'more semantical' ε-valuation, *intensional* and eventually *extensional* semantics have been introduced.

Definition 1. *An* intensional structure $\mathfrak{S} = (\mathcal{U}, \mathcal{I}, \mathcal{A})$ *is a pre-structure in which the valuation of an ε term $\varepsilon x.\varphi$ only depends on the valuation of free variables occurring in φ, and $\mathcal{A}$ points to a value that actually satisfies φ if such a value exists:*

- *If* $\beta_1|_{fv(\varphi)} = \beta_2|_{fv(\varphi)}$ *then* $\mathcal{A}(\varepsilon x.\varphi, \beta_1) = \mathcal{A}(\varepsilon x.\varphi, \beta_2)$
  (where $\beta_i|_{fv(\varphi)}$ is the restriction of β_i to the free variables in φ).
- *If* $\mathfrak{S}, \beta \models \exists x.\varphi$ *then* $\mathfrak{S}, \beta\{x \mapsto \mathcal{A}(\varepsilon x.\varphi, \beta)\} \models \varphi$.

Intensional semantics may still assign different values to syntactically different but logically equivalent terms. *Extensional* structures, on the contrary, are built on a deterministic (total) choice function on the set of applicable values, the *extension*.

Definition 2. *The* extension *Ext of an ε term w.r.t. a structure and an assignment is defined as*

$$Ext(\mathfrak{S}, \beta, \varepsilon x.\varphi) := \{u \in \mathcal{U} \mid (\mathfrak{S}, \beta\{x \mapsto u\}) \models \varphi\} \quad .$$

An intensional structure $\mathfrak{S} = (\mathcal{U}, \mathcal{I}, \mathcal{A})$ with the following property is called an extensional structure*:*

– *If* $Ext(\mathfrak{S}, \beta, \varepsilon x.\varphi) = Ext(\mathfrak{S}, \beta, \varepsilon y.\psi)$ *then* $\mathcal{A}(\varepsilon x.\varphi, \beta) = \mathcal{A}(\varepsilon y.\psi, \beta)$.

Note that extensions may be empty; in that case, $\mathcal{A}$ yields an arbitrary element of the universe. Extensional semantics are strictly stronger than intensional semantics, in the sense that they have more valid formulas. Take, for instance, the formula $\varepsilon x.\varphi \doteq \varepsilon x.\neg\neg\varphi$. It is valid in any extensional structure, but not in all intensional structures.

4.1 A generalization of ε terms.

As we will further discuss in Section 5.2, model field specifications—in contrast to formulae in logic—are highly non-modular as represents clauses may depend on each other. Therefore, it is not always possible to express the value of a model field in terms of an ε term. In the following, we introduce the notion of *generalized ε terms*, which denote values for non-empty finite sequences of variables instead of single variables.

Definition 3 (Generalized ε term, syntax). *Let $\bar{x}$ be a non-empty finite sequence of pairwise distinct variables, let $i \in \mathbb{N}$, and let φ be a formula.*

Then $\varepsilon\bar{x}_i.\varphi$ is a generalized ε term.

Definition 4 (Generalized extension). *The* generalized extension *Ext of a generalized ε term $\varepsilon\bar{x}_i.\varphi$ w.r.t. a structure and an assignment is defined as*

$$\begin{aligned} &Ext(\mathfrak{S}, \beta, \varepsilon\langle x_0, \ldots, x_{n-1}\rangle_i.\varphi) \quad := \\ &\quad \{\langle u_0, \ldots, u_{n-1}\rangle \in \mathcal{U}^n \mid \mathfrak{S}, \beta\{x_j \mapsto u_j \mid 0 \leq j < n\} \models \varphi\} \end{aligned}$$

Note that the generalized extension contains n-tuples and is independent of the index i. We now extend the definition of structures to the case of generalized ε terms. The conditions imposed on the ε-evaluation function $\mathcal{A}$ in the following definition implies both the requirements made in Definitions 1 and 2, i.e., every generalised ε structure is extensional:

Definition 5 (Generalized ε term, semantics). *A pre-structure $\mathfrak{S} = (\mathcal{U}, \mathcal{I}, \mathcal{A})$ with the following properties is called a* generalized ε structure*:*

– ***(Intensionality)*** *If* $Ext(\mathfrak{S}, \beta, \varepsilon\bar{x}_0.\varphi) \neq \emptyset$ *then*

$$\left\langle \mathcal{A}(\varepsilon\bar{x}_0.\varphi, \beta), \ldots, \mathcal{A}(\varepsilon\bar{x}_{|\bar{x}|-1}.\varphi, \beta) \right\rangle \in Ext(\mathfrak{S}, \beta, \varepsilon\bar{x}_0.\varphi)$$

– ***(Extensionality)*** *If* $Ext(\mathfrak{S}, \beta, \varepsilon\bar{x}_0.\varphi) = Ext(\mathfrak{S}, \beta, \varepsilon\bar{y}_0.\psi)$ *then* $\mathcal{A}(\varepsilon\bar{x}_i.\varphi, \beta) = \mathcal{A}(\varepsilon\bar{y}_i.\psi, \beta)$ *for any* i.

5 A Novel Approach to Model Field Semantics

We return to the evaluation of expressions in JML and present a first approach to model field semantics that is inspired by extensional ε term semantics (without using ε terms explicitly). As commonly accepted, program references can be approximately identified with variables in logic, as system states can be identified with valuations. In a way similar to the ε-valuation function $\mathcal{A}$ introduced above, we define a model field valuation function $\varepsilon : \mathcal{S} \times \mathcal{U} \times Id \times Expr \to \mathcal{U}$, which takes a state, a receiver object, a model field's identifier, and a constraining expression (from the represents clause) as parameters. We then build the definition of an extended valuation function val_ε for JML expressions on top.

Let us first define the *extension*, i.e., the set of semantical objects for which a state s validates a boolean JML expression φ with a field identifier `x` and receiver object o which simulates a heap location $(o, \mathtt{x})$, in a way reminiscent to the above definition:

Definition 6 (Extension).

$$Ext_\varepsilon(s, o, \mathtt{x}, \varphi) \;:=\; \left\{ u \in V_{typeof(\mathtt{x})} \mid val_\varepsilon(s, s\{\mathtt{this} \mapsto o, (o, \mathtt{x}) \mapsto u\}, \varphi) = tt \right\}$$

The extension is defined w.r.t. only one state. The reason is that the `\old` operation may not occur in represents clauses.

An extensional ε-valuation function is independent of the syntactical shape of the constraining formula but only depends on its extension. It therefore can be seen as a deterministic choice function $\chi : 2^{\mathcal{U}} \to \mathcal{U}$ applied on the extension set.[2] This seems plausible—except that in program specification, this is against the intuitive view that different locations hold values which are independent of each other. In other words: all model fields with logically equivalent represents clauses would be `==`-equal. Therefore, we introduce (possibly) different choice functions for different model fields through a weaker version of extensionality and instead use a *family* of choice functions that contains a choice function $\chi_{(T,i)}$ for each type $T \in \mathcal{T}$ and identifier $i \in Id$.

Let T' be the type where `x` is declared. Then, an ε-valuation w.r.t. a choice function $\chi_{(T',i)}$ is defined by:

$$\varepsilon(s, o, \mathtt{x}, \varphi) := \chi_{(T',\mathtt{x})}\left(Ext_\varepsilon(s, o, \mathtt{x}, \varphi)\right) \quad .$$

[2] Note that χ is a total function and, in particular, yields an underspecified value $\chi(\emptyset)$.

In addition to the above mentioned information, we need to extract representation clauses from the annotated program. Let $rep(T, \mathtt{x})$ denote the represents clause declared in type T constraining model field $\mathtt{x}$. We are finally able to extend our definition of val from Sect. 3 to val_ε to include model field validation:

Definition 7. *If* $\mathtt{x}$ *is a model field, then*

$$val_\varepsilon(s_0, s_1, a.\mathtt{x}) := \varepsilon(s_1, val_\varepsilon(s_0, s_1, a), \mathtt{x}, rep(val_\varepsilon(s_0, s_1, a), \mathtt{x}))$$

Although extension and valuation are defined mutually recursively, this definition is well-founded since there is only a finite number of model fields which are referenced in a single expression.

5.1 Discussion

Frame properties. Framing is an essential means to specify and verify programs in a modular way. With the following remark, our approach for defining the semantics of model fields allows framing, i.e., restricting the possible assignments for fields. This is a purely semantical criterion—without (syntactically) naming dependencies explicitly.

Assuming a fixed choice function χ and, thus, a fixed valuation ε, there is only one possible object value for each model field as long as the values of its concrete dependencies remain unchanged, even if other parts of the state change.

The value for a model field can be observed in *any* state without additional care. While some authors [13, 15, 16] define model field semantics only for states in which the receiver object's invariant holds, our definition is independent of the particular semantics of invariants.

Handling undefinedness. Our definition of semantical evaluation of model fields is independent of how undefinedness in expressions is handled (e.g., division by zero or null pointer references). In JML—as in Java—an expression φ is only well-defined if its subexpressions are themselves well-defined, namely those which are relevant in a short-circuit evaluation read from left to right. On the top level, a boolean expression is considered valid if it is well-defined and yields the value tt. This can be seen as a non-symmetric, conservative three-valued logic. One could easily extend this notion of well-definedness to model fields where a reference expression is well-defined only if there is a non-empty extension.

Applications. Up to now, there exist various tools which use annotation-based languages as specification input: runtime checkers, static analyzers, and formal deductive verification tools. As our semantics is independent from any verification methodology, we believe that it can be used to check whether those applications implement model fields in a consistent way. For runtime-checking, however, it may be necessary to fix a certain choice function which is easy to compute, e.g., to choose the least element w.r.t. some order.

5.2 An improved approach

The above approach works well in most cases—even when a represents clause contains references to other model fields. However, the evaluation is local to single model fields in the sense that it only establishes the relations between a model field and its dependencies. In the case where references are cyclic, the relation between model fields are ignored. Consider the following two represents clauses:

```
represents x \such_that x >= y; represents y \such_that y >= x;
```

Both are clearly satisfiable, but when evaluated separately, it is not implied that `x` and `y` are assigned the same value. For a sound evaluation in that case, instead of making a choice from a set of *values*, we need to make a choice from the set of *valuations* conforming with all represents clauses simultaneously. Let $\mathcal{L}_s$ be the (finite) set of model field locations (i.e., pairs of receiver objects/classes and field identifiers) whose receiver object is created in state s (or has been statically initialized). Then, the *heap extension HExt*, as motivated by generalized extensions (Def. 4), can be given as follows. It consists of functions from locations to values—the same domain as the heap—under which all represents clauses valuate to true. This means that those functions extend the actual heap. Since there is a valuation for each model field, only one choice function is required.

Definition 8 (Generalized model field valuation). *Let $\bar{\chi}$ be a fixed choice function on $\mathcal{U}^{\mathcal{U}\times Id}$. Then define heap extension $HExt(s)$ and valuation $val_{\bar{\varepsilon}}$:*

$$HExt(s) := \left\{ h \in \mathcal{U}^{\mathcal{U}\times Id} \mid \forall (o, \mathtt{x}) \in \mathcal{L}_s.\, val_{\bar{\varepsilon}}(s, s \oplus h \oplus \{\mathtt{this} \mapsto o\}, rep(o, \mathtt{x})) = tt \right\}$$
$$val_{\bar{\varepsilon}}(s_0, s_1, a.\mathtt{x}) := (\bar{\chi} HExt(s_1))\,(val_{\bar{\varepsilon}}(s_0, s_1, a), \mathtt{x})$$

The heap extension set may lead to fewer values for a particular model field when compared with the simple extension defined above. Those belonged to a partial (local) solution in which not *all* represents clauses are satisfied simultaneously. However, all aspects which we discussed in Sect. 5.1 still apply to this definition.

```
public class LinkedList {                                          1
    private /*@ nullable @*/ LinkedList next;                      2
    private Object contents;                                       3
    /*@ model int index;                                           4
      @ represents index \such_that                                5
      @             next == null || index < next.index; @*/        6
}                                                                  7
```

Fig. 2. Non-functional represents clause: index of a linked list.

Example 1. Figure 2 shows an implementation of a linked list. In JML, it is necessary to add the modifier `nullable` to the `next` list element because as the default all object references must not point to `null` unless declared explicitly.

The represents clause of the model field `index` guarantees that the list is actually acyclic as its value needs to be strictly less than `index` of the next element. Let us determine the value of `this.index` in a state $s = (\eta, \sigma)$ where

$$\sigma(\texttt{this}) = ll_0, \;\; \eta(ll_0, \texttt{next}) = ll_1, \;\; \eta(ll_1, \texttt{next}) = ll_2, \;\; \eta(ll_2, \texttt{next}) = null$$

and $ll_0, ll_1, ll_2 \in V_{\texttt{LinkedList}}$ are the only objects created in s. From this it follows that $\mathcal{L}(s) = \{(ll_0, \texttt{index}), (ll_1, \texttt{index}), (ll_2, \texttt{index})\}$. Then

$$\mathit{HExt}(s) = \left\{h \in (V_{\texttt{int}})^{\mathcal{U} \times Id} \mid h(ll_0, \texttt{index}) < h(ll_1, \texttt{index}) < h(ll_2, \texttt{index})\right\} \;\; ,$$

which is clearly not empty and some function can be chosen.

6 Related Work

Even though JML is designed to be interchangeably used with various specification and program analysis techniques, the issue of handling model fields is still subject to an on-going debate. There are several approaches to integrate model fields into verification. There, semantics are mostly implict in the respective methodology or calculus. A preliminary version of the semantics in this work has also appeared in [4, Sect. 3.1.5].

Substitution-based Approaches. Breunesse and Poll [3] present a semantics using substitutions in expressions. The clear advantage of this technique is that no additional evaluation rules are needed. Given a model field `x` of type T with a represents clause ψ and a JML expression φ which contains `x`, the following transformation is applied: $\varphi \;\; \leadsto \;\;$ `(\forall` T `x;` ψ`;` φ`) && (\exists` T `x; true;` ψ`)`

In the result, φ appears as the body of a quantifier expression with the model field `x` as the quantified variable. This transformation is done for every model field declared in the program. The resulting expression asserts both that φ holds if the represents clause ψ is true as well as the existence of a value that satisfies ψ.

However, this approach is syntactically restricted since the order in which an expression is transformed does matter. Moreover, model fields may depend on each other, so if ψ contains a reference to another model field `y`, the scope of quantification of `y` has to include ψ. Thus, the semantics of model fields is only well-defined if there exists a linear ordering of dependencies and an upper bound on their length.

Concrete Instantiations. There are two approaches by Leino and Müller [13] and Tafat et al. [15], respectively, based on ownership methodology, which is used in Spec# and ACSL among other languages. Model field values are stored on the heap, like concrete or ghost fields. In contrast to JML, they are defined not to change their value instantaneously when the locations change on which they

depend, but at given program points, namely upon invoking the special `pack` operation on its owner.

In these works, represents clauses are not allowed to contain calls to pure methods, and references to other model fields only in a few restricted cases. It is not clear whether there are restrictions on the chosen values in case the represents clause is not functional. Thus it may be possible, according to this definition, that the value of a model field spontaneously changes upon packing even though all dependencies retain their values.

Axiomatic Semantics. Weiß [16] presents a dynamic logic with explicit heap objects. Model fields are translated to function symbols. Represents clauses are introduced through logical axioms. As a simple solution to avoid an inconsistent axiom set, they are guarded by an existentially quantified assumption, which guarantees that the single represents clause is satisfiable. Mutually recursive represents clauses may, however, give rise to inconsistent axiom sets.

Frame Conditions. Much of the above is dedicated to how model fields gain their values. Another important property is to specify when a model field's value does *not change*, known as a frame condition. To this end, Weiß introduces contracts for model fields [16], similar to frame conditions on methods, which have to be respected by implementing represents clauses. Here, the argument of a frame condition is an expression of type `\locset` ('set of locations') which is dynamically valuated. This approach, known as *dynamic frames* theory, is particularly useful when the concrete fields on which a model field depends are not known on the abstract level.

7 Conclusion and Outlook

In this paper, we have presented a semantics for model fields in annotation-based specification languages. The first version is strongly inspired by the notion of ε terms as introduced by Hilbert. We have demonstrated the connection between those two concepts—one from an established theory and one as a current challenge in formal methods in software engineering. While this semantics exposes a 'good behavior' in most cases, the general case requires a different methodology. This second version covers the complete expression sub-language of JML.

To the best of our knowledge this is the first contribution in which model fields are described in all their extent and in an application-independent way. This means that the results can be applied to any verification paradigm. It also provides the basis to independently give a definition of well-definedness.

Model fields are a powerful instrument in code-level specification which hides behind the familiar syntactical guise. However, it is debatable whether there is a real demand for non-functional relations. Commonly, within the technique of abstraction, there are several concrete entities which are related to *one* abstract representation. In the vast majority of instances, there is always a sensible functional representation. In Fig. 2 for instance, we have seen an example where the

represents clause exposes a kind of weakly functional behavior, while it would not do any harm to overspecify the relation and provide values to any case which is yet left undefined.

References

1. Mike Barnett, K. Rustan M. Leino, and Wolfram Schulte. The Spec# programming system: an overview. In Gilles Barthe et al., editors, *Construction and Analysis of Safe, Secure and Interoperable Smart devices*, volume 3362 of *LNCS*, pages 49–69. Springer, 2005.
2. Patrick Baudin, Pascal Cuoq, Jean-Christophe Filliâtre, Claude Marché, Benjamin Monate, Yannick Moy, and Virgile Prevosto. *ACSL: ANSI/ISO C Specification Language, Version 1.5*, 2010.
3. Cees-Bart Breunesse and Erik Poll. Verifying JML specifications with model fields. In *Formal Techniques for Java-like Programs (FTfJP)*, number 408 in Technical Report, ETH Zurich, pages 51–60, July 2003.
4. Daniel Bruns. Formal semantics for the Java Modeling Language. Diploma thesis, Universität Karlsruhe, 2009.
5. Ádám Darvas and Peter Müller. Formal encoding of JML level 0 specifications in JIVE. Technical Report 559, ETH Zürich, 2007.
6. Christian Engel. A translation from JML to JavaDL. Studienarbeit, Fakultät für Informatik, Universität Karlsruhe, February 2005.
7. Martin Giese and Wolfgang Ahrendt. Hilbert's ε-terms in automated theorem proving. In Neil V. Murray, editor, *Automated Reasoning with Analytic Tableaux and Related Methods (TABLEAUX'99)*, volume 1617 of *LNCS*, pages 171–185. Springer, 1999.
8. David Hilbert and Paul Bernays. *Grundlagen der Mathematik*, volume II. Springer, 1939.
9. C. Anthony R. Hoare. Proof of correctness of data representations. *Acta Informatica*, 1:271–281, 1972.
10. Bart Jacobs and Erik Poll. A logic for the Java Modeling Language (JML). Technical Report CSI-R0018, University of Nijmegen, Computing Science Institute, November 2000.
11. Gary T. Leavens, Albert L. Baker, and Clyde Ruby. Preliminary design of JML: A behavioral interface specification language for Java. *ACM SIGSOFT Software Engineering Notes*, 31(3):1–38, March 2006.
12. Gary T. Leavens, Erik Poll, Curtis Clifton, Yoonsik Cheon, Clyde Ruby, David Cok, Peter Müller, Joseph Kiniry, Patrice Chalin, and Daniel M. Zimmerman. *JML Reference Manual*, July 13, 2011.
13. K. Rustan M. Leino and Peter Müller. A verification methodology for model fields. In Peter Sestoft, editor, *European Symposium on Programming (ESOP)*, volume 3924 of *Lecture Notes in Computer Science*, pages 115–130. Springer, March 2006.
14. K. Rustan M. Leino and Greg Nelson. Data abstraction and information hiding. *ACM Transactions on Programming Languages and Systems*, 24(5):491–553, September 2002.
15. Asma Tafat, Sylvain Boulmé, and Claude Marché. A refinment methodology for object-oriented programs. In Bernhard Beckert and Claude Marché, editors, *First International Conference on Formal Verification of Object-Oriented Software (FoVeOOS 2010)*, volume 6528 of *Lecture Notes in Computer Science*, pages 153–167. Springer, January 2011.

16. Benjamin Weiß. *Deductive Verification of Object-Oriented Software — Dynamic Frames, Dynamic Logic and Predicate Abstraction.* PhD thesis, Karlsruhe Institute of Technology, January 2011.