Dynamic Dispatch for Method Contracts through Abstract Predicates

Wojciech Mostowski
Formal Methods and Tools, University of Twente
The Netherlands
w.mostowski@utwente.nl

Mattias Ulbrich
Karlsruhe Institute of Technology, Karlsruhe
Germany
ulbrich@kit.edu

Abstract

Dynamic method dispatch is a core feature of object-oriented programming by which the executed implementation for a polymorphic method is only chosen at runtime. In this paper, we present a specification and verification methodology which extends the concept of dynamic dispatch to design-by-contract specifications.

The formal specification language JML has only rudimentary means for polymorphic abstraction in expressions. We promote these to fully flexible specification-only query methods called *model methods* that can, like ordinary methods, be overridden to give specifications a new semantics in subclasses in a transparent and modular fashion. Moreover, we allow them to refer to more than one program state which give us the possibility to fully abstract and encapsulate two-state specification contexts, i.e., history constraints and method postconditions.

We provide the semantics for model methods by giving a translation into a first order logic and according proof obligations. We fully implemented this framework in the KeY program verifier and successfully verified relevant examples.

Categories and Subject Descriptors D.2.1 [*Software Engineering*]: Requirements/Specification; D.2.4 [*Software Engineering*]: Software/Program Verification

Keywords Modular specification, Design by Contract, Dynamic dispatch, Abstract predicates, JML

1. Introduction

The possibility to override the implementation of a method defined in a super-type is the essential polymorphism feature of the object orientation paradigm. The mechanism which chooses at runtime the implementation to be taken for a method invocation is called *dynamic dispatch*. Also in the context of *design-by-contract* (DbC) [22] and behavioural subtyping [10], different implementations for the same operation can coexist – if they adhere to a common specification. It is most natural that not only the *implementations* but also the *specifications* vary from subtype to subtype, for instance by adding implementation-dependent aspects. The dynamic dispatch mechanism should, hence, also be available for the formulation of *formal* specifications in an equally flexible way.

For instance, the precondition of a method may be weakened in a subclass according to the principle of behavioural subtyping. When at some place in the program this method is invoked, the precondition to be established at that point depends, like the chosen implementing code, on the dynamic type of the receiver object. Instead of spelling out the definition of this specification element, it should be possible to refer to it *symbolically*. Only when the dynamic type of the object is known, one also knows the actual contract definition.

Having an explicit symbol to represent a component of a method contract also increases the modularity of the DbC methodology. A method may thus require in its contract that the precondition for a method call on one of its parameters holds. The corresponding method call on the parameter is then valid without the caller needing to know what the condition actually says. This makes specifications more modular and local since the contracts need not concern themselves with implementation details from remote classes.

In this paper, we propose a universal solution to make dynamic dispatch for specifications possible by employing multi-state abstract functions/predicates through Java Modelling Language (JML) model methods. Model methods are like usual Java methods subject to dynamic dispatch. Since they resemble normal methods in syntax and semantics, this is a most natural extension to the DbC paradigm and, hence, should be easily adoptable by programmers.

Although (one-state) model methods are also already part of the JML syntax definition [7, 17], they lack a clear semantics and are not fully and soundly implemented in any JML-based tool. We provide a precise semantics by giving an explicit logical encoding of overridable model methods in a first-order verification logic. This encoding is used in the implementation of our approach within the KeY program verifier [1]. In KeY Java programs and their JML specifications are translated to proof obligations and then proved correct by the KeY verification engine that provides a high degree of automation, and allows for user guided proof interactions where necessary. The work we present here builds on top of previous work done with KeY to support abstract specifications [27, 29].

Other verification systems have similar support for specification abstraction (see, e.g., [15, 18]), but none of them allow the specifier to refer to two or more program states in one function, i.e., functions refer to one state of the program only. In turn it is, e.g., impossible to define functions that would fully encapsulate a non-trivial relation between the pre and post state of the method. We enrich the concept of abstract predicates by allowing them to refer to more than one program state. This allows us, in particular, to use model methods to abstract and encapsulate specification contexts in which more than one program state is referred to. This is the

MODULARITY'15, March 16–19, 2015, Fort Collins, CO, USA.

ACM 978-1-4503-3249-1/15/03…$15.00.
http://dx.doi.org/10.1145/

case for postconditions or history constraints which may refer to the state before and after a method invocation.

In this context our work provides the following contributions. We define overridable model methods with strict semantics that integrate with the JML specification methodology. We provide the ability to relate several program states within one model method. In particular, *two-state* methods for pre and post program states, and *no-state* methods for defining program independent axioms and lemmas to further improve modularisation of specifications. The resulting mechanism is universal enough to be used in any specification artefact (pre- and postconditions, but also, e.g., framing clauses), and, because of the preserved dynamic dispatch principle, provides true data encapsulation on specification level relieving the specifier from enforcing exposure of private data to specifications. Furthermore, our model methods themselves are equipped with contract specifications, this in effect provides a modular lemma mechanism for the underlying abstract predicates. In the paper we discuss how model methods are integrated into the KeY Dynamic Logic and how the new specifications are translated into proof obligations. The new framework has been fully implemented in KeY and we verified relevant examples with the new implementation. All these examples were either not specifiable and verifiable at all, or extremely difficult to verify in the existing framework.

The paper is organised as follows. Section 2 briefly introduces the Java Modelling Language as implemented in KeY. In Sect. 3 we give a motivating example to demonstrate the use of model methods. The integration of model methods into dynamic logic is described in Sect. 4, Sect. 5 shortly discusses the implementation and other verification examples, and Sect. 6 concludes the paper.

2. The Java Modeling Language and an Extension

Basic JML. JML is a behavioural interface specification language realising the DbC principle [22] for the Java programming language [7, 17]. JML annotations reside in special Java comments starting with the `@` sign. The specifications define constraints for the implementation of the declarations. A program is called correct with respect to its formal specification if no run of the program can ever lead to a state which does not fulfil all specification elements.

JML can be used in different verification scenarios: (1) for runtime verification when actually running the code, (2) for static deductive functional verification in which the implementation is formally proved correct w.r.t. its specification. In this paper we concentrate on the latter application case, although the presented specification concepts can also be integrated into runtime verification concepts.

In the concept of DbC, there are essentially two constructions: *object invariants* (annotated to class declarations) and *method contracts* (annotated to method declarations). An object invariant is a predicate which defines when its object is in a valid state. Method contracts formally define the observable behaviour of methods, and are composed of one or more of the following. A `requires` clause (a.k.a. *precondition*) defines a condition under which the method may be invoked and then has the effects of the contract. An `ensures` clause (a.k.a. *postcondition)* defines a condition which holds after the execution of the method. *History constraints* are like postconditions that apply to all methods of a class. An `assignable` clause (a.k.a. *frame*) defines which part of the heap may be written to during the execution of the method. Finally, an `accessible` clause defines the part of the heap the method may read from at most. In our nomenclature we refer to it as a *dependency frame*, we explain the details in Sect. 4.

The expression language of JML is an extension of side-effect-free Java expressions. It adds a handful of specification-only constructs, most notably the first-order-logic quantifiers (`\forall`, `\exists`). Methods may be invoked in JML expressions if the method does not change existing locations on the heap; such methods are called `pure`. In history constraints and postconditions it is possible to refer to two states of program execution: the state *before* the method was invoked and the state *after* the execution. The before-state is accessed with the `\old` operator applied to an expression. Postconditions can also refer to the result of the method call with the `\result` keyword.

JML offers several other specification elements (e.g., to handle exceptions) that are not essential here, hence we omit them.

JML* – Location Sets and Observer Symbols. JML* is a recent extension of JML to work with location sets and abstract predicates. JML supports the concept of so-called *store-ref expressions* which are syntactic entities corresponding to sets of locations on the heap. Benjamin Weiß proposes in [27] and [29] to make sets of location first class citizens in JML, to follow the specification style and verification method of dynamic frames [16]. A new primitive data type `\locset` is introduced to represent sets of the locations on the heap. A location is a pair of an object reference together with a non-static field name, a pair of an array reference together with an integer index, or the name of a static field. The following set theoretic constructors are introduced: `\nothing`, $\{\!|\cdot|\!\}$, $\cdot \cup \cdot$ to resp. construct the empty set, singletons and set union. The predicate $\cdot \subseteq \cdot$ can be used to express a subset relationship. The expression $\{\!|\texttt{o.f}|\!\} \cup \{\!|\texttt{o.g}|\!\}$, for instance, denotes the two element set $\{(o, f), (o, g)\}$. Expressions of type `\locset` are typically used in `assignable` or `accessible` clauses.

To model abstractions of the program state, it is useful to add declarations to the program which exist only for the sake of specification and verification and are not visible to the compiler. JML provides two means for this purpose: (1) `ghost` declarations, which introduce new, specification-only heap locations. Their values can and must be updated explicitly by specification-only statements within the code of the methods, (2) `model` declarations which do not denote locations, but provide *abbreviations* or *abstractions* of the state. Their value is updated implicitly by changing the value of locations that the model elements *depend* on.

Model fields are used in other verification approaches (e.g., [7, 19, 29]) to abstract from implementation details. Apart from being debated about [4], model fields have obvious shortcomings. First, model fields cannot depend on any arguments, like methods do, so they are truly only state *observing* functions rather than state *querying* functions. Second, as realised in JML*, any additional properties (i.e., lemmas) of model fields are specified globally with class invariants. This destroys modularity, in that (a) the properties are not explicitly attached to a particular model field, i.e., properties of all model fields are thrown into one invariant "bag", (b) consequently, the properties of each model field often need to be reproved several times. Because of these reasons, proper specification inheritance is very limited. In this paper we show how the notion of a model field is naturally extended to a model method to remedy all of the mentioned problems. In turn, we provide a fully functional multi-state abstract predicate mechanism for modular specifications that maintain full data encapsulation of the specified program, mitigating any need to expose private data to specifications.

We continue with a motivating example, which should also explain the workings of JML* in a more accurate way. One more example is briefly discussed towards the end of the paper in Sect. 5.

3. Motivating Example

We motivate and explain our specification approach by means of a small Java example. Though small, it captures a typical and intricate situation which occurs symptomatically if object oriented

```
class Cell {
 int val;

 /*@ ensures \result == val; @*/
 int /*@ pure @*/ get() { return val; }

 /*@ ensures val == v; @*/
 void set(int v) { val = v; }
}

class Client {
 /*@ ensures c.val == v; @*/
 static void callSet(Cell c, int v) { c.set(v); }
}
```

```
class Recell extends Cell {
 int oval;

 /*@ ensures val == oval; @*/
 void undo() { val = oval; }

 /*@ ensures oval == \old(val); @*/
 void set(int v) {
   oval = val;
   super.set(v);
 }
}
```

Figure 1. `Cell`/`Recell` example.

programs are extended by classes overriding methods with additional unforeseen features. Traditionally, such situations would require that the specification of the original code be readjusted to accommodate the new behaviour. Using model methods with dynamic dispatch, such readjustment is not needed.

The challenge, shown in Fig. 1, has originally been proposed in [26] and has been dealt with in [2] using a higher order separation logic. This first figure shows the program annotated with traditional specification means. `Cell` objects encapsulate integer values which can be set using a method `set` and be retrieved using `get`. The class `Recell`, which extends `Cell`, allows an additional one level `undo` operation which restores the cell value to the state before the most recent call to `set`. The class `Client` provides a method `callSet` which indirectly calls the `set` method of the `Cell` argument it receives. This particular indirection may seem artificial, but indirection is a very natural phenomenon in object orientation, e.g., in a situation where this operation is done only conditionally or after some locks have been acquired or in combination with other operations.

The contract of `callSet` copies the postcondition of `Cell.set` literally. It does not guarantee the stronger postcondition of `Recell.set` if the argument is of type `Recell`. The present contract does not suffice to verify the following test case:

```
Recell rc = new Recell();
rc.set(4);
Client.callSet(rc, 5);
rc.undo();
assert rc.get() == 4;
```

While this program would not fail its assertion, the proof for that would not succeed as the abstraction of `callSet` by its contract neglects the additional postcondition `oval == \old(val)` introduced in `Recell` and only ensures the weaker postcondition of `Cell`.

This could be amended by introducing case distinctions on the type of the argument in the postcondition of `Cell.set`. An additional clause `c instanceof Recell ==> ((Recell)c).oval == \old(c.val)` would achieve this. However, it has significant limitations regarding the modularity of the specification: (1) Details on the implementation of `Recell` are revealed where it is not necessary and should be kept under the hood and, more severely, (2) the implementation of `Recell` might not yet be known at the time that `Cell` is implemented or specified. Assume `Cell` and `Client` are part of a library and `Recell` is a user-written extension. How can the library account for all potential extensions?

This is precisely where abstract predicates in the form of model methods can be used to solve the issue. In Fig. 2, the example has been reformulated using a model method `post_set` (lines 4–6) formalising the postcondition of the method `set` (used in line 11). The model method has a body which defines its value. In this case, it returns true if and only if its argument `x` is equal to the value stored in field `val`. Looking at class `Cell` alone, no semantic change has been done.

Things change when the class `Recell` is again added to the scenario. In `Recell`, the model method `post_set` is overridden and adds a condition to the result obtained by `Cell.post_set`. By redefining the predicate locally for all instances of class `Recell`, the semantics of the contract `Cell.set` has now also changed, although syntactically it is the same. As the contract refers to the post-condition only *symbolically*, its semantics is left open and can be redefined by an implementing class. Furthermore, `post_set` makes use of its `two_state` declaration in class `Recell` as the definition relates values from two execution states, namely `\old(get())` and `oval`. The two states that this definition refers to are the pre- and post-state of the method `set`.

The redefinition of `post_set` in `Recell` cannot be arbitrary, however. The model method has got a contract (line 4) saying that whenever its result is true, the condition `val == x` needs to hold. All overriding implementations need to obey that contract, but may add to it. This ensures behavioural subtyping.

The above example test case can be proved correct if the model method invocation `c.post_set(v)` is used as postcondition for `Client.callSet` abstracting away from the actual definition of the postcondition.

Figure 3 shows the scenario including the frame conditions where the frame has also been abstracted by a single state model method `footprint()`. Note how this method is used to specify the part of the heap on which the cell operates. The actual shape of this set of locations is different in the two classes; this can be addressed by giving the exact definition for `footprint()` in the corresponding classes. To provide global constraints on the footprint we can specify a contract for this model method. In the example we added an upper and a lower bound for the location set. Furthermore, the extended version has another model method `pre_set` used to abstract the concrete precondition of `set`. In the classes `Cell` and `Recell` this method always returns true. In a new subclass `IncreaseCell`, it returns true only if the argument `v` is greater than the cell's value `val`. The precondition is only satisfied if the value to be set is strictly increased. According to Liskov's behavioural subtyping paradigm [10, 20], this strengthening of the precondition in a subclass would not be admitted. Using model methods, however,

```
class Cell {
  int val;

  /*@ ensures \result ==> get()==x;
    @ model two_state
  boolean post_set(int x) { return val == x; } @*/

  /*@ ensures \result == val; @*/
  int /*@ pure @*/ get() { return val; }

  /*@ ensures post_set(v); @*/
  void set(int v) { val = v; }
}
```

```
class Recell extends Cell {
  int oval;

  /*@ model two_state
  boolean post_set(int x) { return
      super.post_set(x) && oval == \old(get());
  } @*/

  /*@ ensures get() == \old(oval); @*/
  void undo() { val = oval; }

  void set(int x) { oval = get(); super.set(x); }
}
```

Figure 2. Cell/Recell example annotated with model methods.

```
class Cell {
  int val;

  /*@ accessible \nothing;
    @ ensures \result ⊆ this.* &&
        {|this.val|} ⊆ \result;
    model \locset footprint() {
        return {|this.val|}; } @*/

  /*@ accessible footprint();
    @ ensures \result ==> get()==x;
    @ model two_state
    boolean post_set(int x) {
        return get() == x; } @*/

  /*@ accessible footprint();
    @ model
    boolean pre_set(int x) { return true; } @*/

  /*@ accessible footprint();
    @ ensures \result == val; @*/
  int /*@ pure @*/ get() { return val; }

  /*@ requires pre_set(v);
    @ ensures post_set(v);
    @ assignable footprint(); @*/
  void set(int v) { val = v; }
}
```

```
class Recell extends Cell {
  int oval;

  /*@ model \locset footprint() {
    return {|this.val|} ∪ {|this.oval|};
  } @*/

  /*@ model two_state
  boolean post_set(int x) { return
      super.post_set(x) && oval == \old(get());
  } @*/

  /*@ ensures get() == \old(oval);
    @ assignable footprint(); @*/
  void undo() { val = oval; }

  void set(int x) { oval = get(); super.set(x); }
}


class IncreaseCell extends Cell {
  /*@ model
  boolean pre_set(int v) {
    return v > val;
  } @*/
}
```

Figure 3. Extended Cell/Recell example annotated with footprint specifications.

one can specify such strengthened preconditions without violating the principle since both methods describe the situation locally for their respective enclosing class. The method contract in Line 4 in Fig. 2 says what the postcondition `post_set` must at least imply, it can thus not be weakened arbitrarily. No contract has been set up for `pre_set` such that no restriction exists for the implementation of `pre_set` in subclasses. Model method specifications allow the specifier to choose flexibly how the implementations of different classes relate to each other – depending on the need of the verification scenario.

4. Translation into Java Dynamic Logic

To verify a Java program, we translate the program and its specification into proof obligations in Java Dynamic Logic (JavaDL from now on), the logic of the KeY theorem prover. JavaDL is a hierarchically typed first order logic in which the type system contains the reference types of the Java language (`java.lang.Object` and its subtypes) together with the types *Int*[1], *Bool*, *Heap*, *LocSet* and *Field*. Every type is subtype of the top type *Any*.

We write $f : T_1 \times \ldots \times T_n \to T$ to denote a function symbol mapping n elements of types $T_1, \ldots, T_n$ to an element of type T. We write $s : T$ if s is constant symbol or a logical variable symbol. For an n-ary predicate symbol p, we write $p : T_1 \times \ldots \times T_n$ to denote that it represents a relation on these types. JavaDL uses the standard first-order operators $\neg, \to, \wedge, \vee, \forall, \exists$, and $\leftrightarrow$ for, respectively, negation, implication, conjunction, disjunction, universal and existential quantification, and equivalence.

[1] All numeric primitive Java types `int`, `long`, ... are mapped to *Int*. We do not support floating point types.

JML E	**JavaDL** $\widehat{E}$
`this`, `\result`	$self$, $result$
`\old(`E`)`	$\{h := h_0\}\widehat{E}$
E`.field`$_{C,T}$	$select_T(h, \widehat{E}, \mathsf{C}::\mathsf{field})$
E`.query`$_{C,T}$`(`F_1`,...,`F_n`)`	$\mathsf{C}::\mathsf{query}(h, \widehat{E}, \widehat{F_1}, \ldots, \widehat{F_n})$
E`.twostate`$_{C,T}$`(`F_1`,...,`F_n`)`	$\mathsf{C}::\mathsf{twostate}(h, h_0, \widehat{E}, \widehat{F_1}, \ldots, \widehat{F_n})$

Figure 4. Translation of JML expressions into JavaDL, `name`$_{C,T}$ refers to the entity of type T introduced in class C, `query`$_{C,T}$ is a one-state, `twostate`$_{C,T}$ a two-state model method.

Besides the standard first-order operators, JavaDL provides, for every type T, the membership predicate symbols $\cdot \sqsubseteq T : \mathit{Any} \to \mathit{Bool}$ and $\cdot \sqsubseteq_! T : \mathit{Any} \to \mathit{Bool}$. The formula $t \sqsubseteq T$ is true if the value of the expression t is of type T or of one of its subtypes; the formula $t \sqsubseteq_! T$ is true if the value of t is of type T but not of any strict subtype of T. Thus, $t \sqsubseteq T$ in JavaDL is closely related to the expression `t instanceof T` in Java and $t \sqsubseteq_! T$ to `t.getClass() == T.class`.

JavaDL is a dynamic logic [13] in which Java program code can be used to construct formulas. For a Java code fragment π and a formula φ, the composition $[\pi]\varphi$ is again a formula which holds in a state iff φ holds in the corresponding end state after the execution of π (if it exists)[2]. The substitution of a term s for a (program) variable x in a term t is denoted by $\{x := s\}t$ in which the type of expression s must be a subtype of the type of x.

In JavaDL, the Java heap memory is modelled using the type *Heap* implementing the *theory of arrays* [21]. The elements of *Heap* are the possible memory states of the program. Two heap objects are relevant for the evaluation of JML expressions: The symbol $h : \mathit{Heap}$ holds the current heap state, i.e., the memory at the current execution point, and variable $h_0 : \mathit{Heap}$ refers to the *base heap* of the current method frame, i.e., to the memory in the pre-state of the method call. We assume heaps to be two-dimensional arrays with one index of type *Object* and the other of type *Field* capturing all fields appearing in the program. Every declaration of a member (field, (model) method) m in class C gives rise to a function symbol named $\mathsf{C}::\mathsf{m}$; in case of a field f this is $\mathsf{C}::\mathsf{f} : \mathit{Field}$. In case of a (model) method m, a function symbol $\mathsf{C}::\mathsf{m}$ is introduced which takes the heap as explicit argument. We call such a symbol *observer function symbol* since it gives a value which depends on the heap context, but without itself residing in a location on the heap.

Reading from a location on the heap is done using a family of function symbols $select_T : \mathit{Heap} \times \mathit{Object} \times \mathit{Field} \to T$ for every type T. Two additional variables *self* and *result* exist for the translation of the `this` reference and the result value of the method. Their types depend on the proof obligation context. Fig. 4 shows a synopsis of the translation of the most important JML expressions E to their respective counterpart $\widehat{E}$ in JavaDL.

4.1 Model methods

In JML, expressions can also refer to regular or model methods as long as they are declared pure. Unlike fields that declare locations on the heap, methods do not reside in locations on the heap but compute a value which *depends* on the values of locations on the heap. Model methods are always automatically considered pure since they are meant to observe the heap without changing it. If a JML model method and its contract are defined according to the following general schema (all clauses in `[...]` are optional)

```
class C {
  /*@ [requires pre;]
    @ [ensures post;]
    @ [accessible acc;]
    @ [measured_by mby;]
    @ [two_state] model R m(T_1 p_1, ..., T_n p_n)
    @       { return exp; }
    @*/
}
```

then the function symbol $\mathsf{C}::\mathsf{m} : \mathit{Heap} \times \mathit{Heap} \times C \times T_1 \times \ldots \times T_n \to R$ is introduced to represent the model method in JavaDL. The symbol takes the current heap, the base heap, the receiver object and the method parameters as arguments. The second heap argument is used only if the method is annotated with the modifier `two_state`. For model methods declared without the `two_state` modifier, the second quantification over h_0 and the second heap argument to $\mathsf{C}::\mathsf{m}$ must be dropped in the formulas in this section. The semantics of the symbol is coupled to the expression in the `return` statement by the following definition axiom.

$$\forall h, h_0 : \mathit{Heap}, \mathit{self} : C, p_1 : T_1, \ldots, p_n : T_n; \\ (\mathit{self} \sqsubseteq_! C \wedge \widehat{\mathit{pre}} \to \\ \mathsf{C}::\mathsf{m}(h, h_0, \mathit{self}, p_1, \ldots, p_n) = \widehat{\mathit{exp}}) \,. \quad (1)$$

The function symbol $\mathsf{C}::\mathsf{m}$ is determined by the class (or interface) C in which the method m has been first declared. All method definitions overriding that initial declaration refer to the same function symbol (and not to a new symbol). By constraining the same function, they realise the dynamic dispatch of model methods. That is, the function symbol is always the same, while its meaning implied by the exact type of *self* changes. For any subclass C' of C, another axiom for $\mathsf{C}::\mathsf{m}$ is added. If C' chooses not to override m, an axiom is added as if the definition with the body of the superclass-method had been repeated. The guard $\mathit{self} \sqsubseteq_! C$ (respectively, $\mathit{self} \sqsubseteq_! C'$ in the axiom for C') ensures that the definition only applies if the receiver object *self* is *exactly* of the defining type. These typing guards make sure that (possibly contradicting) definitions of the function $\mathsf{C}::\mathsf{m}$ constrain different parts of its domain.

The axiom is also guarded by the precondition $\widehat{\mathit{pre}}$ of the contract. It is not strictly necessary to restrict the domain in which $\mathsf{C}::\mathsf{m}$ can be applied but we decided that it is better to allow a specifier to say under which conditions a model method is defined. Also to deal with welldefinedness and wellfoundedness (see Sect. 4.3), it is important to be able to restrict the definitions to arguments for which they make sense.

Our model method body (see above) consists only of a single side-effect-free `return` statement and definition (1) can make use of its expression directly. If a one-state model method *did* have a non-trivial method body, the above axiom would need to involve a dynamic logic operator and read

$$\forall h : \mathit{Heap}, \mathit{self} : C, p_1 : T_1, \ldots, p_n : T_n; (\mathit{self} \sqsubseteq_! C \wedge \widehat{\mathit{pre}} \to \\ \left[\texttt{result} = \texttt{self.m}(p_1, ..., p_n);\right] \\ (\mathsf{C}::\mathsf{m}(h, \mathit{self}, p_1, \ldots, p_n) = \mathit{result})) \,,$$

ensuring that the value of the function symbol is the same as the result value of the method call. This formula points out a crucial advantage of dynamic logic in comparison to other program logics like *wp*-calculus or Hoare calculus: Dynamic logic is closed under all its operators which allows us to state the quantified program

[2] $[\pi]\varphi$ is thus semantically equivalent to $wlp(\pi, \varphi)$ of the weakest precondition calculus [11].

formula directly in JavaDL, and not on a meta-level. The dynamic logic thus also allows us to seamlessly extend the presented approach to non-model queries.

Besides its method body, a model method may also have a functional contract (in particular a postcondition). Unlike the body which *defines* the value of the function symbol, the contract *describes a property* of the symbol and is not an axiom, but a theorem. To establish the correctness of the contract theorem, it suffices to prove that the definition makes the postcondition true, i.e., that

$$\begin{aligned}&\forall h, h_0 : \mathit{Heap}, \mathit{self} : C, p_1 : T_1, \ldots, p_n : T_n;\\ &\quad (\mathit{self} \sqsubseteq_! C \wedge \widehat{\mathit{pre}} \rightarrow\\ &\qquad \{\mathit{result} := \mathsf{C{::}m}(h, h_0, c, p_1, \ldots, p_n)\}\widehat{\mathit{post}}) \ . \end{aligned} \quad (2)$$

follows from axiom (1). If (2) is shown for every class C' extending C (with a corresponding type guard), the statement is shown for all conceivable instances of C. Therefore, when using the proved contract as additional assumption, it is safe to omit the type guard $\mathit{self} \sqsubseteq_! C$ from (2). This approach is still modular, however: The verification of C happens independently of that of its subclasses. At the time of verification, one can even be oblivious to the existence of subtypes.

The two-state model method `post_set` from the motivating example in Fig. 2 is translated into a function symbol $\mathsf{Cell{::}post\_set} : \mathit{Heap} \times \mathit{Heap} \times \mathit{Cell} \times \mathit{Int} \rightarrow \mathit{Bool}$ which is constrained as follows:

$$\begin{aligned}&\forall h, h_0 : \mathit{Heap}, c : \mathit{Cell}, v : \mathit{Int};\\ &\quad (c \sqsubseteq_! \mathit{Cell} \rightarrow \qquad (3)\\ &\qquad (\mathsf{Cell{::}post\_set}(h, h_0, c, v) \leftrightarrow \mathsf{Cell{::}get}(h, c) = v))\\ &\wedge\ (c \sqsubseteq_! \mathit{Recell} \rightarrow\\ &\qquad (\mathsf{Cell{::}post\_set}(h, h_0, c, v) \leftrightarrow (\mathsf{Cell{::}get}(h, c) = v \wedge \qquad (4)\\ &\qquad\quad \mathit{select}_{\mathit{Int}}(h, c, \mathsf{Recell{::}oval}) = \mathsf{Cell{::}get}(h_0, c))))\\ &\wedge\ (\mathsf{Cell{::}post\_set}(h, h_0, c, v) \rightarrow \mathsf{Cell{::}get}(h, c) = v) \qquad (5)\end{aligned}$$

True is taken as precondition for `post_set` as none has been explicitly specified. Constraints (3), (4) are model method definitions according to (1); (5) is the contract theorem after (2). It is obvious that both implementations in `Cell` and `Recell` imply this contract.

4.2 Framing

For the sake of a modular proof, the user of an observer symbol should not have access to its definition (i.e., method body). It may be wise to hide it from the user (for encapsulation) or its full definition may even not be known in a modular context in which additional classes may be added later. In order to reason about observers without access to their definitions, it is crucial that one is able to deduce that an observer does not change if only heap locations that it does not depend on have been modified.

An observer function modelling a model method takes heap argument(s). However, its valuation depends not on the entire heap(s) but only on a set of locations on that heap(s). If it can be established that this location set is interpreted equally in two heaps, the model method must result in the same value in both states.

To this end, we make use of the `accessible` clause that a model method may declare. In it, a set of locations acc can be specified describing the locations upon which the evaluation of a method can depend *at most*: if all memory locations in $\widehat{\mathit{acc}}$ have the same value in both heaps, then the values returned by the model method must be the same.

Such knowledge is essential for modular verification and data encapsulation: at verification time, the implementation of a (model) method may not be known (a situation very common in programming against interfaces), but restrictions of the set of accessible locations may be part of the contract. Knowing that an evaluation does not depend on recent changes on the heap lifts both specifications and proofs to a higher level of abstraction, i.e., the equality of expressions can be established without knowing their exact definitions.

The fact that two heaps h_1, h_2 evaluate the locations in $\widehat{\mathit{acc}}$ equally is formally expressed by the formula $\mathit{same}(h_1, h_2, \widehat{\mathit{acc}})$ defined by

$$\begin{aligned}&\mathit{same}(h_1, h_2, s) := \forall o : \mathit{Object}, f : \mathit{Field};\\ &\quad (\{(o, f)\} \subseteq \{h := h_1\}s) \rightarrow\\ &\qquad\qquad \mathit{select}_{\mathit{Any}}(h_1, o, f) = \mathit{select}_{\mathit{Any}}(h_2, o, f) \ .\end{aligned}$$

This yields the following conditional equality for model methods (in which $\widehat{\mathit{pre}}_i$ and $\widehat{\mathit{acc}}_i$ abbreviate $\{h := h_i\}\widehat{\mathit{pre}}$ and $\{h := h_i\}\widehat{\mathit{acc}}$, respectively):

$$\begin{aligned}&\forall h_1, h_2, h_1', h_2' : \mathit{Heap}, \mathit{self} : C, p_1 : T_1, \ldots, p_n : T_n;\\ &\quad (\mathit{self} \sqsubseteq_! C \wedge \widehat{\mathit{pre}}_1 \wedge \widehat{\mathit{pre}}_2 \wedge\\ &\qquad \mathit{same}(h_1, h_2, \widehat{\mathit{acc}}_1) \wedge \mathit{same}(h_1', h_2', \widehat{\mathit{acc}}_{1'}) \rightarrow\\ &\qquad\quad \mathsf{C{::}m}(h_1, h_1', \mathit{self}, p_1, \ldots, p_n) =\\ &\qquad\qquad\qquad \mathsf{C{::}m}(h_2, h_2', \mathit{self}, p_1, \ldots, p_n)) \ . \end{aligned} \quad (6)$$

The evaluations of the query symbol must be shown equal under the conditions that the (1) two pre-state heaps h_1 and h_2 satisfy $\widehat{\mathit{pre}}$, (2) they evaluate equivalently on the set $\widehat{\mathit{acc}}$, and (3) two post-state heaps h_1' and h_2' also evaluate equivalently on $\widehat{\mathit{acc}}$. Like in the case of the functional method contract in (2), an additional type guard $\mathit{self} \sqsubseteq_! C$ may be used when proving the *dependency contract* for class C. The stronger version without the type guard may be used when adding (6) as an assumption in other proofs.

The one-state method `get` in Fig. 3 has `accessible` clause `this.footprint()`, its concrete one-state instance of (6) therefore has only one pair of heap variables and reads (after simplification):

$$\begin{aligned}&\forall h_1, h_2 : \mathit{Heap}, c : \mathit{Cell};\\ &\quad (\mathit{same}(h_1, h_2, \mathsf{Cell{::}footprint(heap, c)}) \rightarrow\\ &\qquad\qquad \mathsf{Cell{::}get}(h_1, c) = \mathsf{Cell{::}get}(h_2, c)) \end{aligned} \quad (7)$$

For *exact* instances of *Cell*, the footprint is defined as the singleton set containing `this.val`, the *same* predicate in (7) is thus equivalent to $\mathit{select}_{\mathit{Any}}(h_1, c, \mathsf{Cell{::}val}) = \mathit{select}_{\mathit{Any}}(h_2, c, \mathsf{Cell{::}val})$ in that case.

4.3 Termination

Showing termination for programs is optional, considering the partial correctness problem alone can be challenging enough. For the definition of model methods, however, it is a central point that must not be omitted. A model method definition gives rise to a universally quantified axiom claiming that the function has certain properties even if it may be unsatisfiable. Consider for instance the problematic declaration

```
class X {
  /*@ model int bad() {
    @   return this.bad() + 1;
    @ }
    @*/
}
```

for which the model method would be translated into the axiom

$$\begin{aligned}&\forall h : \mathit{Heap}, \mathit{self} : X; (\mathit{self} \sqsubseteq_! X \rightarrow\\ &\qquad\qquad \mathsf{X{::}bad}(h, \mathit{self}) = \mathsf{X{::}bad}(h, \mathit{self}) + 1) \ ,\end{aligned}$$

```
/*@ requires \disjoint(footprintUntilLeft(t), t.footprint());
  @ ensures \result ==> t.left == null || leftSubTree(t.left);
  @ ensures \result ==> footprint() == footprintUntilLeft(t) ∪ t.footprint(); ...
  @ accessible footprintUntilLeft(t);
  @ measured_by height;
  @ model boolean leftSubTree(Tree t) { return t == this || (left != null && left.leftSubTree(t)); }
  @*/
```

Figure 5. An excerpt of the solution to the `Tree` challenge from [5].

which is obviously inconsistent. Consistency can be guaranteed if termination (or *wellfoundedness*) of all recursive method references is checked. To this end, an additional proof obligation per model method is generated to ensure this. Here, the `measured_by` clauses are employed to avoid such unsatisfiable recursive definitions. We require that all definitions are primitive recursive. The variant mby specifies for each method a termination measurement which must be decreased in all referenced (model) method invocations in exp.

5. Implementation

The model methods have been fully implemented in the current official KeY release 2.2. The translation in the actual implementation is more sophisticated than in our presentation. In particular, the implementation also accounts for the wellformedness of heap expressions, possible `null`-references, exceptions, object creation, etc. A lot of the effort has gone into extending the JML* parser to accept the new syntax. On the level of the prover engine, the previously existing support for model fields and parameter-less observer symbols was extended to support model methods. In particular, the KeY data structures were changed to allow for the observer symbols to take additional heap arguments as well as formal parameter arguments. Consequently, the generation of the corresponding proof rules and proof obligations changed accordingly. The only really new thing in terms of the implementation are the contract rules that allow the use of model method specifications as lemmas.

The `Cell` example that we have used in this paper is part of the KeY distribution, along with other small examples. All of these examples are proven correct fully automatically by KeY. We did, however, also test our implementation on a more complex example, namely a binary tree deletion of minimal element challenge from the VerifyThis 2012 verification competition. Although this challenge does not, e.g., require the use of two-state model methods or in fact even inheritance, the solution that uses model methods is far more elegant than all the other solutions we have (unsuccessfully) tried previously. The challenge and our solution are described in full in [5], here we only present the culprit of our solution.

The competition challenge is about verifying an iterative procedure for removing the minimal element from a binary search tree with an obvious recursive linked data structure representation. That is, the class `Tree` declares field `val` for the node value, and two fields, `left` and `right`, linking the left and the right subtrees, with `null` denoting the leaves of the tree. The essence of our solution to the challenge is the `leftSubTree(Tree t)` model method, which by recursion checks whether the given tree `t` is one of the sub-tree nodes reachable through following the `left` links from the current node. Through specifying an appropriate method contract for `leftSubTree`, this method is delegated to maintain a set of crucial facts about the binary tree structure. For example, if the tree `t` is in fact a node somewhere in the path of `left` links, then we know that so is `t.left` (if not null) or that we can partition the current tree footprint between all the tree nodes before `t` is reached and the footprint of `t` itself. These footprints are again defined with model methods. Figure 5 shows an excerpt of the specification for `leftSubTree`. The reminder of the solution is to maintain the validity of the `leftSubTree` predicate while the tree is being iterated by the algorithm to find the node to be removed. By the specification of `leftSubTree` this removal is guaranteed to only change a small part of the tree and keep the overall binary search tree structure intact. Because of the use of several model methods that are mutually recursive, this challenge is only provable interactively by the current version of KeY, nevertheless, model methods were the enabling factor to solve the challenge at all.

We also apply our specification method to the verification of concurrent Java programs using permission accounting. In permission accounting verification approaches programs are annotated with permission expressions to guard every memory location access. A full permission grants a write access, while a partial permission grants only a read access. The construction of the verification method and the permission splitting and recombining system guarantee that verified programs are data-race free. In our work we developed a new symbolic permission system that is an alternative to the classical fractional systems [3] and we plugged it into KeY verification logic through the use of a new additional permission heap [23]. This new heap integrates seamlessly into our model methods specification methodology which we use to encapsulate permission specifications for concurrent Java programs. With this encapsulation we achieve specification and verification capabilities similar to resource invariants [24] and concurrent abstract predicates [12, 14] commonly used in permission based verification methods. The current development version of KeY now includes a working support for permission based verification with a handful of examples specified using (also two-state) model methods.

6. Related Work and Conclusion

Parkinson and Bierman have presented in [26] a separation logic framework for programs with inheritance improving on their earlier paper [25]. One of the declared goals and actual achievements of the paper was to avoid repetition of proofs of a method unless it has been overridden. Our motivating example presented in Sect. 3 has been taken from this publication. The approach put forward in [26] rests on the differentiation of *static* versus *dynamic* specifications. Dynamic specifications correspond in our solution to contracts for model methods, while static specifications are reflected in the definitions of model methods. Another contribution of [26] is the extension of the concept of an abstract predicate to abstract predicate families in their (higher order) specification logic. The publication [2] also uses the example from [26]. The authors present an approach in higher-order logic separation logic, formalised in Coq, that allows even more powerful quantifications than can be achieved with abstract predicate families. Their proofs are interactive. Cok presents in [8] a survey on using model fields and methods in JML specifications. Overriding of model fields is covered, but they are not used as a means to abstract away from method contracts. The Dafny language and verification system [18] uses functions to define abstraction predicates which correspond to our model methods. The Dafny language does not support subtyping,

nor does it support two-state predicates. In [6], the authors abstract from contract components by predicates to decouple deductive reasoning about programs from the applicability check of contracts. This increases the reuse potential of proofs in case of changing specifications and, thus, makes the program verification process more modular. However, subtyping between predicates is not considered. Darvas [9] covers the issue of queries in specifications in a logical setting similar to ours. In particular, welldefinedness and wellfoundedness are treated in depth. We employ techniques from his thesis to conduct termination proofs in our approach. The approach does not distinguish between contracts (postconditions) and definitions (method bodies) and needs a satisfiability check for the specifications. An advantage of our approach is that *satisfiability* of the model method description does not need to be shown as the return statement always gives an explicit witness to the value of the method. Inheritance is not considered in [9].

The first occurrence of calculus rules for dynamic dispatch of regular method invocations in program verification is by Soundarajan and Fridella [28], the concept has since been introduced into many verification tools for languages with polymorphism.

Conclusion We have presented a specification style which uses overridable model methods to abstract away from concrete method specifications and, hence, allows us to refer to them symbolically. Our specifications give rise to proof obligations with dynamic frames as presented in [27, 29]. We go a step further than [27, 29] by enabling fully flexible state queries with parameters and by providing means for two-state specifications and a fully modular lemma mechanism for model methods via contracts. The main contribution of this work is that model methods make the concept of modular method specification more flexible. Dynamic method dispatch gives model methods a semantics that depends on the typing context. Behavioural subtyping [10] can canonically be preserved using method contracts, but the approach is more flexible and also allows for contract relationships that do not adhere to behavioural subtyping.

Acknowledgments

We are very grateful to Peter H. Schmitt for his constructive suggestions and valuable comments which helped improve this paper and to Jesper Bengtson for pointing out this problem to us and ensuing discussions. The work of Wojciech Mostowski is supported by ERC grant 258405 for the VerCors project. The work of Mattias Ulbrich is supported by the DFG (German Research Foundation) under the Priority Programme SPP1593: Design For Future – Managed Software Evolution.

References

[1] B. Beckert, R. Hähnle, and P. H. Schmitt, editors. *Verification of Object-Oriented Software: The KeY Approach*, volume 4334 of *LNCS*. Springer-Verlag, 2007.

[2] J. Bengtson, J. B. Jensen, F. Sieczkowski, and L. Birkedal. Verifying object-oriented programs with higher-order separation logic in Coq. In *Proceedings of ITP*, 2011.

[3] J. Boyland. Checking interference with fractional permissions. In *Static Analysis Symposium*, volume 2694 of *LNCS*, pages 55–72. Springer, 2003.

[4] C.-B. Breunesse and E. Poll. Verifying JML specifications with model fields. In *Formal Techniques for Java-like Programs. Proceedings of the ECOOP'2003 Workshop*, 2003.

[5] D. Bruns, W. Mostowski, and M. Ulbrich. Implementation-level verification of algorithms with KeY. *Software Tools for Technology Transfer*, 2013. On-line first, to appear.

[6] R. Bubel, R. Hähnle, and M. Pelevina. Fully abstract operation contracts. In *Proceedings of ISoLA*, volume 8803 of *LNCS*, pages 120–134. Springer, 2014.

[7] P. Chalin, J. R. Kiniry, G. T. Leavens, and E. Poll. Beyond assertions: Advanced specification and verification with JML and ESC/Java2. In *Proceedings of FMCO*, volume 4111 of *LNCS*, pages 342–363. Springer-Verlag, 2006.

[8] D. R. Cok. Reasoning with specifications containing method calls and model fields. *Journal of Object Technology*, 2005.

[9] Á. Darvas. *Reasoning About Data Abstraction in Contract Languages*. PhD thesis, ETH Zurich, 2008.

[10] K. K. Dhara and G. T. Leavens. Forcing behavioral subtyping through specification inheritance. In *Proceedings of ICSE*, pages 258–267. IEEE Computer Society, 1996.

[11] E. W. Dijkstra. *A Discipline of Programming*. Prentice Hall, Inc., 1976. ISBN 013215871X.

[12] T. Dinsdale-Young, M. Dodds, P. Gardner, M. J. Parkinson, and V. Vafeiadis. Concurrent abstract predicates. In T. D'Hondt, editor, *ECOOP 2010 – Object-Oriented Programming*, volume 6183 of *LNCS*, pages 504–528. Springer, 2010.

[13] D. Harel, D. Kozen, and J. Tiuryn. *Dynamic Logic*. MIT Press, 2000.

[14] B. Jacobs and F. Piessens. Expressive modular fine-grained concurrency specification. *SIGPLAN Not.*, 46(1):271–282, January 2011.

[15] B. Jacobs, J. Smans, P. Philippaerts, F. Vogels, W. Penninckx, and F. Piessens. VeriFast: A powerful, sound, predictable, fast verifier for C and Java. In *NASA Formal Methods*, volume 6617 of *LNCS*, pages 41–55. Springer, 2011.

[16] I. Kassios. The dynamic frames theory. *Formal Aspects of Computing*, 23, 2011.

[17] G. T. Leavens, E. Poll, C. Clifton, Y. Cheon, C. Ruby, D. Cok, P. Müller, J. Kiniry, P. Chalin, D. M. Zimmerman, and W. Dietl. JML Reference Manual, 2008.

[18] K. R. M. Leino. Dafny: An automatic program verifier for functional correctness. In *Proceedings of LPAR-16*, volume 6355 of *LNCS*, pages 348–370. Springer, 2010.

[19] K. R. M. Leino and P. Müller. A verification methodology for model fields. In *Proceedings of ESOP*, pages 115–130, 2006.

[20] B. Liskov and J. M. Wing. Specifications and their use in defining subtypes. In A. Paepcke, editor, *Proceedings of OOPSLA*, pages 16–28, Washington DC, USA, 1993. ACM Press.

[21] J. McCarthy. Towards a mathematical science of computation. In *Information Processing 1962*, pages 21–28, 1963.

[22] B. Meyer. Applying "design by contract". *Computer*, 25(10), 1992.

[23] W. Mostowski. A case study in formal verification using multiple explicit heaps. In *Proceedings of FORTE/FMOODS*, volume 7892 of *LNCS*, pages 20–34. Springer, 2013.

[24] P. W. O'Hearn. Resources, concurrency and local reasoning. *Theoretical Computer Science*, 375(1–3):271–307, 2007.

[25] M. J. Parkinson and G. M. Bierman. Separation logic and abstraction. In *Proceedings of POPL*, 2005.

[26] M. J. Parkinson and G. M. Bierman. Separation logic, abstraction and inheritance. In *Proceedings of POPL*, 2008.

[27] P. H. Schmitt, M. Ulbrich, and B. Weiß. Dynamic frames in Java dynamic logic. In *Post-Proceedings of FoVeOOS*, volume 6528 of *LNCS*, pages 138–152. Springer, 2011.

[28] N. Soundarajan and S. Fridella. Reasoning about polymorphic behavior. In *Proceedings of TOOLS*, pages 346–358. IEEE Computer Society, 1998.

[29] B. Weiß. Predicate abstraction in a program logic calculus. *Science of Computer Programming*, 76(10):861–876, 2011.