Dynamic Dispatch for Method Contracts Through Abstract Predicates

Wojciech Mostowski[1](✉) and Mattias Ulbrich[2]

[1] Centre for Research on Embedded System, Halmstad University, Halmstad, Sweden
wojciech.mostowski@hh.se

[2] 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. Finally, we provide an elegant and flexible mechanism to specify restrictions on specifications in subtypes. Thus behavioural subtyping can be enforced, yet it still allows for other specification paradigms.

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. We have also implemented an extension to KeY to support permission-based verification of concurrent Java programs. In this context model methods provide a modular specification method to treat code synchronisation through API methods.

1 Introduction

The possibility to override the implementation of a method defined in a supertype 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) [34] and behavioural subtyping [16], 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

© The Author(s) 2016
S. Chiba et al. (Eds.): ToMC I, LNCS 9800, pp. 238–267, 2016.
DOI: 10.1007/978-3-319-46969-0_7

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 external 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 already part of the JML syntax definition [11,28], 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 [2,4]. 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 [45,47].

Other verification systems have similar support for specification abstraction (see, e.g., [24,29]), 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 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. Finally, within the scope of such lemma annotations, we provide a flexible mechanism to specify the relationship between supertype and subtype predicate implementations. Thus one can enforce specification paradigms like behavioural subtyping [32] and others, and gains liberty to integrate such schemes within one program. 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 previously existing framework.

We have also applied model methods to support modular verification of concurrent Java programs specified with permission annotations [22,38]. In particular, we have used model methods to provide fully modular and reusable specifications for Java API synchronisation methods. In this paper we show an example of how we achieve this.

This paper is an extended and revised version of our paper presented at the Modularity conference held in Fort Collins, U.S. in March 2015 [39]. In particular, Sects. 5 and 7 are new in this version, the former discusses how we can specify behavioural subtyping (or other schemes) in our specification approach, the latter discusses two use cases for model methods using substantial examples. Furthermore, at its end, Sect. 4 now discusses in more detail how termination clauses for recursive model methods are checked. Finally, in Sect. 6 we now also briefly discuss the verification performance gains that can be achieved with abstract specifications. The complete 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, while Sect. 5 discusses our mechanism to enforce behavioural subtyping in specifications. Sect. 6 shortly discusses some implementation practicalities and mentions one more small example. Section 7 presents two more uses cases of providing generic API specifications with model methods, Sect. 8 reports on related work, and finally Sect. 9 concludes the paper.

2 The Java Modeling Language and an Extension

2.1 Basic JML

JML is a behavioural interface specification language realising the DbC principle [34] for the Java programming language [11,28]. 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 with respect to its specification. In this paper we concentrate on the latter application case, although the presented specification concepts can also be integrated into runtime verification methods.

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.

2.2 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 [45,47] to make sets of location first class citizens in JML, to follow the specification style and verification method of dynamic frames [25]. A new primitive data type **`\locset`** is introduced to represent sets of 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.

JML* employs location sets to model relevant segments of the heap. However, if other approaches that model heap structures differently by means of other abstract datatypes (like, e.g., in the flexible framework presented in [26]) are at the base of a specification language, model methods could also be used to abstract from them.

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 which are not visible to the compiler. JML provides two means for this purpose: (1) **`ghost`** field 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`** field 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. Such state abstractions can be in particular used to abstract **`assignable`** and **`accessible`** frames mentioned above. In JML* both ghost and model fields can be used for this purpose. While abstracting location sets through ghost variables tends to make reasoning easier than with model fields, the latter provide a cleaner specification style. With model elements, the specifier needs not explicitly update the specification state of the verified programs (which on the other hand helps the verification engine to find proofs). In our examples and explanations to follow we also show how abstractions of locations sets are achieved with model methods and what are the corresponding issues for behavioural subtyping. Finally, despite that they can be used for similar purpose, ghost and model fields are in principle independent concepts as far as the reasoning logic is concerned. Ghost fields have successfully been used for the specification and verification of dynamic frames within the Dafny verifier [29].

Model fields [12] are used in other verification approaches (e.g., [11,30,47]) to abstract from implementation details. Apart from being debated about [8], 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

```
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);
 }
}
```

Fig. 1. `Cell`/`Recell` example.

are thrown into one invariant “bag”, (b) consequently, the properties of each model field often need to be re-proved 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. More examples are briefly discussed towards the end of the paper.

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

The challenge, shown in Fig. 1, has originally been proposed in [43] and has been dealt with in [5] 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 setPost (lines 4–6) formalising the postcondition of the method set (used in line 15). 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 setPost is overridden and adds a condition to the result obtained by Cell.setPost. 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, setPost makes use of its **two_state** declaration in class Recell as the definition relates values from two execution

```
   class Cell {
2   int val;

4   /*@ ensures \result ==> get()==x;
      @ model two_state
6   boolean setPost(int x) {
      return val == x;
8   } @*/

10  /*@ ensures \result == val; @*/
    int /*@ pure @*/ get() {
12    return val;
    }
14
    /*@ ensures setPost(v); @*/
16  void set(int v) { val = v; }
   }
```

```
class Recell extends Cell {
  int oval;

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

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

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

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

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 setPost 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.setPost(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(). Furthermore, the extended version has another model method setPre 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 [16,21,32], this strengthening of the precondition in a subclass would not be admitted. Using model methods, however, 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 setPost must at least imply, it can thus not be weakened arbitrarily. No contract has been set up for setPre such that no restriction exists for the implementation of setPre 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.

Method frames are also subject to behavioural subtyping. The footprint() model method is declared and defined once in the superclass Cell to contain all

```
class Cell {
  int val;

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

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


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

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

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

```
class Recell extends Cell {
  int oval;

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

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


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


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

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

locations of the `Cell` object. In the context of the `Cell` class alone the postcondition we have specified for `footprint()` may seem obvious and redundant. However, it limits the shape of the footprint for the sub-classes with an upper bound, and this particular postcondition enforces frame related behavioural subtyping on the methods that refer to this model method in their assignable clauses. Namely, the footprint of the sub-classing objects can only shrink or stay the same, but it cannot grow beyond the extension of the super class. In this example, the (syntactical) upper bound of **`this.*`** covers the footprints of all classes in Fig. 3, however, as with other specifications, the subtyping requirement for frames can be limiting. We elaborate on this in Sect. 5, where we discuss how the user can specify the relationship between of supertype and subtype implementations of model methods with certain flexibility. If strict behavioural subtyping is required (according to [32]), it can be achieved, but it is not mandatory in our framework.

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 [2]. 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 $\neg, \rightarrow, \wedge, \vee, \forall, \exists$, and $\leftrightarrow$ for, respectively, negation, implication, conjunction, disjunction, universal and existential quantification, and equivalence.

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 [20] 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* [33]. 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 $\mathsf{C{::}m}$; in case of a field f this is $\mathsf{C{::}f} : \mathit{Field}$. In case of a (model) method m, a function symbol $\mathsf{C{::}m}$ 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 $\mathit{select}_T : \mathit{Heap} \times \mathit{Object} \times \mathit{Field} \to T$ 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.

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

[2] $[\pi]\varphi$ is thus semantically equivalent to $\mathit{wlp}(\pi, \varphi)$ of the weakest precondition calculus [17].

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

Fig. 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.

Figure 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 *strictly* pure since they are meant to observe the heap without changing it. "Strictly" means that not even object creation is allowed that is normally considered pure behaviour. The reason for also excluding object creation is the non-determinism it would introduce. If a JML model method and its contract are defined according to the following general schema (all clauses in [...] are optional)

```
class C {
  /*@ [private] behavior
    @   [requires pre;]
    @   [ensures post;]
    @   [accessible acc, [acc′];]
    @   [measured_by mby;]
    @ [two_state] model R m(T1 p1, ..., Tn pn) { return exp; }
    @*/
}
```

then the function symbol $\mathsf{C::m} : \mathit{Heap} \times \mathit{Heap} \times C \times T_1 \times \ldots \times T_n \to R$ 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**. The second argument to the **accessible** clause is also only relevant if **two_state** is specified; we come back to the declaration of the clause shortly in Sect. 4.2. For model methods declared without the **two_state** modifier, the second quantification over h_0 and the second heap argument to

C::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.

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

The function symbol C::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 C::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 copied. 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 C::m constrain different parts of its domain. Finally, even though the axiom is constrained to *exact* instances of a class C, for all subclasses of C the axiom is repeated, either by looking up the model method definition freshly introduced in a subclass, or by copying the definition from the superclass when the model method is not overridden.

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 C::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. To extend the framework to methods with non-trivial method bodies, the above axiom would need to involve a dynamic logic operator and read (for a one-state model method)

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

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 Hoare logic: dynamic logic is closed under all its operators which allows us to state the quantified program formula directly in JavaDL. That is, the program modality is not required to be a top level operator, it appears as a sub-formula under the quantifier. In Hoare calculus this would require using meta-constructs over the program correctness triplet.

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

$$\forall h, h_0 : \mathit{Heap}, \mathit{self} : C, p_1 : T_1, \ldots, p_n : T_n;\\ (\mathit{self} \sqsubseteq_! C \land \widehat{\mathit{pre}} \to \{\mathit{result} := \mathsf{C{::}m}(h, h_0, c, p_1, \ldots, p_n)\}\widehat{\mathit{post}}). \quad (2)$$

follows from axiom (1).[3]

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. If the contract is annotated with the Java modifier **private**,[4] it applies to class C only and is not to be inherited to subclasses. Hence, the proof obligation and theorem are concerned only with exact instances of the class.

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

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

True is taken as precondition for `setPost` as none has been explicitly specified. Constraints (3) and (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.

[3] One also has to show that the method's framing condition specified with the **assignable** clause holds. Recall that model methods are assumed to be strictly pure, so it would be required to additionally prove that no locations on either of the two heaps h and h_0 are changed or created. This is trivial to show when we only allow single side-effect-free **return** statements for model method bodies, so we omit this here for clarity. However, our implementation does include this check in the anticipation of also accepting model methods with non-trivial bodies. In Sect. 5 we explain more about **assignable** frame conditions for regular Java methods in the context of behavioural subtyping.

[4] The modifier is applied to the *contract*, not the model method declaration itself.

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[5] 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 $\widehat{acc}$ have the same value in both heaps, then the values returned by the model method must be the same. In JavaDL, every location is a pair of an object and a field. This subsumes array references and static field references: in case of arrays the field part is obtained from the *Int* value representing the index into the array, for static field references a fixed dummy object reference is used.

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 $\widehat{acc}$ equally is formally expressed by the formula $\mathit{same}(h_1, h_2, \widehat{acc})$ defined by

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

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

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

[5] We also call it the *dependability* location set, while other approaches may have still different name for it, e.g., *influence* set in [31] that essentially serves the same purpose of providing the read frame of a method.

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 $\widehat{pre}$, (2) they evaluate equivalently on the set $\widehat{acc}$, and (3) two post-state heaps h'_1 and h'_2 also evaluate equivalently on $\widehat{acc}'$. [6] Like in the case of the functional method contract in (2), an additional type guard $self \sqsubseteq_! C$ 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):

$$\forall h_1, h_2 : Heap, c\colon Cell; (same(h_1, h_2, \mathsf{Cell::footprint}(\mathsf{heap}, \mathsf{c})) \rightarrow \\ \mathsf{Cell::get}(h_1, c) = \mathsf{Cell::get}(h_2, c)). \quad (7)$$

For *exact* instances of *Cell*, the footprint is defined as the set containing all object fields (**`this.*`**), with `val` being the only field the *same* predicate in (7) is thus equivalent to $select_{Any}(h_1, c, \mathsf{Cell::val}) = select_{Any}(h_2, c, \mathsf{Cell::val})$ 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

$$\forall h : Heap, self : X; (self \sqsubseteq_! X \rightarrow \mathsf{X::bad}(h, self) = \mathsf{X::bad}(h, self) + 1),$$

which is obviously inconsistent. Consistency can be guaranteed if termination (or *wellfoundedness*) of all recursive method references is checked. 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*. To this end, an additional proof obligation per model method is generated to ensure this. Assuming that the variant of a model method referenced in *exp* is mby', it has to be shown that mby' is a strict non-negative predecessor of mby, i.e., $0 \leq mby' < mby$.

[6] Note that there are two separate accessible sets, $\widehat{acc}$ for the pre-state heap and $\widehat{acc}'$ for the post-state heap, as these can be specified separately according to the schema in Sect. 4.1. If only the post-state heap is referenced in the model method definition, then only one accessible clause *acc* is specified with the other one assumed empty (like for model method `setPost` in Fig. 3).

In practice, one may also encounter mutually recursive definitions of model methods. In this case simple integer expressions as termination clauses are in general not sufficient. For that reason, we additionally allow tuples of integer expressions with a standard lexicographic order to serve as termination clauses and the above mechanism is modified accordingly to check the lexicographic ordering of the expressions instead. Furthermore, to weaken the resulting proof obligations, we use the welldefinedness checking technique described in [14] exploiting the logical structure of the expression.

5 Support for Behavioural Subtyping

Behavioural subtyping is a built-in feature of JML: Reasoning modularly in the face of extensible class hierarchies is impossible if the substitution principle does not apply. In terms of contracts, this principle means that every (non-private) method contract introduced in a class must be obeyed by all extending subclasses as well.

Additional contracts may be specified in subclasses giving new use cases for a method in this subclass. Hence, by adding new contracts, one can weaken the overall precondition or strengthen the postcondition. It is, however, not possible to make the precondition stronger or the postcondition weaker. Frame conditions specified through the **assignable** clauses are technically part of the postcondition, as they are checked with the following formula attached to a postcondition:

$$\forall o : \mathit{Object}, f : \mathit{Field}; (\{(o, f)\} \subseteq \widehat{\mathit{frame}} \vee \mathit{select}_{\mathit{Any}}(h, o, f) = \mathit{select}_{\mathit{Any}}(h', o, f)),$$

where $\widehat{\mathit{frame}}$ is translation of the clause **assignable** `frame;` of the method under inspection and h with h' are the method's pre-state and post-state heaps, respectively. As a postcondition, the frame condition cannot be weakened either meaning that the frame cannot *grow* in subclasses.

Programmers employ inheritance not only to implement behavioural subtyping [35], but also for purposes like, e.g., code reuse or specialisation. Hence, a more flexible way to deal with the inheritance of specification clauses is desired which allows more liberal notions of inheritance. In particular, in the context of framing subclasses are likely to extend the footprint of the superclass to provide additional functionality, rather than shrinking it. For example, in our `Recell` class in Fig. 3 (page 9) a new field `oval` is added.

By using model methods one is not bound to behavioural subtyping in the same rigorous fashion as with pure JML. In addition to the possibility to name specification artefacts (possibly without knowing their meaning), nothing is a priori said about the relationship of an abstract predicate definition in one class and its redefinition in a subclass. These may, in general, be unrelated, i.e., without logical consequence relationship in either way.

In this unrestricted fashion, model methods would be too arbitrary, and in most practical cases one wants to establish a formal relationship between different realisations of the same predicate. To this end, we introduced new keywords with which consequences can be expressed concisely.

The special postcondition `\covariant` can be used to specify that any reimplementation a model method must at least guarantee what is guaranteed in this implementation. For example, the method `postCondition` specified as

```
/*@ model_behaviour
      ensures \covariant;
      model boolean postCondition() { return R; } @*/
```

requires all redefinitions to imply R, for instance by returning R in conjunction with more information. The identifier `\covariant` is an abbreviation for the expression `\result ==>` R. Correspondingly, a method can be specified to be `\contravariant` by which the condition R `==>` `\result` is abbreviated. For location sets model methods expressing frames the two keywords express subset relation instead of implication, so, e.g., `\covariant` is an abbreviation for `\result` $\subseteq R$.

With these new annotations, it is possible to stipulate behavioural subtyping. A generic abstract method specification using model methods and behavioural subtyping would hence read:

```
/*@ ensures \contravariant;
  @ model abstract boolean preCondition();
  @
  @ ensures \covariant;
  @ model abstract two_state boolean postCondition();
  @
  @ ensures \covariant;
  @ model abstract \locset footprint(); @*/

/*@ normal_behaviour
  @ requires preCondition();
  @ ensures postCondition();
  @ assignable footprint(); @*/
void method() { ... }
```

Programming languages other than Java have different notions of behavioural subtyping. In the Eiffel [36] programming language, for instance, preconditions are – like postconditions – *co*variant. Using the model method framework, it is perfectly possible to argue and reason about such constructions also within Java and JML. With treating variance by annotations, one is more flexible and can easily choose between different paradigms within the same program. Finally, in [21] several other notions of behavioural subtyping are surveyed with different *grading* in the context of object orientation and class invariants are discussed.

6 Model Methods in Practice

The model methods have been fully implemented in the current official KeY release 2.4. The translation in the actual implementation is more sophisticated

```
/*@ 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(Treet){
     return t == this || (left!= null && left.leftSubTree(t));
    }@*/
```

Fig. 5. An excerpt of the solution to the `Tree` challenge from [9].

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. That is, we had to implement generation of proof rules representing formula (2) for every model method specified with a contract rule. The implementation of all the other formulas was an extension and adoption of existing rules for model fields.

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. One particular example that we used as a test bed when implementing model methods is 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 [9], here we only present the essence 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 recursive and mutually dependent, 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.

Apart from increased expressiveness, model methods also allow one to optimise the verification effort in terms of the size of the underlying proof, especially in the context of an evolving specification when earlier proof attempts can be reused to prove subsequent versions of the program correct. This is because model methods allow us to reason without looking into definitions, at least up to a certain point. In other words, some part the proof is independent of the specification and consequently can be reused. The actual effort in a repeated proof attempt starts only when the model method definitions are expanded and the proof has to be guided according to these (possibly updated) definitions. Earlier studies [10] showed that the corresponding gains can be as large as halving the proof effort when one compares a situation of applying proof reuse enabled by using abstract specifications (in our case model methods) to a situation when no proof reuse is possible due to fully concrete specifications that require a complete new proof with each new version of the specification.

7 Example Use Cases

In this section we describe two somewhat more elaborate use cases for model methods. The first one is the well known visitor design pattern, the second one sits in the context of permission-based reasoning about concurrent programs, a capability recently added to KeY. In both examples model methods are crucial to provide generic, client independent specifications of library methods that can be later instantiated for a particular case.

7.1 Specifying the Visitor Design Pattern

The visitor design pattern [19] is particularly well suited to be specified and verified using abstract predicates as devised in this paper.

The situation is thus: A library defines a class hierarchy in which a number of classes implement a common interface `Component`. This interface requires a method `accept(Visitor v)`. The respective subclasses that implement the interface realise this acceptance method by delegating the call to a method specially crafted for this particular subclass within the visitor. By this extra indirection, the visitor pattern thus allows dynamic method dispatch by the type of its argument (and not of the receiver).

```
interface Component {
 /*@ public normal_behaviour
     requires \invariant_for(v);
     requires v.preVisit(this);
     ensures \invariant_for(v);
     ensures v.postVisit(this);
     assignable v.modVisit(this); @*/
 void accept(Visitor v);
}
```

```
class CompA extends Component {
  int val;
  void accept(Visitor v) { v.visitA(this);}
}

class CompB extends Component {
  String val;
  void accept(Visitor v) { v.visitB(this);}
}
```

```
interface Visitor {
  /*@ ensures \contravariant;
      ensures \result ==> \invariant_for(c);
      model boolean preVisit(Component c)
           ↪{return false;}

      ensures \covariant;
      ensures \result ==> \invariant_for(c);
      model two_state boolean postVisit
           ↪(Component c) { return true;}

      ensures \covariant;
      model \locset modVisit(Component c)
           ↪{return \everything;}@*/
```

```
  /*@ public normal_behaviour
    @ requires preVisit(a);
    @ ensures postVisit(a);
    @ assignable modVisit(a);
    @*/
  void visitA(CompA a);

  /*@ public normal_behaviour
    @ requires preVisit(b);
    @ ensures postVisit(b);
    @ assignable modVisit(b);
    @*/
  void visitB(CompB b);
}
```

Fig. 6. Generic library specifications for the visitor pattern.

```
class LenVisitor implements Visitor {
  int len;

  /*@ model boolean preVisit(Component c) {
        return (len >= 0 && \invariant_for(c)); }
      model boolean postVisit(Component c) {
        return (len >= \old(len) && \invariant_for(c)); }
      model \locset modVisit(Component c) {return this.*;}@*/

  void visitA(CompA a){len += Math.abs(a.val);}
  void visitB(CompB b){len += b.val.length();}
}

//@ requires \invariant_for(a) && \invariant_for(b);
static void test(CompA a, CompB b) {
  LenVisitor lv = new LenVisitor();
  a.accept(lv); b.accept(lv);
  assert lv.len >= 0;
}
```

Fig. 7. Using the visitor pattern.

The use case (the actual visitor implementation) of the pattern is usually not known at the time the library is defined. Moreover, often there is not only one visitor, but an application requires multiple different visitor implementations to perform different (not a-priori known) tasks on the data structure.

It is evident that the component hierarchy and the visitor interface cannot be specified concretely with all use cases in mind. Even if they were known a priori, incorporating all possible visitation purposes into one specification would make that clumsy, unnecessarily large, and non-scalable. That is where model methods can be used to an advantage: The visitor interface and the component type hierarchy can be specified with model methods in contracts to leave the details of the specification open and to be refined in the implementations while the library specification stays agnostic to these implementations.

Figure 6 shows the general specification setup. The interface `Component` is the base type for all components open for visitation. The JML contract of method `accept` uses the model methods `preVisit` and `postVisit` in the pre- and postcondition. Two classes `CompA` and `CompB` implement the interface and inherit the contract for their delegating implementations of the `accept` method. `CompA` wraps an integer value while `CompB` contains a `String` value.

The visitor interface defines model methods `preVisit`, `postVisit` and `modVisit` to abstract away from the precondition, postcondition, and the frame clause of the `visit` methods. The concrete visitation methods use these model methods in their abstract contracts and thus remain ignorant of the purpose of later introduced visitors without loss of precision. To enforce the behavioural subtyping principle for the possible implementations of the visitor interface, the three model methods are equipped with subtyping enforcing contracts as stipulated in Sect. 5. Furthermore, the definitions of these three model methods are made most general in each case to allow for arbitrary subclass modification as far as behavioural subtyping is concerned. That is, the **`false`** precondition can always be made weaker, while the **`true`** postcondition and **`\everything`** frame can always be made stronger.

However, the framework of components and visitor must fit together. There are proof obligations to be discharged showing that the `accept` methods fulfil the interface's contracts. Since they just replicate the visitor's contract as their own, these conditions are easily proved.

As a concrete example use case of the visitor pattern, Fig. 7 shows a visitor which accumulates the lengths of `Components`. The implementation named `LenVisitor` stores the accumulated length in a field named `len`. For `CompA` the absolute value of the integer field is added to `len` and for `CompB` the length of the string is added. In the test case, it is to be proved that the length field of the concrete visitor is always positive.

Note that the specification for the visitor interface and the component types did not have to be changed for the specification of the use case. This is the essence of using abstract predicates to enable modular design and to reduce the specification effort. However, it does not necessarily imply a substantial reduction of the proof effort. As noted at the end of Sect. 6, the use of abstract predicates can support proof reuse, but does not in general mean that proofs can be skipped. In particular, if we were to subclass the LenVisitor class in some simple way without modifying either of the model methods or the visit methods, we would still have to reverify the visit methods in the new subclass. The reason is that these methods, remaining syntactically the same, may still call other methods declared in the LenVisitor class that can get new implementations in the subclass of LenVisitor. This reproving can be avoided by proving contracts not for exact instances a concrete class but for instances of all subclasses as well. Such stronger proof obligations are more difficult to prove, however, since they must not depend on overridable definitions from the declared class. The bottom line is though, that after introducing a new subclass, no superclass has to be ever re-specified or re-proved to be correct.

7.2 Permission-Based Reasoning with Model Methods

Permission-based reasoning is an established method for verifying concurrent programs with the design-by-contract style specifications enriched with *fractional* permission annotations [7]. In the context of concurrent execution, permission annotations protect access to heap locations for each thread, adherence to these annotations is verified locally for each thread and primarily establishes non-interference of threads; programs are guaranteed to be free of data races. Additionally, programs are also verified to satisfy classical DbC functional properties, however, the scope of such properties is limited; heap locations potentially modified by other threads can never be relied upon in the context of the local thread [6].

The main complication in permission-based reasoning are program synchronisation points, e.g., mutual exclusion locks or spawned threads. Permissions are then transferred between one thread and the lock, or between multiple threads. In most approaches synchronisation is generally considered to be a primitive operation of the programming language and the verification logic is equipped to handle the corresponding program constructs. In particular, in Separation Logic like approaches [44] the notion of a resource invariant (or monitor) serves as a primary means to specify the behaviour of (primitive) locks [41]. Using a *would-be* Separation Logic-like JML notation, Fig. 8 shows a simple example of a Java program with **synchronized** block protecting a single variable counter for multiple thread use. The **monitor** keyword specifies the necessary resource invariant Perm(**this**.c, 1) – on synchronisation the currently running thread temporarily receives a full (read and write) permission on the counter variable granting the thread the right to change the associated memory location. As stated above, no functional specification can be given for **this**.c outside of the **synchronized** block, in particular in the contract of increase, due to the lack of a suitable permission.

The **synchronized** block is a very basic synchronisation method and by itself provides only a mutual exclusion locking. Supporting more synchronisers, e.g., a semaphore for shared read access, requires combining **synchronized** with additional Java code. To this end, the Java API offers a synchronisation

```
public class Counter {
  private int c; //@ monitor Perm(this.c, 1);

  public void increase() { synchronized(this) { this.c++; } }
}
```

Fig. 8. An example resource invariant with a permission annotation.

framework, the `java.util.concurrent` library [27], that provides a collection of synchronisation mechanisms implemented with more primitive operations, like compare-and-swap or volatile variables. For most of those API methods, resource invariants do not scale directly and need extensions. For example, for a read-write lock the invariant differs depending on the actual lock mode (read or write). Ultimately, the library methods should have generic specifications that the clients can instantiate to achieve a particular data protection for multiple thread access, and model methods provide just the right mechanism to achieve this.

Supporting permission-based reasoning in the KeY system is achieved by operating on two heaps simultaneously in JavaDL [37] – the regular memory heap like before, and an additional permission heap that tracks permissions to the corresponding heap locations. These two heaps are integrated into the logic, but they are also made explicit in JML* specifications to allow the separation of functional properties (actual values on the heap) and non-interference properties (permissions to heap locations). Our definitions from Sect. 4 extend naturally to account for the additional permission heap, it is handled in a similar fashion as the pre- and post-heap to support the **`two_state`** predicates.

In this context, Fig. 9 shows a modular, reusable specification of a lock, which is then instantiated to protect a shared counter. Interfaces `LockSpec` and `Lock` are our library-level specifications, the `Counter` class is the client. `LockSpec` is simply a specification template for the lock, and in fact it could have been in its entirety a *model interface*, but technically it could not yet be supported by KeY. In `LockSpec` four methods are declared that clients should instantiate and the `consistent` method is defined that ensures the consistency of client provided specifications. The framing of regular and permission heaps is encapsulated with further model methods, however, for clarity the figure shows only one of them (`fpPerm`).

The `state` predicate of the lock defines the state of the permissions on the heap for when the lock is engaged and when it is released. The client defines the lock state over just one permission to the `val` field of the `Counter` class. The lock state is expressed using a symbolic permission data type [22] denoted in `[[...]]`. When not engaged, the lock holds the full permission to the `val` field, when locked the permission is temporarily transferred from the lock object to the currently running thread **`\ct`**. The `status` predicate provides an abstraction of the lock state – directly stated read and/or write permissions that the lock grants, here a write permission to the `val` field. The predicate also serves

```
public interface LockSpec {
  //@ accessible<permissions> fpPerm(); ...; model \locset fpPerm();
  //@ accessible...; model boolean state(boolean locked);
  //@ accessible...; model boolean status(boolean locked);
  //@ accessible...; model two_state boolean lockTr();
  //@ accessible...; model two_state boolean unlockTr();
  /*@ ensures \result;
      accessible...;
      model final two_state boolean consistent() { return
        (\old(state(false)) && \old(status(false)) && lockTr() ==>
           (state(true) && status(true))) &&
        (\old(state(true)) && \old(status(true)) && unlockTr() ==>
           (state(false) && status(false))); } @*/ }

public interface Lock {
  //@ public instance ghost LockSpec spec;

  //@ requires spec.status(false);
  //@ ensures spec.status(true) && spec.lockTr();
  //@ assignable<permissions> spec.fpPerm(); ...
  public void lock();

  //@ requires spec.status(true);
  //@ ensures spec.status(false) && spec.unlockTr();
  //@ assignable<permissions> spec.fpPerm(); ...
  public void unlock();
}

public class Counter implements LockSpec {
  private int val;
  private Lock lock; //@ invariant lock.spec == this && ...;

  /*@ model boolean state(boolean locked) {return \perm(val) ==
       locked ? [[ \ct, lock ]] : [[ lock ]]; } @*/
  /*@ model boolean status(boolean locked) { return locked ?
      \writePerm(\perm(val)) : !\readPerm(\perm(val)); } @*/
  /*@ model two_state boolean lockTr() { return \perm(val) ==
      \transferPermAll(lock, \ct, \old(\perm(val))); } @*/
  /*@ model two_state boolean unlockTr() { return \perm(val) ==
      \returnPerm(\ct, lock, \old(\perm(val))); } @*/

  //@ requires status(false); ensures status(false);
  //@ assignable<permissions> fpPerm(); ...
  public void increase() { lock.lock(); val++; lock.unlock(); }
}
```

Fig. 9. An example of a generic specification of a mutual exclusion lock and the corresponding client instantiation (strongly abbreviated for clarity, full version of the example is available with the development version of the KeY system).

as a *token* that indicates whether the lock is engaged or not. The transfer predicates `lockTr` and `unlockTr` define the permission transfers that occur on lock engaging and releasing. They are **`two_state`** because they describe the relation between the permissions before and after the call to the `lock` and `unlock` methods of the lock. The client defines the transfer functions to fully transfer the permission to `val` between the lock and the current thread. Through its simple postcondition, the `consistent` predicate ensures that there is consistency between the above methods. Namely, the application of the transfer function `lockTr` to an unlocked state and status of the lock should result in a locked one, and vice versa. Then, the `fpPerm` predicate defines the frame of the permission heap that the specification works with – only the permissions to the `val` field are changed. The specifications of the lock itself merely points to the predicates defined in the `LockSpec` interface. The binding between the client instantiated lock specification and the lock is provided by the `spec` ghost field of `Lock` – the client refers itself as holding the specification. Finally, the top-level specification for the `increase` method simply states that it should be used only when the lock is not already engaged. The verification of `increase` establishes that all concurrent accesses to the `val` field are non-interfering.

Our specification does not enforce the lock to be a mutual exclusion or a shared one, this distinction is only made by the client. Particular API instances of the `Lock` interface would have such distinction embedded in the implementation, like the `ReentrantReadWriteLock` that provides two sub-locks for the shared and exclusive access. A fully configurable specification that can be related to particular implementations of the lock, like presented in [3], would require further extensions of our specification shown here. Still, the example we presented here is a typical use case for model methods to provide generic, client independent specifications for the purpose of reasoning about data non-interference.

8 Related Work

Parkinson and Bierman have presented in [43] a separation logic framework for programs with inheritance improving on their earlier paper [42]. 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 [43] 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 [43] is the extension of the concept of an abstract predicate to abstract predicate families in their (higher order) specification logic. The publication [5] also uses the example from [43]. 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. The specification and verification framework for Eiffel function delegates presented by Nordio et al. [40] allows reasoning about programs using function objects by means of fixed abstract specification predicates for pre-/post- and frame-conditions. In comparison, in our approach there are no predefined predicates or place holders for particular specification elements, we simply provide a flexible mechanism to define, overload, and use abstract predicates anywhere in the specification and allow for arbitrary composition of these predicates, and additionally we provide a flexible mechanism to optionally enforce behavioural subtyping. Both approaches, however, allow for the use of two-state predicates.

Cok presents in [13] 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 [29] uses functions to define abstraction predicates which correspond to our model methods. Until recently the Dafny language did not support subtyping [1], and it does not support two-state predicates. In [10], 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 [14,15] 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 and two-state predicates are not considered by Darvas.

The first occurrence of calculus rules for dynamic dispatch of regular method invocations in program verification is by Soundarajan [46], the concept has since been introduced into many verification tools for languages with polymorphism.

In the context of concurrent reasoning using permissions we mentioned the mechanism of resource invariants commonly used in Separation Logic based approaches [43]. This encapsulation idea has been later developed towards the notion of concurrent abstract predicates, now used in at least two Separation Logic formalisms [18,23]. Concurrent abstract predicates provide essentially the same functionality for Separation Logic as model methods in our work, namely they enable data and specification encapsulation and generic library specifications that can be instantiated by clients.

9 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 [45,47]. We go a step further than [45,47] 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 and in some cases the verification performance can be improved. Dynamic method dispatch gives model methods a semantics that depends on the typing context. Behavioural subtyping [16] 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. In cases where behavioural subtyping or another scheme of specification inheritance should be preserved we provided a flexible mechanism that can enforce that.

Acknowledgements. 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 and by the Swedish Knowledge Foundation grant for the project AUTO-CAAS. The work of Mattias Ulbrich is supported by the DFG (German Research Foundation) under the Priority Programme SPP1593: Design For Future – Managed Software Evolution.

Open Access. This chapter is distributed under the terms of the Creative Commons Attribution 4.0 International License (http://creativecommons.org/licenses/by/4.0/), which permits use, duplication, adaptation, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, a link is provided to the Creative Commons license and any changes made are indicated.

The images or other third party material in this chapter are included in the work's Creative Commons license, unless indicated otherwise in the credit line; if such material is not included in the work's Creative Commons license and the respective action is not permitted by statutory regulation, users will need to obtain permission from the license holder to duplicate, adapt or reproduce the material.

References

1. Ahmadi, R., Leino, K.R.M., Nummenmaa, J.: Automatic verification of Dafny programs with traits. In: Proceedings of the 17th Workshop on Formal Techniques for Java-Like Programs (FTfJP), pp. 4:1–4:5. ACM (2015)
2. Ahrendt, W., Beckert, B., Bruns, D., Bubel, R., Gladisch, C., Grebing, S., Hähnle, R., Hentschel, M., Herda, M., Klebanov, V., Mostowski, W., Scheben, C., Schmitt, P.H., Ulbrich, M.: The KeY platform for verification and analysis of Java programs. In: Giannakopoulou, D., Kroening, D. (eds.) VSTTE 2014. LNCS, vol. 8471, pp. 55–71. Springer, Heidelberg (2014). doi:10.1007/978-3-319-12154-3_4
3. Amighi, A., Blom, S., Huisman, M., Mostowski, W., Zaharieva-Stojanovski, M.: Formal specifications for Java's synchronisation classes. In: Lafuente, A.L., Tuosto, E. (eds.) 22nd Euromicro International Conference on Parallel, Distributed, and Network-Based Processing, pp. 725–733. IEEE Computer Society (2014)
4. Beckert, B., Hähnle, R., Schmitt, P.H. (eds.): Verification of Object-Oriented Software: The KeY Approach. LNCS, vol. 4334. Springer, Heidelberg (2007). doi:10.1007/978-3-540-69061-0

5. Bengtson, J., Jensen, J.B., Sieczkowski, F., Birkedal, L.: Verifying object-oriented programs with higher-order separation logic in Coq. In: Eekelen, M., Geuvers, H., Schmaltz, J., Wiedijk, F. (eds.) ITP 2011. LNCS, vol. 6898, pp. 22–38. Springer, Heidelberg (2011). doi:10.1007/978-3-642-22863-6_5
6. Blom, S., Huisman, M., Zaharieva-Stojanovski, M.: History-based verification of functional behaviour of concurrent programs. In: Calinescu, R., Rumpe, B. (eds.) SEFM 2015. LNCS, vol. 9276, pp. 84–98. Springer, Heidelberg (2015). doi:10.1007/978-3-319-22969-0_6
7. Boyland, J.: Checking interference with fractional permissions. In: Cousot, R. (ed.) SAS 2003. LNCS, vol. 2694, pp. 55–72. Springer, Heidelberg (2003). doi:10.1007/3-540-44898-5_4
8. Eisenbach, S., Leavens, G.T., Müller, P., Poetzsch-Heffter, A., Poll, E.: Formal techniques for Java-like programs. In: Buschmann, F., Buchmann, A.P., Cilia, M.A. (eds.) ECOOP 2003. LNCS, vol. 3013, pp. 62–71. Springer, Heidelberg (2004). doi:10.1007/978-3-540-25934-3_7
9. Bruns, D., Mostowski, W., Ulbrich, M.: Implementation-level verification of algorithms with KeY. Int. J. Softw. Tools Technol. Transfer **17**, 729–744 (2015)
10. Bubel, R., Hähnle, R., Pelevina, M.: Fully abstract operation contracts. In: Margaria, T., Steffen, B. (eds.) ISoLA 2014. LNCS, vol. 8803, pp. 120–134. Springer, Heidelberg (2014). doi:10.1007/978-3-662-45231-8_9
11. Chalin, P., Kiniry, J.R., Leavens, G.T., Poll, E.: Beyond assertions: advanced specification and verification with JML and ESC/Java2. In: Boer, F.S., Bonsangue, M.M., Graf, S., Roever, W.-P. (eds.) FMCO 2005. LNCS, vol. 4111, pp. 342–363. Springer, Heidelberg (2006). doi:10.1007/11804192_16
12. Cheon, Y., Leavens, G., Sitaraman, M., Edwards, S.: Model variables: cleanly supporting abstraction in design by contract. Softw. Pract. Exp. **35**(6), 583–599 (2005)
13. Cok, D.R.: Reasoning with specifications containing method calls and model fields. J. Object Technol. **4**, 77–103 (2005)
14. Darvas, Á.: Reasoning About Data Abstraction in Contract Languages. Ph.D. thesis, ETH Zurich (2008)
15. Darvas, Á., Leino, K.R.M.: Practical reasoning about invocations and implementations of pure methods. In: Dwyer, M.B., Lopes, A. (eds.) FASE 2007. LNCS, vol. 4422, pp. 336–351. Springer, Heidelberg (2007). doi:10.1007/978-3-540-71289-3_26
16. Dhara, K.K., Leavens, G.T.: Forcing behavioral subtyping through specification inheritance. In: Proceedings of ICSE, pp. 258–267. IEEE Computer Society (1996)
17. Dijkstra, E.W.: A Discipline of Programming. Prentice Hall, Inc., Englewood Cliffs (1976)
18. Dinsdale-Young, T., Dodds, M., Gardner, P., Parkinson, M.J., Vafeiadis, V.: Concurrent Abstract Predicates. In: D'Hondt, T. (ed.) ECOOP 2010. LNCS, vol. 6183, pp. 504–528. Springer, Heidelberg (2010). doi:10.1007/978-3-642-14107-2_24
19. Gamma, E., Helm, R., Johnson, R.E., Vlissides, J.: Design Patterns, Elements of Reusable Object-Oriented Software. Addison-Wesley, Reading (1999)
20. Harel, D., Kozen, D., Tiuryn, J.: Dynamic Logic. MIT Press, Cambridge (2000)
21. Hatcliff, J., Leavens, G.T., Leino, K.R.M., Müller, P., Parkinson, M.: Behavioral interface specification languages. ACM Comput. Surv. **44**(3), 16:1–16:58 (2012)
22. Huisman, M., Mostowski, W.: A symbolic approach to permission accounting for concurrent reasoning. In: 14th International Symposium on Parallel and Distributed Computing (ISPDC 2015), pp. 165–174. IEEE Computer Society (2015)
23. Jacobs, B., Piessens, F.: Expressive modular fine-grained concurrency specification. SIGPLAN Not. **46**(1), 271–282 (2011)

24. Jacobs, B., Smans, J., Philippaerts, P., Vogels, F., Penninckx, W., Piessens, F.: VeriFast: a powerful, sound, predictable, fast verifier for C and Java. In: Bobaru, M., Havelund, K., Holzmann, G.J., Joshi, R. (eds.) NFM 2011. LNCS, vol. 6617, pp. 41–55. Springer, Heidelberg (2011). doi:10.1007/978-3-642-20398-5_4
25. Kassios, I.: The dynamic frames theory. Formal Aspects Comput. **23**, 267–288 (2011)
26. Kulczycki, G., Smith, H., Harton, H., Sitaraman, M., Ogden, W.F., Hollingsworth, J.E.: The location linking concept: a basis for verification of code using pointers. In: Joshi, R., Müller, P., Podelski, A. (eds.) VSTTE 2012. LNCS, vol. 7152, pp. 34–49. Springer, Heidelberg (2012). doi:10.1007/978-3-642-27705-4_4
27. Lea, D.: The java.util.concurrent synchronizer framework. Sci. Comput. Program. **58**(3), 293–309 (2005)
28. Leavens, G.T., Poll, E., Clifton, C., Cheon, Y., Ruby, C., Cok, D., Müller, P., Kiniry, J., Chalin, P., Zimmerman, D.M., Dietl, W.: JML Reference Manual (2008)
29. Leino, K.R.M.: Dafny: an automatic program verifier for functional correctness. In: Clarke, E.M., Voronkov, A. (eds.) LPAR 2010. LNCS (LNAI), vol. 6355, pp. 348–370. Springer, Heidelberg (2010). doi:10.1007/978-3-642-17511-4_20
30. Leino, K.R.M., Müller, P.: A verification methodology for model fields. In: Sestoft, P. (ed.) ESOP 2006. LNCS, vol. 3924, pp. 115–130. Springer, Heidelberg (2006). doi:10.1007/11693024_9
31. Leino, K.R.M., Müller, P.: Verification of equivalent-results methods. In: Drossopoulou, S. (ed.) ESOP 2008. LNCS, vol. 4960, pp. 307–321. Springer, Heidelberg (2008). doi:10.1007/978-3-540-78739-6_24
32. Liskov, B., Wing, J.M.: Specifications and their use in defining subtypes. In: Paepcke, A. (ed.) Proceedings of OOPSLA, Washington DC, USA, pp. 16–28. ACM Press (1993)
33. McCarthy, J.: Towards a mathematical science of computation. Inf. Process. **1962**, 21–28 (1963)
34. Meyer, B.: Applying "design by contract". Computer **25**(10), 40–51 (1992)
35. Meyer, B.: The many faces of inheritance: a taxonomy of taxonomy. IEEE Comput. **29**(5), 105–108 (1996)
36. Meyer, B.: Object-oriented Software Construction, 2nd edn. Prentice-Hall, Upper Saddle River (1997)
37. Mostowski, W.: A case study in formal verification using multiple explicit heaps. In: Beyer, D., Boreale, M. (eds.) FMOODS/FORTE -2013. LNCS, vol. 7892, pp. 20–34. Springer, Heidelberg (2013). doi:10.1007/978-3-642-38592-6_3
38. Mostowski, W.: Dynamic frames based verification method for concurrent Java programs. In: Gurfinkel, A., Seshia, S.A. (eds.) VSTTE 2015. LNCS, vol. 9593, pp. 124–141. Springer, Heidelberg (2016). doi:10.1007/978-3-319-29613-5_8
39. Mostowski, W., Ulbrich, M.: Dynamic dispatch for method contracts through abstract predicates. In: 15th International Conference on MODULARITY (MODULARITY 2015), pp. 109–116. ACM (2015)
40. Nordio, M., Calcagno, C., Meyer, B., Müller, P., Tschannen, J.: Reasoning about function objects. In: Vitek, J. (ed.) TOOLS 2010. LNCS, vol. 6141, pp. 79–96. Springer, Heidelberg (2010). doi:10.1007/978-3-642-13953-6_5
41. O'Hearn, P.W.: Resources, concurrency and local reasoning. Theor. Comput. Sci. **375**(1–3), 271–307 (2007)
42. Parkinson, M.J., Bierman, G.M.: Separation logic and abstraction. In: Proceedings of POPL (2005)
43. Parkinson, M.J., Bierman, G.M.: Separation logic, abstraction and inheritance. In: Proceedings of POPL (2008)

44. Parkinson, M.J., Summers, A.J.: The relationship between separation logic and implicit dynamic frames. In: Barthe, G. (ed.) ESOP 2011. LNCS, vol. 6602, pp. 439–458. Springer, Heidelberg (2011). doi:10.1007/978-3-642-19718-5_23
45. Schmitt, P.H., Ulbrich, M., Weiß, B.: Dynamic frames in Java dynamic logic. In: Beckert, B., Marché, C. (eds.) FoVeOOS 2010. LNCS, vol. 6528, pp. 138–152. Springer, Heidelberg (2011). doi:10.1007/978-3-642-18070-5_10
46. Soundarajan, N., Fridella, S.: Reasoning about polymorphic behavior. In: Proceedings of TOOLS, pp. 346–358. IEEE Computer Society (1998)
47. Weiß, B.: Predicate abstraction in a program logic calculus. Sci. Comput. Program. **76**(10), 861–876 (2011)