Dynamic Frames in Java Dynamic Logic

Peter H. Schmitt, Mattias Ulbrich, and Benjamin Weiß

Karlsruhe Institute of Technology
Institute for Theoretical Computer Science
D-76128 Karlsruhe, Germany
`{pschmitt,mulbrich,bweiss}@ira.uka.de`

Abstract. In this paper we present a realisation of the concept of dynamic frames in a dynamic logic for verifying Java programs. This is achieved by treating sets of heap locations as first class citizens in the logic. Syntax and formal semantics of the logic are presented, along with sound proof rules for modularly reasoning about method calls and heap dependent symbols using specification contracts.

1 Introduction

To successfully support modular verification of object-oriented software, it is essential to be able to define relevant portions of memory and reason about the effects of method execution on them. Portions of memory, i.e., sets of heap locations, are called *frames* in this context or—since they themselves are subject to change during program execution—*dynamic frames*. The theoretical concept of dynamic frames was introduced in [7] and first implemented in [21] and later in [10]. Specification with dynamic frames is related to the use of data groups [11], separation logic [16, 20], and to approaches based on ownership types [1, 15].

In this paper we investigate the integration of the dynamic frames specification style into the verification of sequential Java programs based on *dynamic logic* [5]. In many verification methods, the task of verifying that a property φ holds after execution of a program `p` is solved by successively computing *weakest preconditions* [4] in first-order predicate logic of parts of the program starting from its end. In dynamic logic, the weakest precondition can be directly written, thanks to the modal operator $[\cdot]$, as the formula $[\mathtt{p}]\varphi$. Dynamic logic can be augmented with a symbolic representation of state changes called *updates* [18]. This extension allows giving inference rules for dynamic logic that compute (first-order) weakest preconditions by performing a *forward symbolic execution* of the program `p` starting from the beginning. The proof tree that unfolds by successive applications of these rules will eventually contain only first-order proof subgoals. This form of verification is the foundation of the KeY system [2]. Dynamic logic is also used for Java verification in the KIV system [22].

An issue in program verification to be addressed no matter how proof obligations at the program level are transformed to first-order proof goals is the representation of the heap. In a closed-world setting, where the entire program is known at verification time, an explicit heap representation can be dispensed with, saving some complexity. This was e.g. realised in the KeY system. In a modular setting, where one strives for abstract specification of interfaces and local reasoning, the situation is different: here, reasoning about which frame is changed by a program, or about which frame the execution of a program depends on, becomes crucial. In this setting, the flexibility provided by an explicit representation of the heap seems to offer decisive advantages.

In Sect. 2 we motivate the use of dynamic frames with a simple example. The dynamic logic to be presented will explicitly represent dynamic frames as sets of locations. Syntax and semantics and some exemplary proof rules of this logic are given in Sect. 3. Contract-based proof obligations and proof rules for verifying dynamic frames specifications are defined in Sect. 4. Conclusions in Sect. 5 wrap up the paper.

2 Motivating Example

As an example, we consider the Java program shown in Fig. 1. The intention behind the `List` interface is that objects of this type represent lists of objects. The interface provides methods for querying the size of the list, retrieving an element out of the list at a given index, and appending an element to the end of the list. Class `ArrayList` implements the interface with the help of an array, and class `Client` is an artificial snippet of client code using the interface.

Our goal is to specify this program following the *design by contract* paradigm [14]. That is, we are interested in providing *pre- and postconditions* for the methods of the program, where we refer to a pair of a pre- and a postcondition as a *method contract*. Furthermore, the goal is to *verify* the correctness of these contracts using dynamic logic, and to do so in a *modular* (or *local*) fashion: the verification of a given method should not make use of implementational details that are not visible in this method. For example, when verifying `m` in `Client`,

Java

```
interface List {
  int size();
  Object get(int i);
  void add(Object o);
}
class Client {
  public int x;
  Object m(List l) {
    x++;
    return l.get(0);
  }
}
```

```
class ArrayList implements List {
  private int n = 0;
  private Object[] a = new Object[10];
  public int size() {
    return n;
  }
  public Object get(int i) {
    if(0 <= i && i < n) return a[i];
    else return null;
  }
  //method "add" omitted
}
```

Java

Fig. 1. Example program

we do not want to make use of the fact that there is only one implementation of `List`, nor of the internals of this particular implementation. Instead, reasoning about the dynamically bound call to `get` should be based only on the contract for `get` in the interface. For subtypes of the interface, we only require that all overriding method bodies satisfy the contracts given at the level of the interface; this means that we enforce *behavioural subtyping* [12].

A main difficulty in specifying an interface such as `List` is that we do not have access to any implementational data structures for writing our specifications. The general solution is to use *data abstraction* [6]: we specify the interface in a more abstract fashion, using either some form of *abstract fields* (sometimes called *model fields* [3]), or side-effect free methods present in the program. Here, we choose to specify `get` with the help of the `size` method, and with the help of an abstract Boolean field *inv*:

$$\textit{pre}\colon \texttt{this.}\textit{inv} \land 0 \le \texttt{i} \land \texttt{i} < \texttt{this.size()} \qquad \textit{post}\colon \texttt{res} \not\doteq \texttt{null}$$

We use a dot to distinguish some syntactic operators of the logic (such as $\doteq$) from meta-level operators (such as $=$). Java's `==` operator translates to $\doteq$ in the logic. The identifier `res` refers to the method's return value.

In class `ArrayList`, the meaning of the symbol `size` is defined by the method body for `size`. Similarly, we need to give a definition for the abstract field *inv*, which we do with the following axiom:

$$\begin{aligned} &\textit{exactInstance}_{\texttt{ArrayList}}(\texttt{this}) \\ &\to \big(\texttt{this.}\textit{inv} \;\leftrightarrow\; \texttt{this.a} \not\doteq \texttt{null} \land \texttt{this.n} < \texttt{this.a.length} \\ &\qquad\qquad\qquad \land\; \forall \textit{Int}\, i; (0 \le i \land i < \texttt{this.n} \to \texttt{this.a[}i\texttt{]} \not\doteq \texttt{null})\big) \end{aligned} \tag{1}$$

For a type A and an expression `e`, the formula $\textit{exactInstance}_A(\texttt{e})$ evaluates to true in a state if the dynamic type of `e` is A. Intuitively, *inv* represents an "object invariant" for `List`, i.e., a consistency property on its objects, where the exact nature of this property is defined privately in subclasses of the interface. With the definition for `ArrayList` in (1), the implementation of `get` in `ArrayList` satisfies the method contract for `get`.

For method `m` in `Client`, we give the following method contract:

$$\textit{pre}\colon \texttt{l} \not\doteq \texttt{null} \land \texttt{l.}\textit{inv} \land 0 < \texttt{l.size()} \qquad \textit{post}\colon \texttt{res} \not\doteq \texttt{null}$$

Can we verify that `m` complies with this contract, provided that all implementations of `get` satisfy the contract for `get`? Unfortunately, the answer is no. The problem is that even though the precondition guarantees properties about the *initial* values of `l.`*inv* and `l.size()`, this does not imply that these properties still hold when `get` is called at the end of `m`, because of the intervening change to `x`. This is an instance of a general problem when using data abstraction in specifications [8, Challenge 3]: without further measures, any change to the heap can affect the value of an abstract field or of a method in an unknown way.

As a solution, we introduce *dependency contracts* (also known as *depends clauses* [9]) into our specifications. A dependency contract restricts the set of memory locations that are allowed to influence the value of an abstract field or of a method, provided that some precondition holds. An example for a correct dependency contract for method `size` in `ArrayList` is one which states that the method result is allowed to depend only on $\{(\texttt{this}, \texttt{n})\}$, where the expression $\{(\texttt{this}, \texttt{n})\}$ refers to the set consisting of the single memory location given by the field `n` for the object represented by the expression `this`.

How can we express a useful dependency contract for *inv* or `size` in `List`, even though here we do not have access to the locations implementing the list? We see that the need for data abstraction also extends to location sets. Our solution is to use *dynamic frames* [7], i.e., abstract fields that evaluate to sets of memory locations. For the specification of `List`, we declare a dynamic frame *locs*. In `ArrayList`, we define *locs* via the following axiom:

$$\mathit{exactInstance}_{\texttt{ArrayList}}(\texttt{this}) \;\rightarrow\; \texttt{this.}\mathit{locs} \doteq (\texttt{this.*} \mathbin{\dot\cup} \texttt{this.a.*}) \qquad (2)$$

The expression `o.*` refers to the set of all fields of the object represented by the expression `o`. If `o` has an array type, then `o.*` denotes all components of the array.

We use the dynamic frame *locs* to give dependency contracts for both *inv* and `size`: both are supposed to depend at most on the locations in *locs*. These dependency contracts are satisfied in `ArrayList`, because both $\texttt{this.}\mathit{inv}$ (as defined by (1)) and `this.size()` (as defined by the method body in Fig. 1) read only locations that are members of $\texttt{this.}\mathit{locs}$ as defined by (2).

Finally, we modify the precondition of `m` in `Client` to be as follows:

$$\mathit{pre}\colon \texttt{l} \neq \texttt{null} \wedge \texttt{l.}\mathit{inv} \wedge 0 < \texttt{l.size()} \wedge (\texttt{this}, \texttt{x}) \mathrel{\dot\notin} \texttt{l.}\mathit{locs}$$

Now, when reasoning about the correctness of `m`, we know that the location $(\texttt{this}, \texttt{x})$ is not a member of the (unknown) set of locations $\texttt{l.}\mathit{locs}$ on which $\texttt{l.}\mathit{inv}$ and `l.size()` may depend. Thus, changing the value of this location cannot have an effect on the values of $\texttt{l.}\mathit{inv}$ and `l.size()`, and so $\texttt{l.}\mathit{inv} \wedge 0 < \texttt{l.size()}$ must still be true when method `get` is called at the end of `m`. Together with the method contract for `get`, this guarantees that the return value of `get` is different from `null`, and thus that the postcondition of `m` is satisfied.

In general, we also need *modifies clauses* in method contracts, which fix a set of locations that may at most be modified by a method, provided that the precondition of the contract holds upon method entry. In the example, `get` and `size` are supposed to not have side effects, so we can use modifies clauses of $\dot\emptyset$ (an empty set of locations). For `add`, $\texttt{this.}\mathit{locs}$ can serve as a modifies clause.

Also, as the value of the dynamic frame $\texttt{this.}\mathit{locs}$ is itself state-dependent, specifications of the behaviour of *locs* itself are needed in order to make the specification fully useful for modular verification. We can give a dependency contract for $\texttt{this.}\mathit{locs}$ stating that its value depends at most on the locations in $\texttt{this.}\mathit{locs}$ itself; this is satisfied by the definition (2), because the only location it reads is $(\texttt{this}, \texttt{a})$, which itself is defined to be a member of $\texttt{this.}\mathit{locs}$. We may also want to specify (via method contracts) that after the construction of an `ArrayList` object, the set $\texttt{this.}\mathit{locs}$ contains only freshly allocated locations, and that method `add` can add to the set only freshly allocated locations (the latter is sometimes called the "swinging pivots property" [11, 7]).

3 Java Dynamic Logic With an Explicit Heap Model

In this section, we present a dynamic logic and a sequent calculus for the modular verification of Java source code wrt. dynamic frames style specifications. It is a variation of the dynamic logic underlying the KeY verification tool [2]. The main difference is in the logical modelling of heap memory. For complete formal definitions please see the technical report [19], which accompanies this paper.

3.1 Syntax and Semantics

The *syntax* of the logic is based on a *signature* Σ, which comprises a set $\mathcal{T}$ of *types*, a partial order $\sqsubseteq$ called the *subtype relation*, and disjoint sets of (logical) *variables* $\mathcal{V}$, *program variables* $\mathcal{PV}$, *function symbols* $\mathcal{F}$, and *predicate symbols* $\mathcal{P}$. All variables and symbols are typed. We use the notation $x : A$ to indicate that the type of x is A, the notation $f : A_1, \ldots, A_n \to B$ to indicate that the function symbol f maps arguments of types $A_1, \ldots, A_n$ to type B, and the notation $p : A_1, \ldots, A_n$ to indicate that the predicate symbol p represents a relation on the types $A_1, \ldots, A_n$. The signature Σ is specific to a Java program to be verified. All types of this program also appear as types in $\mathcal{T}$, and all local variables appear as program variables in $\mathcal{PV}$. In contrast to program variables, *logical* variables may not appear in programs, but may be quantified. The type $\mathit{Any} \in \mathcal{T}$ is a supertype of all types of the program.

The set Fma_Σ of *formulas* and the set Trm_Σ of *terms* are defined mostly as in classical typed first-order logic. For any type $A \in \mathcal{T}$, we have the set $\mathit{Trm}^A_\Sigma \subseteq \mathit{Trm}_\Sigma$ of terms of type A. In addition to the operators of first-order logic, Java dynamic logic includes modal operators $[\mathtt{p}]$ and $\langle \mathtt{p} \rangle$ for every executable Java program fragment $\mathtt{p}$. If $\varphi \in \mathit{Fma}_\Sigma$ is a formula, then both $[\mathtt{p}]\varphi$ and $\langle \mathtt{p} \rangle\varphi$ are also formulas. Our version of dynamic logic also includes another kind of modal operator, called *updates* [18]. An update is denoted as $\mathtt{a}_1 := t_1 \parallel \ldots \parallel \mathtt{a}_n := t_n$, where $\mathtt{a}_1, \ldots, \mathtt{a}_n \in \mathcal{PV}$, and where $t_1, \ldots, t_n$ are terms such that the type of t_i is a subtype of the type of $\mathtt{a}_i$. The set of updates is called Upd_Σ. If u is an update and t is a term or formula, then $\{u\}t$ is also a term or formula, respectively.

The *semantics* of a term or formula is given by an *interpretation* which maps all function symbols to functions and all predicate symbols to relations, and by a *state* which maps all program variables to values. First-order terms and formulas are evaluated as usual. The formula $[\mathtt{p}]\varphi$ holds in a state s if the execution of $\mathtt{p}$ started in s either does not terminate, or terminates in a state s' such that φ holds in s' (*partial* correctness). The formula $\langle \mathtt{p} \rangle\varphi$ holds if $[\mathtt{p}]\varphi$ holds, and if additionally $\mathtt{p}$ does indeed terminate (*total* correctness). Like a program $\mathtt{p}$, an update u changes the state: executing the update $\mathtt{a}_1 := t_1 \parallel \ldots \parallel \mathtt{a}_n := t_n$ in a state s leads to an updated state s' which is identical to s, except that the program variables $\mathtt{a}_i$ have been assigned the values of the terms t_i in parallel. Evaluating $\{u\}t$ in s is the same as evaluating t in the updated state s'. A formula is called *logically valid* if it holds for all interpretations and all states.

3.2 Sequent Calculus

The calculus we use to reason about logical validity of formulas is a *sequent calculus*. A proof in the sequent calculus is a tree of so-called *sequents* $\Gamma \Rightarrow \Delta$, in which Γ (called the *antecedent*) and Δ (the *succedent*) are finite sets of formulas. A sequent $\Gamma \Rightarrow \Delta$ has the same semantic truth value as the formula $\bigwedge \Gamma \rightarrow \bigvee \Delta$.

An *inference rule* of the sequent calculus has a number of sequents as its *premisses* and a single sequent as its *conclusion*; it is *sound* if logical validity of all premisses implies logical validity of the conclusion. In addition to inference rules, our calculus contains *rewrite rules*, which allow rewriting a term or formula at an arbitrary position in a sequent. A rewrite rule is sound if the original and the rewritten term or formula are equal resp. logically equivalent. We formulate both sequent and rewrite rules schematically to achieve a finite representation of the calculus. For example, in the following two (sound) rule schemata, the *schema formulas* φ and ψ can be instantiated with arbitrary formulas, and Γ and Δ with arbitrary sets of formulas:

$$(\mathsf{andRight})\quad \frac{\Gamma \Rightarrow \varphi, \Delta \quad \Gamma \Rightarrow \psi, \Delta}{\Gamma \Rightarrow \varphi \wedge \psi, \Delta} \qquad\qquad (\mathsf{andIdem})\quad \psi \wedge \psi \;\rightsquigarrow\; \psi$$

Starting with the sequent to prove as root, a *proof tree* is constructed by applying sequent and rewrite rules. For the application of a sequent rule to a leaf in the proof tree, this sequent must be identical to the conclusion of the rule. The rule's premisses are then added as new children to the former leaf. A rewrite rule $t_1 \rightsquigarrow t_2$ can be applied to a leaf by replacing one occurrence of t_1 in its sequent by t_2. Provided that all applied rules are sound, it is guaranteed that at any time during this process, validity of all the leaves implies validity of the root sequent. If one arrives at a tree whose leaves are all obviously valid, one has proven the validity of the original proof obligation.

3.3 Heap Model

In contrast to [2, 18], where the Java heap is modelled via a *non-rigid* function symbol $\mathtt{f} : A \rightarrow B$ for every Java field `f` of type `B` declared in class `A`, here we follow [17, 22, 1, 21] in modelling the heap using the *theory of arrays* [13]. The fields of our Java program are represented as constant symbols of a type $\mathit{Field} \in \mathcal{T}$, which are axiomatised to have distinct values. Heaps now occur "explicitly" in formulas, as terms of a type $\mathit{Heap} \in \mathcal{T}$. The values of this type are arrays indexed by locations, i.e., by pairs of $(\mathit{Object}, \mathit{Field})$ values. Reading from and writing to a heap is done with the help of the function symbols $\mathit{select}_A : \mathit{Heap}, \mathit{Object}, \mathit{Field} \rightarrow A$ and $\mathit{store} : \mathit{Heap}, \mathit{Object}, \mathit{Field}, \mathit{Any} \rightarrow \mathit{Heap}$. These are standard, except that for convenience we use a separate symbol select_A for every type $A \in \mathcal{T}$, which implicitly casts the retrieved value to a desired type A. A global program variable $\mathtt{heap} : \mathit{Heap} \in \mathcal{PV}$ holds the current heap of the program. We will in the following often use the more concise notation $o.f$ instead of $\mathit{select}_A(\mathtt{heap}, o, f)$.

The axiom of the theory of arrays manifests itself in the rewrite rule selectOf-Store depicted in Fig. 2: The value $\mathit{select}_A(\mathit{store}(h, o, f, t), o', f')$ of a location

$$select_A(store(h,o,f,t),o',f') \rightsquigarrow \qquad \text{(selectOfStore)}$$
$$if(o \doteq o' \wedge f \doteq f')\,then(cast_A(t))\,else(select_A(h,o',f'))$$

$$select_A(anon(h,s,h'),o,f) \rightsquigarrow \qquad \text{(selectOfAnon)}$$
$$\begin{aligned}&if\big(((o,f)\mathrel{\dot\in} s \wedge f \not\doteq created) \vee (o,f)\mathrel{\dot\in} freshLocs(h)\big)\\ &then(select_A(h',o,f))\\ &else(select_A(h,o,f))\end{aligned}$$

$$cast_A(t) \rightsquigarrow t \qquad \text{for } t \in Trm_\Sigma^{A'} \text{ and } A' \sqsubseteq A \qquad \text{(cast)}$$

$$(o,f)\mathrel{\dot\in} freshLocs(h) \rightsquigarrow \qquad \text{(inFreshLocs)}$$
$$o \not\doteq \texttt{null} \wedge select_{Boolean}(h,o,created) \doteq FALSE$$

$$[\texttt{a = }t\texttt{;}\ldots]\varphi \rightsquigarrow \{\texttt{a} := t\}[\ldots]\varphi \qquad \text{(assignLocal)}$$

$$[o\texttt{.}f\texttt{ = }t\texttt{;}\ldots]\varphi \rightsquigarrow \{\texttt{heap} := store(\texttt{heap},o,f,t)\}[\ldots]\varphi \qquad \text{(assignField)}$$

Fig. 2. A selection of rewrite rules for heap modifications and location sets

(o,f) retrieved from a modified heap $store(h,o,f,t)$ depends on whether the retrieved location is the previously modified one, i.e., whether $(o',f') \doteq (o,f)$ holds. If so, the assigned value t is read, otherwise the retrieval is delegated to the embedded heap h as $select_A(h,o',f')$. The type coercion operation $cast_A(t)$ can later be removed using the rule cast if the heap has been used consistently.

In our logic, all states share a common semantic domain (this is known as the *constant domain assumption*). Therefore, we need a means to explicitly distinguish between already-created and not-yet-created objects in the sense of Java. We use an implicit ("ghost") field $created : Field$ for this purpose: we consider an object o to be created in a state if and only if $o\texttt{.}created$ evaluates to true in this state. Allocating an object via Java's `new` operator implicitly sets its *created* field to true.

Dynamic frames are supported via a type $LocSet \in \mathcal{T}$. Terms of type $LocSet$ evaluate to sets of memory locations. Our signatures contain the symbols $\dot\emptyset$, $\dot\cup$, $\dot\cap$, $\dot\setminus$, $\dot\in$, $\dot\subseteq$, *disjoint* and *allLocs*, which are pre-defined to have their expected set-theoretical semantics. The function symbol $freshLocs : Heap \to LocSet$ yields for every heap the set of locations (o,f) for which the object o is not yet created in this heap. The corresponding rule inFreshLocs is shown in Fig. 2.

When dispatching a method call in a proof with the help of a contract for the called method (Sect. 4), we use a special heap modification function $anon : Heap, LocSet, Heap \to Heap$. Roughly, the heap $anon(h,s,h')$ is identical to h' in the locations of s, and it is identical to the "original" heap h in all other locations. The exact semantics of *anon* is described by the rewrite rule selectOfAnon in Fig. 2: independently of the set s, going from h to $anon(h,s,h')$ for some unknown h' (a process which we call an "anonymisation" of the heap h wrt. the set s) never leads to deallocating existing objects, but always implicitly allows for the allocation of new objects. This resembles the behaviour of method calls in Java.

We also introduce a unary predicate symbol $\mathit{wellFormed} : \mathit{Heap}$, which can be axiomatised as

$$\forall \mathit{Heap}\, h; \big(\mathit{wellFormed}(h) \leftrightarrow \forall \mathit{Object}\, o, p; \forall \mathit{Field}\, f;$$
$$(\mathit{select}_{\mathit{Any}}(h, o, f) \doteq p \rightarrow (p \doteq \texttt{null} \lor \mathit{select}_{\mathit{Boolean}}(h, p, \mathit{created}) \doteq \mathit{TRUE}))\big) \ ,$$

i.e., a heap h is considered well-formed if any object p which is referenced by some location (o, f) is either the `null` object or an object which has already been created. The semantics of Java guarantees that $\mathit{wellFormed}(\texttt{heap})$ holds for all states occurring during the execution of a Java program.

3.4 Symbolic Execution

A central component of our calculus is a set of rule schemata that allow us to transform formulas with program modalities and updates into formulas without. This process is called *symbolic execution.* Programs are systematically processed in a forward manner: whenever we encounter a formula $[\texttt{p;q}]\varphi$, we handle the statement `p` first, and leave the formula $[\texttt{q}]\varphi$ to be treated later. This forward treatment of programs is based on the concept of updates. There is also a set of rules which handle the simplification and application of updates to terms and formulas. The theory of rule-based update treatment has been elaborated in [18].

Two rules for symbolic execution, namely assignLocal and assignField, are shown in Fig. 2. The corresponding rules for the modality $\langle\cdot\rangle$ read accordingly. Both rules are used to execute assignment statements, either for a local variable `a` or for a field reference $o\texttt{.}f$. Let t be a side-effect free Java expression which (after some syntactic adaptions like `==` to $\doteq$, `&&` to $\land$, etc.) can be read as a term in our logic. An assignment statement `a = `t`;` which assigns to `a` the value of the expression t, describes then the same state modification as the update $\texttt{a} := t$. This is captured in the symbolic execution rule assignLocal. An assignment to a location $o\texttt{.}f$ is treated differently: it corresponds to a modification of the global program variable `heap`. We do not show the rules for other language features here, as they are numerous and largely orthogonal to the focus of this paper. We also ignore Java exceptions throughout the paper, which allows for a more readable presentation of rules and proof obligations. For a more complete treatment of Java language features, please refer to [2].

Fig. 3 depicts a small example proof. Therein, $\texttt{o} \in \mathcal{PV}$ is a local variable of a reference type, $\texttt{f} : \mathit{Field} \in \mathcal{F}$ is a constant symbol, and $\texttt{a} : \mathit{Int} \in \mathcal{PV}$ is a local variable. Symbolic execution first converts the two Java assignments into corresponding updates. The updates are then simplified into a single update that performs both state changes in parallel. The left sub-update $\texttt{heap} := \mathit{store}(\texttt{heap}, \texttt{o}, \texttt{f})$ can be simplified away, because the variable `heap` does not occur in the scope of the update any more, and thus its value is irrelevant. The rule selectOfStore is applied inside the remaining update, followed by an obvious simplification of the resulting *if-then-else*-term. The type cast operator can be removed with the cast rule, because 0 is of type *Int*. Finally, the update is applied to the sub-formula $\texttt{a} \doteq 0$ as a substitution, resulting in an obviously valid formula. Hence, we have proven that the original formula is valid as well.

$$\begin{array}{ll}
 & [\texttt{o.f = 0; a = o.f;}](\texttt{a} \doteq 0) \\
\overset{\text{assignField}}{\rightsquigarrow} & \{\texttt{heap} := \mathit{store}(\texttt{heap}, \texttt{o}, \texttt{f}, 0)\}[\texttt{a = o.f;}](\texttt{a} \doteq 0) \\
\overset{\text{assignLocal}}{\rightsquigarrow} & \{\texttt{heap} := \mathit{store}(\texttt{heap}, \texttt{o}, \texttt{f}, 0)\}\{\texttt{a} := \mathit{select}_{\mathit{Int}}(\texttt{heap}, \texttt{o}, \texttt{f})\}(\texttt{a} \doteq 0) \\
\overset{\mathit{upd.\ simpl.}}{\rightsquigarrow} & \{\texttt{heap} := \mathit{store}(\texttt{heap}, \texttt{o}, \texttt{f}, 0) \parallel \texttt{a} := \mathit{select}_{\mathit{Int}}(\mathit{store}(\texttt{heap}, \texttt{o}, \texttt{f}, 0), \texttt{o}, \texttt{f})\}(\texttt{a} \doteq 0) \\
\overset{\mathit{upd.\ simpl.}}{\rightsquigarrow} & \{\texttt{a} := \mathit{select}_{\mathit{Int}}(\mathit{store}(\texttt{heap}, \texttt{o}, \texttt{f}, 0), \texttt{o}, \texttt{f})\}(\texttt{a} \doteq 0) \\
\overset{\text{selectOfStore}}{\rightsquigarrow} & \{\texttt{a} := \mathit{if}(\texttt{o} \doteq \texttt{o} \land \texttt{f} \doteq \texttt{f})\mathit{then}(\mathit{cast}_{\mathit{Int}}(0))\mathit{else}(\mathit{select}_{\mathit{Int}}(\texttt{heap}, \texttt{o}, \texttt{f}))\}(\texttt{a} \doteq 0) \\
\overset{\mathit{simpl.}}{\rightsquigarrow} & \{\texttt{a} := \mathit{cast}_{\mathit{Int}}(0)\}(\texttt{a} \doteq 0) \\
\overset{\text{cast}}{\rightsquigarrow} & \{\texttt{a} := 0\}(\texttt{a} \doteq 0) \\
\overset{\mathit{upd.\ appl.}}{\rightsquigarrow} & 0 \doteq 0
\end{array}$$

Fig. 3. Example proof

4 Contracts and Proof Obligations

Both abstract fields, such as *inv* and *locs* in Sect. 2, and side-effect free methods such as `size` are represented in the logic as so-called *observer symbols.*

Definition 1 (Observer symbols). *An* observer symbol for type A with argument types $B_1, \ldots, B_n$ *is either a function symbol* $\mathit{obs} : \mathit{Heap}, A, B_1, \ldots, B_n \to B \in \mathcal{F}$ *or a predicate symbol* $\mathit{obs} : \mathit{Heap}, A, B_1, \ldots, B_n \in \mathcal{P}$*, where* $A \sqsubseteq \mathit{Object}$*.*

As syntactic sugar, we sometimes write $o.\mathit{obs}(p_1, \ldots, p_n)$ to denote the term or formula $\mathit{obs}(\texttt{heap}, o, p_1, \ldots, p_n)$. This (deliberately) resembles the notation $o.f$ for field access terms $\mathit{select}_A(\texttt{heap}, o, f)$. Nevertheless, an observer symbol does not give rise to a memory location; instead, it "observes" (i.e., it depends on) the values of memory locations. For an observer symbol `m` representing a side-effect free method without parameters, we sometimes write o`.m()` instead of o`.m`.

We have seen in Sect. 2 that the value of abstract fields is defined via axioms such as (1) and (2). Similarly, observer symbols representing *methods* are defined via axioms such as the following (where `this` and `r` are fresh program variables):

$$\begin{aligned}
&\mathit{exactInstance}_{\texttt{ArrayList}}(\texttt{this}) \\
&\to \forall \mathit{Int}\, i; \big(\texttt{this.size()} \doteq i \;\leftrightarrow\; \langle \texttt{r = this.size();} \rangle \texttt{r} \doteq i\big)
\end{aligned} \tag{3}$$

The axiom uses the modal operator $\langle\cdot\rangle$ to connect the observer symbol `size` with a call to method `size` in class `ArrayList`.

Axioms (1), (2), (3) are supposed to hold for all values of the program variables `this` and `heap`. The corresponding universally quantified versions of the axioms can be used as assumptions in proofs for the correctness of `ArrayList`. We could also allow using them in other proofs, but this is undesirable for reasons of modularity: the axioms are implementational secrets of `ArrayList`, and should not be exposed to other classes.

Besides observer symbols and axioms, a *specification* in our setting consists of a set of *method contracts* constraining the behaviour of methods, and of a set of *dependency contracts* constraining the dependencies of observer symbols. Both kinds of contract give rise to *proof obligations*, i.e., formulas whose validity must be proven in order for the program to be considered correct. On the other hand, both kinds of contract can also be used as assumptions in the proofs of other contracts, via special *rules*. Subsect. 4.1 defines method contracts, the corresponding proof obligation, and the corresponding rule; Subsect. 4.2 does the same for dependency contracts. Note that for simplicity of presentation, we omit the treatment of void methods, static methods, static fields, and constructors.

4.1 Method Contracts

Definition 2 (Method contracts). *A* method contract *mct is a tuple*

$$mct = \big(\texttt{m}, \texttt{this}, (\texttt{p}_1, \ldots, \texttt{p}_n), \texttt{res}, \texttt{hPre}, pre, post, mod, \tau\big)$$

where $\texttt{m}$ *is a Java method; where* $\texttt{this} : A \in \mathcal{PV}$ *such that* $\texttt{m}$ *is defined for receiver objects of type* A*; where* $\texttt{p}_1, \ldots, \texttt{p}_n, \texttt{res} \in \mathcal{PV}$ *such that their types correspond to the declared signature of* $\texttt{m}$*; where* $\texttt{hPre} : Heap \in \mathcal{PV}$*; and where* $pre, post \in Fma_\Sigma$*,* $mod \in Trm_\Sigma^{LocSet}$*, and* $\tau \in \{partial, total\}$*.*

The program variables $\texttt{this}$ and $\texttt{p}_1, \ldots, \texttt{p}_n$ may be used in the precondition *pre*, in the postcondition *post* and in the modifies clause *mod* to represent the receiver object of $\texttt{m}$ and the arguments to $\texttt{m}$, respectively. The variables $\texttt{res}$ and $\texttt{hPre}$ can be used in *post* to refer to the method's return value and to the value of $\texttt{heap}$ in the pre-state. The "termination marker" τ indicates whether the contract demands partial or total correctness.

Definition 3 (Proof obligation for method contracts). *Given a method contract* $mct = \big(\texttt{m}, \texttt{this}, (\texttt{p}_1, \ldots, \texttt{p}_n), \texttt{res}, \texttt{hPre}, pre, post, mod, \tau\big)$ *with* $\texttt{this} : A$*, and given a type* $B \sqsubseteq A$*, the proof obligation* $CorrectMethodContract(mct, B) \in Fma_\Sigma$ *is defined as*

$$\begin{aligned} & pre \wedge reachableState \wedge exactInstance_B(\texttt{this}) \\ & \to \{\texttt{hPre} := \texttt{heap}\}[\![\texttt{res = this.m(}\texttt{p}_1, \ldots, \texttt{p}_n\texttt{);}]\!](post \wedge frame), \end{aligned}$$

where $[\![\cdot]\!]$ *stands for* $[\cdot]$ *if* $\tau = partial$ *and for* $\langle\cdot\rangle$ *if* $\tau = total$*, and where*

– *reachableState is the formula*

$$\begin{aligned} & wellFormed(\texttt{heap}) \wedge \texttt{this} \neq \texttt{null} \wedge \texttt{this}.created \doteq TRUE \\ & \wedge \bigwedge_{i \in \{1,\ldots,n\},\ \texttt{p}_i : A \text{ for some } A \sqsubseteq Object} (\texttt{p}_i \doteq \texttt{null} \vee \texttt{p}_i.created \doteq TRUE) \end{aligned}$$

– *frame is the formula*

$$\begin{aligned} \forall Object\, o; \forall Field\, f; \big((o, f) \,\dot{\in}\, \{\texttt{heap} := \texttt{hPre}\}\big(mod \,\dot{\cup}\, freshLocs(\texttt{heap})\big) \\ \vee\, o.f \doteq \{\texttt{heap} := \texttt{hPre}\}o.f\big) \end{aligned}$$

The *reachableState* property is guaranteed by Java itself: the heap is well-formed, the receiver object is created, and all objects passed as arguments are either `null` or created. The formula *frame* is the frame condition generated from the modifies clause *mod*: after executing `m`, only locations in *mod* (interpreted in the pre-state) and "fresh" locations may have changed compared to the pre-state.

For method `get` with *pre* and *post* from Sect. 2, $\tau = \mathit{total}$, and $B =$ `ArrayList`, we get the following instance of *CorrectMethodContract*:

$$\begin{aligned}&\mathtt{this}.\mathit{inv} \land 0 \leq \mathtt{i} \land \mathtt{i} < \mathtt{this.size()} \land \mathit{wellFormed}(\mathtt{heap})\\ &\land \mathtt{this} \not\doteq \mathtt{null} \land \mathtt{this}.\mathit{created} \doteq \mathit{TRUE} \land \mathit{exactInstance}_{\mathtt{ArrayList}}(\mathtt{this})\\ &\to \{\mathtt{hPre} := \mathtt{heap}\}\langle \mathtt{res = this.get(i);}\rangle(\mathtt{res} \not\doteq \mathtt{null} \land \mathit{frame})\end{aligned}$$

where *frame* with a modifies clause $\mathit{mod} \doteq \emptyset$ states that only fresh locations may have been changed by `m`. The formula is valid under the assumption of (the universally quantified versions of) axioms (1) and (3). When proving this, one of the first steps is to inline the body of method `get`, which is possible because we know the exact type of `this` and, hence, do not have to consider dynamic dispatch.

The following rule allows using a method contract as an assumption:

Definition 4 (Rule useMethodContract).

$$\frac{\begin{array}{l}\Gamma \Rightarrow \{u\}\{w\}(\mathit{pre} \land \mathit{reachableState}),\ \Delta\\ \Gamma \Rightarrow \{u\}\{w\}\{\mathtt{hPre} := \mathtt{heap}\}\{v\}(\mathit{post} \land \mathit{reachableState}' \to [\![\ldots]\!]\varphi),\ \Delta\end{array}}{\Gamma \Rightarrow \{u\}[\![\mathtt{r = o.m(}\mathtt{p}'_1,\ldots,\mathtt{p}'_n\mathtt{);}\ \ldots]\!]\varphi,\ \Delta}$$

where:

- $\mathtt{o} \in \mathit{Trm}^A_\Sigma$ *for some* $A \in \mathcal{T}$ *such that there is a method contract*

$$\mathit{mct} = (\mathtt{m}, \mathtt{this}, (\mathtt{p}_1,\ldots,\mathtt{p}_n), \mathtt{res}, \mathtt{hPre}, \mathit{pre}, \mathit{post}, \mathit{mod}, \tau)$$

  where $\mathtt{this} : A$*; where* $\tau = \mathit{total}$ *if the modality* $[\![\cdot]\!]$ *is* $\langle\cdot\rangle$*, and where* τ *does not matter otherwise; and where* `this`, $\mathtt{p}_1,\ldots,\mathtt{p}_n$, `res` *and* `hPre` *do not occur in the formula* $[\![\mathtt{r = o.m(}\mathtt{p}'_1,\ldots,\mathtt{p}'_n\mathtt{);}\ \ldots]\!]\varphi$
- $\mathtt{p}'_1,\ldots,\mathtt{p}'_n$ *are terms*
- $\mathit{reachableState} \in \mathit{Fma}_\Sigma$ *is as in Def. 3, and* $\mathit{reachableState}'$ *is the formula*

$$\mathit{wellFormed}(\mathtt{heap}) \land (\mathtt{res} \doteq \mathtt{null} \lor \mathtt{res}.\mathit{created} \doteq \mathit{TRUE})$$

  if $\mathtt{res} : B$ *for some* $B \sqsubseteq \mathit{Object}$*, and the formula* $\mathit{wellFormed}(\mathtt{heap})$ *otherwise*
- $v = \big(\mathtt{heap} := \mathit{anon}(\mathtt{heap}, \mathit{mod}, h) \parallel \mathtt{r} := r' \parallel \mathtt{res} := r'\big)$
- $w = \big(\mathtt{this} := \mathtt{o} \parallel \mathtt{p}_1 := \mathtt{p}'_1 \parallel \ldots \parallel \mathtt{p}_n := \mathtt{p}'_n\big)$
- $h : \mathit{Heap} \in \mathcal{F}$ *and* $r' \in \mathcal{F}$ *are fresh symbols, i.e., they do not yet occur anywhere in the proof when applying the rule*

Like *reachableState*, $\mathit{reachableState}'$ is a property guaranteed by Java. The update v "anonymises" the locations that may be changed by the call to `m`, namely the members of the modifies clause *mod*, by setting them to unknown values with the help of the new symbol h. It also sets the result variable `r`, and its counterpart `res`, to an unknown value r'. The update w instantiates the variables used in the contract with the corresponding terms in the method call.

Instead of using *anon*, we could also anonymise (or "havoc" [10]) the entire heap, and use a framing formula like *frame* in Def. 3 to express that some locations do *not* change. The advantage of our approach is that it avoids the universal quantifiers of *frame* in applications of useMethodContract.

The useMethodContract rule is sound, provided that for all subtypes $B \sqsubseteq A$ of the static receiver type A, the proof obligation $\mathit{CorrectMethodContract}(\mathit{mct}, B)$ is logically valid. A proof of this theorem is contained in [19]. We forbid "circular" applications of the rule, such as applying the rule on a call to the method which is itself being verified in the current proof. An extension to support recursion is possible, but beyond the scope of this paper.

4.2 Dependency Contracts

Definition 5 (Dependency contracts). *A* dependency contract *is a tuple*

$$\mathit{depct} = (\mathit{obs}, \mathtt{this}, (\mathtt{p}_1, \ldots, \mathtt{p}_n), \mathit{pre}, \mathit{dep})$$

where obs is an observer symbol for type A' with argument sorts $B_1, \ldots, B_n$; where $\mathtt{this} : A \in \mathcal{PV}$ such that $A \sqsubseteq A'$; where $\mathtt{p}_1 : B_1, \ldots, \mathtt{p}_n : B_n \in \mathcal{PV}$; and where $\mathit{pre} \in \mathit{Fma}_\Sigma$, $\mathit{dep} \in \mathit{Trm}_\Sigma^{\mathit{LocSet}}$.

The program variables `this` and $\mathtt{p}_1, \ldots, \mathtt{p}_n$ can be used in the precondition *pre* and the depends clause *dep* to stand for the receiver object and the parameters of *obs*, respectively. An example for a dependency contract in the context of the program of Sect. 2 is $(\mathit{inv}, \mathtt{this}, (), \mathtt{this}.\mathit{inv}, \mathtt{this}.\mathit{locs})$, which demands that the value of $\mathtt{this}.\mathit{inv}$ should depend only on locations in $\mathtt{this}.\mathit{locs}$, provided that $\mathtt{this}.\mathit{inv}$ is true at the time.

Definition 6 (Proof obligation for dependency contracts). *For a dependency contract $\mathit{depct} = (\mathit{obs}, \mathtt{this}, (\mathtt{p}_1, \ldots, \mathtt{p}_n), \mathit{pre}, \mathit{dep})$ with $\mathtt{this} : A$, and for a type $B \sqsubseteq A$, the proof obligation $\mathit{CorrectDependencyContract}(\mathit{depct}, B) \in \mathit{Fma}_\Sigma$ is defined as follows:*

$$\begin{aligned}
&\mathit{pre} \wedge \mathit{reachableState} \wedge \mathit{exactInstance}_B(\mathtt{this}) \\
&\rightarrow \mathtt{this}.\mathit{obs}(\mathtt{p}_1, \ldots, \mathtt{p}_n) \\
&\quad \equiv \{\mathtt{heap} := \mathit{anon}(\mathtt{heap}, \mathit{allLocs} \mathbin{\dot{\setminus}} \mathit{dep}, h)\}\big(\mathtt{this}.\mathit{obs}(\mathtt{p}_1, \ldots, \mathtt{p}_n)\big)
\end{aligned}$$

where $\mathit{reachableState} \in \mathit{Fma}_\Sigma$ is as in Def. 3, where $h : \mathit{Heap} \in \mathcal{F}$ is fresh, and where $\equiv$ stands for $\doteq$ if $\mathit{obs} \in \mathcal{F}$ and for $\leftrightarrow$ if $\mathit{obs} \in \mathcal{P}$.

The proof obligation formalises the notion of *obs* "depending" only on the locations in *dep*: if we change all locations except for *dep* in an unknown way, then this must not affect *obs*. For the dependency contract for *inv* above, and for $B = \texttt{ArrayList}$, we get the following instance of *CorrectDependencyContract*:

$$\begin{aligned}&\texttt{this.}\mathit{inv} \land \mathit{wellFormed}(\texttt{heap}) \land \texttt{this} \not\doteq \texttt{null} \land \texttt{this.}\mathit{created} \doteq \mathit{TRUE}\\ &\land\ \mathit{exactInstance}_{\texttt{ArrayList}}(\texttt{this})\\ &\to\ \big(\texttt{this.}\mathit{inv} \leftrightarrow \{\texttt{heap} := \mathit{anon}(\texttt{heap}, \mathit{allLocs} \mathbin{\dot{\setminus}} \texttt{this.}\mathit{locs}, h)\}(\texttt{this.}\mathit{inv})\big)\end{aligned}$$

The formula is valid under the assumption of axioms (1) and (2), because all locations read by (1) are defined to be a part of $\texttt{this.}\mathit{locs}$ by (2). Analogously, (3) defines `this.size()` such that it also depends only on the locations in $\texttt{this.}\mathit{locs}$ as defined by (2).

Definition 7 (Rule useDependencyContract).

$$\frac{\Gamma,\ \mathit{guard} \to \mathit{equal}\ \Rightarrow\ \Delta}{\Gamma\ \Rightarrow\ \Delta}$$

where:

- *the term or formula* $\mathit{obs}(h^{new}, \texttt{o}, \texttt{p}'_1, \ldots, \texttt{p}'_n)$ *occurs in* Γ *or* Δ, *where* $h^{new} = f_1(f_2(\ldots(f_m(h^{base}, \ldots))))$ *with* $f_1, \ldots, f_m \in \{\mathit{store}, \mathit{anon}\}$, $h^{base} \in \mathit{Trm}^{\mathit{Heap}}_{\Sigma}$
- $\texttt{o} \in \mathit{Trm}^{A}_{\Sigma}$ *for some* $A \in \mathcal{T}$ *such that there is a dependency contract* $\mathit{depct} = (\mathit{obs}, \texttt{this}, (\texttt{p}_1, \ldots, \texttt{p}_n), \mathit{pre}, \mathit{dep})$, *where* $\texttt{this}{:}A$, *and where both* `this` *and* $\texttt{p}_1, \ldots, \texttt{p}_n$ *do not occur in* Γ *or* Δ
- $\texttt{hPre}{:}\mathit{Heap} \in \mathcal{PV}$ *is fresh*, $\mathit{mod} = \mathit{allLocs} \mathbin{\dot{\setminus}} \mathit{dep}$
- $\mathit{reachableState}, \mathit{frame} \in \mathit{Fma}_{\Sigma}$ *are as in Def. 3*, $w \in \mathit{Upd}_{\Sigma}$ *is as in Def. 4*
- $\mathit{noDeallocs} \in \mathit{Fma}_{\Sigma}$ *is the formula*

$$\begin{aligned}&\mathit{freshLocs}(\texttt{heap}) \mathrel{\dot{\subseteq}} \mathit{freshLocs}(\texttt{hPre})\\ &\land\ \texttt{null.}\mathit{created} \doteq \{\texttt{heap} := \texttt{hPre}\}\texttt{null.}\mathit{created}\end{aligned}$$

- *guard is the formula*

$$\begin{aligned}&\{w\}\big(\{\texttt{heap} := h^{base}\}(\mathit{pre} \land \mathit{reachableState})\\ &\qquad \land \{\texttt{hPre} := h^{base} \parallel \texttt{heap} := h^{new}\}(\mathit{frame} \land \mathit{noDeallocs})\big)\end{aligned}$$

- *equal is the formula* $\mathit{obs}(h^{new}, \texttt{o}, \texttt{p}'_1, \ldots, \texttt{p}'_n) \equiv \mathit{obs}(h^{base}, \texttt{o}, \texttt{p}'_1, \ldots, \texttt{p}'_n)$, *where* $\equiv$ *stands for* $\doteq$ *if* $\mathit{obs} \in \mathcal{F}$ *and for* $\leftrightarrow$ *if* $\mathit{obs} \in \mathcal{P}$

The useDependencyContract rule adds an assumption *guard* $\to$ *equal* to the sequent, which relates the value of *obs* in the heaps h^{base} and h^{new}. Property *noDeallocs* holds for all heap changes occurring in Java programs, where objects can be created but this process cannot be undone (we do not consider garbage collection). Property *frame* expresses that the locations in *dep* have not changed when going from h^{base} to h^{new}. If *guard* holds, then the dependency contract guarantees that *obs* has the same value for both heaps. The rule is sound if for all subtypes $B \sqsubseteq A$ of the static receiver type A the proof obligation

$CorrectDependencyContract(depct, B)$ is logically valid; this is proven in [19]. Like for method contracts, we do not allow "circular" applications of the rule.

Automatic application of this rule is not as straightforward as for useMethodContract, because the rule is nondeterministic in the choice of h^{base}, and because it can be applied repeatedly, which could lead to non-termination of automatic proof search. However, we can avoid non-termination by avoiding duplicate applications of the rule for the same pair of heap terms. To avoid a finite, but large number of "unsuccessful" applications where *guard* cannot be proven, a strategy that seems to work well in practice is to apply the rule only for choices of h^{base} for which $obs(h^{base}, \mathtt{o}, \mathtt{p}'_1, \ldots, \mathtt{p}'_n)$ already occurs somewhere in the sequent.

We conclude our treatment of dependency contracts by returning to the example of verifying method `m` from Sect. 2. The precondition of `m` guarantees that the invariant of `l` holds initially, i.e., that $inv(\mathtt{heap}, \mathtt{l})$ is true. To establish the precondition of the method call `l.get(0)` in the body of `m`, we need to establish that $inv(store(\mathtt{heap}, \mathtt{this}, \mathtt{x}, t), \mathtt{l})$ also holds (for some term t). Modularity deters us from using (1) to deduce this. Instead, we apply useDependencyContract, with $obs = inv$ and $h^{base} = \mathtt{heap}$. We get the following instantiation for *guard* (already slightly simplified):

$$\begin{aligned}
&inv(\mathtt{heap}, \mathtt{l}) \land wellFormed(\mathtt{heap}) \land \mathtt{l} \neq \mathtt{null} \land \mathtt{l}.created \doteq TRUE \\
&\land \forall Object\, o; \forall Field\, f; \big((o, f) \dot{\in} \big((allLocs \dot{\setminus} locs(\mathtt{heap}, \mathtt{l})) \dot{\cup} freshLocs(\mathtt{heap})\big) \\
&\qquad\qquad\qquad\qquad \lor select_{Any}(store(\mathtt{heap}, \mathtt{this}, \mathtt{x}, t), o, f) \\
&\qquad\qquad\qquad\qquad\quad \doteq select_{Any}(\mathtt{heap}, o, f)\big) \;\land noDeallocs
\end{aligned}$$

As the only location changed between the two heaps is $(\mathtt{this}, \mathtt{x})$, and as the precondition of `m` guarantees that $(\mathtt{this}, \mathtt{x}) \dot{\notin} locs(\mathtt{heap}, \mathtt{l})$ holds, we can prove that the instantiation of *guard* is satisfied. This allows us to use the instantiation of *equal*, namely $inv(store(\mathtt{heap}, \mathtt{this}, \mathtt{x}, t), \mathtt{l}) \leftrightarrow inv(\mathtt{heap}, \mathtt{l})$, to prove that $inv(store(\mathtt{heap}, \mathtt{this}, \mathtt{x}, t), \mathtt{l})$ holds. After an analogous derivation about the dependencies of `size`, we can establish that the precondition of `get` holds, and then conclude with the help of useMethodContract that the postcondition of `m` holds.

5 Conclusions

We have presented an extension of Harel's dynamic logic from [5] that includes explicit representations of sets of heap locations and we have demonstrated how this logic can be used to support reasoning about dynamic frames style specifications. We have focused on the details of the logic and completely ignored issues of the specification interface and the implementation of the generation of proof obligations. Suffice it to say here that the whole approach has been implemented in a variant of the KeY system[1] and successfully tested on some simple examples. The implemented system in particular comprises an extension and modification of the Java Modeling Language, JML, for dynamic frames style specifications using model fields.

[1] available at `http://i12www.ira.uka.de/~bweiss/keyheap/`

References

1. M. Barnett, R. DeLine, M. Fähndrich, K. R. M. Leino, and W. Schulte. Verification of object-oriented programs with invariants. *Journal of Object Technology (JOT)*, 3(6):27–56, 2004.
2. B. Beckert, R. Hähnle, and P. H. Schmitt, editors. *Verification of Object-Oriented Software: The KeY Approach.* LNCS 4334. Springer, 2007.
3. Y. Cheon, G. T. Leavens, M. Sitaraman, and S. H. Edwards. Model variables: cleanly supporting abstraction in design by contract. *Software—Practice and Experience*, 35(6):583–599, 2005.
4. E. W. Dijkstra. Guarded commands, nondeterminacy and formal derivation of programs. *Communications of the ACM*, 18(8):453–457, 1975.
5. D. Harel, D. Kozen, and J. Tiuryn. *Dynamic Logic.* MIT Press, 2000.
6. C. A. R. Hoare. Proof of correctness of data representations. *Acta Informatica*, 1:271–281, 1972.
7. I. T. Kassios. Dynamic frames: Support for framing, dependencies and sharing without restrictions. In *FM 2006*, LNCS 4085, pages 268–283. Springer, 2006.
8. G. T. Leavens, K. R. M. Leino, and P. Müller. Specification and verification challenges for sequential object-oriented programs. *Formal Aspects of Computing*, 19(2):159–189, 2007.
9. K. R. M. Leino. *Toward Reliable Modular Programs.* PhD thesis, California Institute of Technology, 1995.
10. K. R. M. Leino. Specification and verification of object-oriented software. Lecture Notes, Marktoberdorf International Summer School, 2008.
11. K. R. M. Leino, A. Poetzsch-Heffter, and Y. Zhou. Using data groups to specify and check side effects. In *PLDI 2002*, pages 246–257. ACM Press, 2002.
12. B. Liskov and J. M. Wing. A behavioral notion of subtyping. *ACM Transactions on Programming Languages and Systems*, 16(6):1811–1841, 1994.
13. J. McCarthy. Towards a mathematical science of computation. In *Information Processing 1962*, pages 21–28, 1963.
14. B. Meyer. Applying "design by contract". *Computer*, 25(10):40–51, 1992.
15. P. Müller, A. Poetzsch-Heffter, and G. T. Leavens. Modular invariants for layered object structures. *Science of Computer Programming*, 62(3):253–286, 2006.
16. M. J. Parkinson and G. M. Bierman. Separation logic and abstraction. In *POPL 2005*, pages 247–258. ACM Press, 2005.
17. A. Poetzsch-Heffter. Specification and verification of object-oriented programs. Habilitationsschrift, Technische Universität München, 1997.
18. P. Rümmer. Sequential, parallel, and quantified updates of first-order structures. In *LPAR 2006*, LNCS 4246, pages 422–436. Springer, 2006.
19. P. H. Schmitt, M. Ulbrich, and B. Weiß. Dynamic frames in Java dynamic logic: Formalisation and proofs. Technical Report 2010-11, Department of Computer Science, Karlsruhe Institute of Technology, 2010.
20. J. Smans, B. Jacobs, and F. Piessens. Implicit dynamic frames: Combining dynamic frames and separation logic. In *ECOOP 2009*, LNCS 5653, pages 148–172. Springer, 2009.
21. J. Smans, B. Jacobs, F. Piessens, and W. Schulte. An automatic verifier for Java-like programs based on dynamic frames. In *FASE 2008*, LNCS 4961, pages 261–275. Springer, 2008.
22. K. Stenzel. A formally verified calculus for full Java Card. In *AMAST 2004*, volume 3116 of *LNCS*, pages 491–505. Springer, 2004.