Verification of Modifies Clauses in Dynamic Logic with Non-rigid Functions

Christian Engel[1], Andreas Roth[2], Peter H. Schmitt[1], and Benjamin Weiß[1]

[1] Institute for Theoretical Computer Science
University of Karlsruhe, D-76128 Karlsruhe, Germany
`{engelc,pschmitt,bweiss}@ira.uka.de`
[2] SAP Research CEC Darmstadt, D-64283 Darmstadt, Germany

Abstract. For modular verification of object-oriented programs, it is necessary to constrain *what* may be changed by a method in addition to *how* it is changed. Doing so with the classical means of pre- and postconditions is cumbersome, and even impossible if the program context is not entirely known. Therefore, specifications make use of an additional construct, known as a "frame property" or "modifies clause", which lists the memory locations that can at most be modified. Deductively verifying the correctness of such modifies clauses is difficult because the focus is on those locations which are *not* mentioned in the modifies clause. We present a novel approach to encode the correctness of modifies clauses as compact and readable proof obligations in dynamic logic. These proof obligations can be discharged efficiently with existing dynamic logic calculi, such as the one implemented in the KeY verification system. Additionally, we describe how a variant of our technique can be used for the verification of loops.

1 Introduction

The classical ingredients of the design by contract paradigm, pre- and postconditions and class invariants, are well known, [19]. Less well known but equally essential for concise specification and efficient reasoning is information on what does not change during program execution. In different specification languages this goes by different names, such as modifies clause or modifier set, assignable clause [16], frame property [8], or change information [7]. In Morgan's book [20] change information is even part of the basic definition in its theory of programs.

Once specifications include such modifies clauses, we need to verify their correctness, i.e., prove that a program indeed does not change any location outside the specified scope. The straightforward approach that inspects every single location of the program contradicts the intention behind using modifies clauses in the first place; it produces huge proof obligations, and it is non-modular in the sense that even changes in completely unrelated parts of the program can lead to changes in the proof obligation. We present a novel approach for verifying modifies clauses in the framework of dynamic logic [13, 5] without these disadvantages. Additionally, we show how this approach can be used for the verification of loops. A first version of the results of this paper is contained in [22].

The approach has been fully implemented in the program verification system KeY [4, 1] and successfully tested on a number of examples. KeY is a theorem prover for dynamic logic which allows the deductive verification of sequential Java programs with respect to specifications written e.g. in the Java Modeling Language, JML [16]. KeY features both powerful automation and support for interactive proving.

Related work A large number of related techniques and tools for checking modifies clauses have been proposed. The ChAsE tool [11] performs a lightweight syntactical analysis, which is efficient but unsound. Spoto and Poll [26] describe a sound technique based on abstract interpretation. Another static analysis method, which involves building and analysing a graph-based representation of the relevant parts of the heap, is proposed by Sălcianu and Rinard [27] and made more modular by Barnett et al [3].

Techniques for the verification of modifies clauses are also employed by other tools for object-oriented program verification. ESC/Java2 [12] creates for every assignment in a method a separate predicate logic verification condition. These verification conditions express that the assignment does not affect other locations than those in the method's modifies clause. This is a more restrictive interpretation of modifies clauses than ours, as temporary modifications to other locations are not tolerated, even though such modifications are not observable for clients. In the Boogie verifier [2], modifies clauses are translated into postconditions which use quantification over field names to express that all locations except those in the modifies clause have not been changed. Krakatoa [18] avoids the need for such quantifications in its proof obligations by first overapproximating the fields which are potentially affected in a pre-processing step, and then only including these fields in the generated postcondition.

Our technique is unique in that it expresses modifies clause correctness directly as a formula on the source code level. This formula can be used as the input of a program logic theorem prover. In contrast, ESC/Java2, Boogie and Krakatoa all handle modifies clause correctness on the level of intermediate representations, generate predicate logic verification conditions from there on, and only then invoke a theorem prover. Our approach is advantageous particularly in cases where automatic proving fails, since the familiarity of the source code eases interactive verification. To our knowledge none of the other approaches employs modifies clauses for loop verification in a comparable way.

Running example Throughout this paper we use the Java method `foo` given in Fig. 1 for illustration. This example has been introduced in [26]. It demonstrates that due to possible aliasing, checking the correctness of modifies clauses is nontrivial, even for a simple program comprising just a short sequence of assignments like this one. In particular, while a modifies clause consisting of `this.next`, `b.next.next` and `this.i` may at first seem correct for `foo`, it is not: if `this` and `b` refer to the same object, a forbidden modification to `b.next.next.next` (which in this case is the same as `this.next.next.next`) occurs in line 6. A correct modifies clause is contained in Fig. 1 as a comment in JML syntax.

Java + JML
```
1   class MyClass {
2     MyClass next;
3     int i;
4     //@ assignable next, b.next.next, b.next.next.next, i;
5     void foo(MyClass b) {
6       next = b.next.next;
7       b.next.next = b;
8       i++;
9     }
10  }
```
Java + JML

Fig. 1. The example method and modifies clause used throughout the paper.

Outline Sect. 2 reviews quickly the needed background information on dynamic logic. Sect. 3 introduces our formalism for representing modifies clauses. Many program logics use some concept of a *state transformer*; this role is played in our approach by *updates* explained in Sect. 4. The core contributions of this paper are Def. 11 and Theorem 1 in Sect. 5. Sect. 6 contains a patient walkthrough of the verification of our running example. The application of the approach to invariant based loop verification is discussed in Sect. 7, and conclusions in Sect. 8 round off the presentation. App. A contains the proof of the main theorem, and App. B sketches how object creation can be handled in our setting.

2 Dynamic Logic with Non-rigid Functions

First-order dynamic logic (DL) extends first-order predicate logic by a modality $[p]$ for every program p of some imperative programming language: for any formula ψ, the formula $[p]\psi$ expresses that if p terminates in a state s, then s satisfies ψ. A Hoare triple $\{\varphi\}p\{\psi\}$ [14] can thus be expressed as $\varphi \to [p]\psi$. Unlike Hoare logic, dynamic logic is closed under the standard logical operators as well as the modal operators $[p]$.

The semantic domain of DL formulas are *Kripke structures*. A Kripke structure is a pair (S, ρ), where S is the set of all possible program *states* and ρ is a function associating with each program p a *transition relation* $\rho(p) \subseteq S^2$ such that $(s_1, s_2) \in \rho(p)$ iff executing p in s_1 leads to s_2. The states themselves are first-order structures which all share the same universe $\mathbf{U}$. Given a state s and a variable assignment β, the evaluation of terms to values (i.e., elements of $\mathbf{U}$) by a function $val_{s,\beta}$ and the validity of first-order formulas $s, \beta \models \varphi$ is defined in the standard way. The semantics of the modal operators $[p]$ is defined as

$$s, \beta \models [p]\psi \quad iff \quad s', \beta \models \psi \;\; for\ all\ s'\ with\ (s, s') \in \rho(p)$$

A DL formula φ is called *valid* if $s, \beta \models \varphi$ for all states s of every Kripke structure and all variable assignments β.

We model program memory as *non-rigid* function symbols, i.e., symbols whose interpretation may vary between states. Local program variables are represented as non-rigid constant symbols, and object attributes as unary non-rigid function symbols, whose argument is the object for which the attribute is to be accessed. Similarly, array slots are modelled as a single binary non-rigid function symbol, whose arguments are the array object itself and the index into the array. Non-rigid function symbols representing program memory are more specifically called *location function symbols* to distinguish them from other non-rigid symbols (e.g., symbols modelling query methods). In contrast to non-rigid symbols in general, *rigid* symbols (e.g., arithmetic operators) are interpreted in the same way in all states.

Compared with an explicit modelling of stack and heap, using non-rigid symbols achieves a higher level of abstraction, which improves readability and helps with handling aliasing [5]. It excludes pointer arithmetic, though. We do not impose further assumptions on the programming language here, except that its programs can be modelled in the described way based on Kripke structures and non-rigid symbols. A detailed description of dynamic logic with non-rigid symbols for a minimalist object-oriented language is contained in [5]; a full-grown version for Java programs is defined in [4, Chapt. 3] and implemented in the KeY tool. We conclude this section with an example for a formula in the latter:

$$\texttt{self.i} \doteq i' \rightarrow [\texttt{self.foo(b);}]\texttt{self.i} \doteq i' + 1$$

Here, `self` and `b` are local program variables, and i' is a rigid constant symbol. Instead of the usual predicate logic notation `i(self)`, this term is written as `self.i` in order to resemble the equivalent Java expression. Overall, the formula expresses that our example method `foo` from Fig. 1 increments the attribute `i` of the receiver object by one.

3 Modifies Clauses

In this section, we formally define the concepts of (memory) locations and modifies clauses in the context of dynamic logic as described in Sect. 2.

Definition 1 (Locations). *A* location *is a tuple $(f, \bar{v})$, where f is a location function symbol with arity n, and where $\bar{v}$ (with $\bar{v} := v_1, \ldots, v_n$) are values.*

Locations are the units of state change. An example location in the program of Fig. 1 is $(\texttt{i}, o)$, where $o \in \mathbf{U}$ is some Java object. Given a state s, the value stored in $(\texttt{i}, o)$ is $s(\texttt{i})(o)$. Locations are special in that they consist both of a syntactical symbol f and semantical elements $\bar{v}$. A completely syntactical description of locations is possible using *location descriptors*, which form the elements of *modifies clauses*.

Definition 2 (Modifies clauses). *A* location descriptor *is a construct*

$$\textit{for } \bar{x};\ \varphi;\ f(\bar{t})$$

where $\bar{x}$ (with $\bar{x} := x_1, \ldots, x_m$) are logical variables (bound in $\varphi, \bar{t}$), φ is a formula, f is a location function symbol with arity n, and $\bar{t}$ (with $\bar{t} := t_1, \ldots, t_n$) are terms. Location descriptors must not contain free logical variables. A modifies clause *is a finite set of location descriptors.*

A location descriptor of the form *for* ; *true*; $f(\bar{t})$ can be abbreviated as $f(\bar{t})$. The following modifies clause corresponds to the `assignable` comment in Fig. 1:

$$\{\texttt{self.next},\ \texttt{b.next.next},\ \texttt{b.next.next.next},\ \texttt{self.i}\} \tag{1}$$

As a second example, the location descriptor (*for* x; $0 \leq x < \texttt{a.length}$; $\texttt{a}[x]$) describes all slots of the array object referenced by the program variable `a`.

Definition 3 (Semantics of modifies clauses). *A location descriptor $ld :=$ (for $\bar{x}$; φ; $f(\bar{t})$) is evaluated to a set of locations:*

$$val_{s,\beta}(ld) = \bigcup_{*} \{(f, val_{s,\beta'}(\bar{t}))\}$$

where the union $*$ is over all β' for which there exist values $\bar{v}$ such that $\beta' = \beta_{\bar{x}}^{\bar{v}}$ and $s, \beta' \models \varphi$. For a modifies clause mod, its evaluation is defined as $val_{s,\beta}(mod) = \bigcup_{ld \in mod} val_{s,\beta}(ld)$.

Here and throughout the paper, we take some liberty regarding the notation of tuples (such as $\bar{t}$) and single elements (for which $val_{s,\beta'}$ is actually defined). We trust that this does not cause any ambiguity. As usual, $\beta_{\bar{x}}^{\bar{v}}$ denotes the variable assignment which is identical to β except that $\beta_{\bar{x}}^{\bar{v}}(\bar{x}) = \bar{v}$. An example for the above definition is $val_{s,\beta}(\{\texttt{self.next}\}) = \{(\texttt{next}, val_{s,\beta}(\texttt{self}))\}$.

We now define what it means for a program (such as a method) to *respect* a modifies clause: all heap locations which after executing the program hold a different value than before must be mentioned in the modifies clause.

Definition 4 (Respecting modifies clauses). *A program p* respects *a modifies clause mod under the precondition φ iff for all Kripke structures (S, ρ) and for all pairs of states $(s_{pre}, s_{post}) \in \rho(p)$, all variable assignments β, all location function symbols f not representing local program variables, and all value tuples $\bar{v}$ (with $\bar{v} := v_1, \ldots, v_n$, where n is the arity of f) the following implication holds:*

$$\begin{gathered} s_{pre}, \beta \models \varphi \quad \textit{and} \quad s_{pre}(f)(\bar{v}) \neq s_{post}(f)(\bar{v}) \\ \textit{together imply} \\ (f, \bar{v}) \in val_{s_{pre},\beta}(mod) \end{gathered}$$

We do not impose restrictions on modifications of local variables, because these are invisible to the caller of a method. Note that modifies clauses are always evaluated in s_{pre}, i.e., the pre-state *before* program execution.

Besides constraining the locations which may be *modified* by a program, modifies clauses can also be used to describe the locations on which something *depends*. We now define a class of non-rigid symbols which have the same interpretation in all states where the locations of a modifies clause have the same values.

Definition 5 (Location dependent symbols, [9, 10]). *A location dependent predicate symbol is a non-rigid predicate symbol parametrised by a modifies clause, denoted as* $P[mod]$*, such that for every Kripke structure* (S, ρ)*, for every two states* $s, s' \in S$*, and for every variable assignment* β *the following implication is true:*

$$\begin{gathered} val_{s,\beta}(mod) = val_{s',\beta}(mod) \\ \textit{and} \\ \textit{for all } (f, \bar{v}) \in val_{s,\beta}(mod): \; s(f)(\bar{v}) = s'(f)(\bar{v}) \\ \textit{together imply} \\ s(P[mod]) = s'(P[mod]) \end{gathered}$$

For such symbols, it is useful to allow a special modifies clause denoted as $*$, which stands for the whole heap. Its semantics is the same as that of the set made up by location descriptors ($\textit{for}\ \bar{x};\ \textit{true};\ f(\bar{x})$) for every location function symbol f which is not a local program variable; unlike this set, it does not require knowing the entire program context in advance. Location dependent predicate symbols have a variety of uses. For example, reachability between objects can be encoded using a symbol $reach[mod]$, where mod describes the fields which may occur in the reference chain. More importantly here, they are central to our approach for reasoning about modifies clause satisfaction, which is described in Sect. 5.

4 State Updates

Besides location dependent predicate symbols, the second major ingredient to our approach for verifying modifies clauses is an extension to dynamic logic called *updates* [24], which can be seen as a generalisation of syntactic substitutions.

Definition 6 (Updates). *An* update u *is a construct of one of the following two forms:*

- $u = (\textit{for}\ \bar{x};\ \varphi;\ f(\bar{t}) := t')$*, where* $\bar{x}$ *(with* $\bar{x} := x_1, \ldots, x_m$*) are logical variables (bound in* $\varphi, \bar{t}, t'$*),* f *is a location function symbol with arity* n*, and* $\bar{t}$ *(with* $\bar{t} := t_1, \ldots, t_n$*) and* t' *are terms; or*
- $u = (u_1 \mid u_2)$*, where* u_1 *and* u_2 *are updates.*

For every update u*, term* t*, formula* ψ*,* $\{u\}t$ *is a term and* $\{u\}\psi$ *is a formula.*

Intuitively, an update of the first kind changes the interpretation of the function symbol f for the arguments described by $\bar{t}$ to the value of t', for all assignments of values to the variables $\bar{x}$ which satisfy φ. An update of the form *for*; *true*; $f(\bar{t}) := t'$ can be abbreviated as $f(\bar{t}) := t'$. An update $u_1 \mid u_2$ intuitively means parallel execution of u_1 and u_2. In a term $\{u\}t$ or a formula $\{u\}\varphi$, the subterm t and the subformula φ are evaluated in the post-state of the update u. A more formal definition of update semantics follows, which builds on the concept of locations introduced in Def. 1 to first define *semantic updates*, and then an evaluation of (syntactic) updates to such semantic updates.

Definition 7 (Semantic updates). *A* semantic update *is a set of pairs (l, v), where l is a location and v a value, such that for no l the set contains (l, v_1) and (l, v_2) where $v_1 \neq v_2$. A semantic update U can be seen as a function on states, where for each state s the output state $U(s)$ is partially defined by*

$$U(s)(f)(\bar{v}) := \begin{cases} v & \textit{if } ((f,\bar{v}),v) \in U \\ s(f)(\bar{v}) & \textit{otherwise} \end{cases}$$

for all location function symbols f and values $\bar{v}$ (with $\bar{v} := v_1, \ldots, v_n$, where n is the arity of f).

Note that nothing is defined about the post-update interpretation of non-rigid symbols which are not location function symbols. In particular, the interpretation of location dependent predicate symbols may be affected in an unknown way, except for the restriction imposed in Def. 5.

Definition 8 (Semantics of updates). *An update u is evaluated to a semantic update:*

- *If $u = (\mathit{for}\ \bar{x};\ \varphi;\ f(\bar{t}) := t')$:*

$$val_{s,\beta}(u) := win\Big(\bigcup_{*}\{((f, val_{s,\beta'}(\bar{t})), val_{s,\beta'}(t'))\}\Big)$$

  where the union $*$ is over all β' for which there exist values $\bar{v}$ such that $\beta' = \beta_{\bar{x}}^{\bar{v}}$ and $s, \beta' \models \varphi$. The function win ensures that every location is mapped to at most one value. This is achieved by imposing a well-ordering on the set of values, and choosing the smallest tuple $\bar{v}$ in case a clash occurs (for details refer to [24]).
- *If $u = (u_1 \mid u_2)$:*

$$val_{s,\beta}(u) := (U_1 \cup U_2) \setminus C$$

  where $U_1 := val_{s,\beta}(u_1)$, $U_2 := val_{s,\beta}(u_2)$, and where $C := \{((f,\bar{v}),v) \in U_1 \mid ((f,\bar{v}),v') \in U_2 \text{ for some } v' \neq v\}$

The semantics of a term $\{u\}t$ is defined as $val_{s,\beta}(\{u\}t) := val_{s',\beta}(t)$, where s' is the state resulting form applying u to s, i.e., $s' := val_{s,\beta}(u)(s)$. For a formula $\{u\}\psi$, $(s,\beta) \models \{u\}\psi$ is defined to hold iff $(s',\beta) \models \psi$, where again $s' := val_{s,\beta}(u)(s)$.

The similarity between location descriptors and updates is *not* coincidental; every modifies clause has a canonical representation as an update, called an *anonymising update* for the modifies clause.

Definition 9 (Anonymising updates). *An* anonymising update *for a location descriptor $ld := (for\ \bar{x};\ \varphi;\ f(\bar{t}))$ is an update*

$$\mathcal{V}(ld) := (for\ \bar{x};\ \varphi;\ f(\bar{t}) := f'(\bar{t}))$$

where f' is a fresh rigid function symbol with the same arity as f. An anonymising update for a modifies clause $mod = \{ld_1, \ldots, ld_k\}$ is an update $\mathcal{V}(mod) := \mathcal{V}(ld_1) \mid \cdots \mid \mathcal{V}(ld_k)$ (in an arbitrary order).

Note that anonymising updates are not unique, and specific to a particular situation during proof construction, as the function symbols f' must be "fresh" in the sense of Skolem constants. An anonymising update assigns unknown values to all locations described by the modifies clause. It can be used, for example, to reason about a method call using a method contract: if the method respects the modifies clause of the contract, then an anonymising update for the modifies clause provides a worst case approximation of the method's behaviour (similar to the `havoc` statements in ESC/Java and Boogie). The following is an anonymising update for the modifies clause (1) of our running example:

$$\begin{aligned} &\texttt{self.next} := \mathit{next}'(\texttt{self}) \ \mid\ \texttt{b.next.next} := \mathit{next}'(\texttt{b.next}) \\ \mid\ &\texttt{b.next.next.next} := \mathit{next}'(\texttt{b.next.next}) \ \mid\ \texttt{self.i} := i'(\texttt{self}) \end{aligned}$$

5 Verification of Modifies Clauses

Proving that a method implementation respects a modifies clause according to Def. 4 is difficult because the focus is on those locations which are *not* mentioned in the modifies clause: for all those, we have to show that they are not modified by a call to the method. A straightforward encoding into proof obligations where all these locations are listed explicitly is possible, but undesirable: it would negate the original motivation for introducing modifies clauses, which was precisely to avoid having to reason about these locations explicitly. Furthermore the approach would be non-modular, as even a slight extension of the program context would change the proof obligation.

In the following we present a different approach, which produces compact and readable proof obligations that remain unchanged if the program context is extended. The basic idea is to construct a formula ψ whose interpretation depends exactly on the heap locations not contained in the modifies clause. If the validity of this ψ in the pre-state of the considered method implies its validity in the corresponding post-state, then we know that the method respects the modifies clause. That is, our proof obligation to be verified mechanically then is simply:

$$\psi \rightarrow [p]\psi \tag{2}$$

How can we construct such a formula ψ? Simply using the "complement" of the modifies clause as parametrisation of a location dependent predicate symbol (Def. 5) is not an option, because this would again negate the advantages of using modifies clauses in the first place. However, we can combine a predicate symbol $P[*]$ depending on the whole heap with an anonymising update $\mathcal{V} := \mathcal{V}(\mathit{mod})$ to get:

$$\psi := \{\mathcal{V}\}P[*]$$

This choice of ψ has the desired property: since the anonymising update sets all locations in the modifies clause to unknown but fixed values, differences in the interpretation of these locations do not affect the interpretation of the occurrence of the predicate symbol.

Before we can formalise this approach in Def. 11, we have to solve one more problem: since the second occurrence of ψ in (2) is in the scope of the modal operator $[p]$, the anonymising update $\mathcal{V}$ has a different effect here than in the first occurrence of ψ, as it is wrongly evaluated in the post-state of p. For our proof obligation to be correct, we need to make the interpretation of the anonymising update state independent. This is achieved with the transformation *pre* defined below.

Definition 10 (pre transformation). *Let $ld := \mathit{for}\ \bar{x};\ \varphi;\ f(\bar{t})$ be a location descriptor. We define the transformation pre as follows:*

$$\mathit{pre}(ld) := \mathit{for}\ \bar{x};\ q(\bar{x});\ f(\bar{g}(\bar{x}))$$

where q is a fresh rigid predicate symbol, and $\bar{g}$ (with $\bar{g} := g_1, \ldots, g_n$, where n is the arity of f) are fresh rigid function symbols. For a modifies clause mod, we define $\mathit{pre}(\mathit{mod}) := \{\mathit{pre}(ld) \mid ld \in \mathit{mod}\}$. The formula $\mathit{DefPre}(\mathit{mod})$ can be used to axiomatise the symbols q and $\bar{g}$:

$$\mathit{DefPre}(\mathit{mod}) := \bigwedge\{\forall\bar{x}.\big((q(\bar{x}) \leftrightarrow \varphi) \land \bar{g}(\bar{x}) \doteq \bar{t}\big) \mid (\mathit{for}\ \bar{x};\ \varphi;\ f(\bar{t})) \in \mathit{mod}\}$$

Definition 11 (RespectsModifies proof obligation). *For a modifies clause mod, a formula φ, and a program p, the formula $\mathit{RespectsModifies}(p, \mathit{mod}, \varphi)$ is defined as follows:*

$$\mathit{RespectsModifies}(p, \mathit{mod}, \varphi) := \varphi \land \mathit{DefPre} \land \{\mathcal{V}^{pre}\}P[*] \rightarrow [p]\{\mathcal{V}^{pre}\}P[*]$$

where $\mathcal{V}^{pre} := \mathcal{V}(\mathit{mod}^{pre})$, $\mathit{mod}^{pre} := \mathit{pre}(\mathit{mod})$, $\mathit{DefPre} := \mathit{DefPre}(\mathit{mod})$, and where $P[*]$ is a fresh location dependent predicate symbol.

Theorem 1. *For any modifies clause mod, any formula φ, and any program p, the formula $\mathit{RespectsModifies}(p, \mathit{mod}, \varphi)$ is valid iff p respects mod under the precondition φ.*

A proof of Theorem 1, which states that the formula *RespectsModifies* captures exactly the desired property (established in Def. 4), is given in App. A. The validity of *RespectsModifies* formulas can be proven in a largely automated fashion with dynamic logic calculi such as the ones defined in [5] and [4, Chapt. 3]. In Sect. 6 we sketch such a proof for our running example.

6 Example

Instantiating *RespectsModifies* (Def. 11) for the running example (Fig. 1) yields:

$$\varphi \wedge \mathit{DefPre} \wedge \{\mathcal{V}^{pre}\}P[*] \rightarrow [\texttt{self.foo(b);}]\{\mathcal{V}^{pre}\}P[*] \tag{3}$$

$$\begin{aligned}\text{with:}\quad \mathit{DefPre} &= \big(g_1 \doteq \texttt{self} \wedge g_2 \doteq \texttt{b.next} \wedge\ g_3 \doteq \texttt{b.next.next}\big)\\ \mathcal{V}^{pre} &= \big(g_1\texttt{.next} := \mathit{next}'(g_1) \mid g_2\texttt{.next} := \mathit{next}'(g_2)\\ &\quad \mid g_3\texttt{.next} := \mathit{next}'(g_3) \mid g_1\texttt{.i} := i'(g_1)\big)\end{aligned}$$

where `self` and `b` are local program variables, and where $g_1, g_2, g_3, \mathit{next}', i'$ are rigid function symbols. The formula φ may contain preconditions and class invariants, stating for example that the occurring variables and fields do not contain the value `null`. However, these conditions are not relevant for us in the following walkthrough of the proof, as we ignore exceptional behaviour and focus only on the normal execution path of `foo`.

We go about proving the validity of (3) by iteratively rewriting the formula until we arrive at something which is obviously valid, in the style of a sequent calculus (for details on the calculus see [4, Chapt. 3]). The critical part of (3) is the modal operator [`self.foo(b);`], which we handle by rules whose effect can be understood as *symbolic execution*: the program is traversed in a forward manner, using logical symbols instead of concrete values for the memory locations. Symbolic execution first unfolds the call to `foo` by inserting the method's body, and then tackles the assignments by converting them into updates, leading to the formula:

$$\varphi \wedge \mathit{DefPre} \wedge \{\mathcal{V}^{pre}\}P[*] \rightarrow \{u_1\}\{u_2\}\{u_3\}\{\mathcal{V}^{pre}\}P[*] \tag{4}$$

$$\begin{aligned}\text{with:}\quad u_1 &= (\texttt{self.next} := \texttt{b.next.next})\\ u_2 &= (\texttt{b.next.next} := \texttt{b})\\ u_3 &= (\texttt{self.i} := \texttt{self.i} + 1)\end{aligned}$$

We now transform the update sequence $\{u_1\}\{u_2\}\{u_3\}\{\mathcal{V}^{pre}\}$ into a normal form by applying update rewriting rules [24, 23]. The first step is to combine $\{u_1\}\{u_2\}$ into a single update $\{u_{12}\}$:

$$\begin{aligned}&\{u_1\}\{u_2\}\\ \rightsquigarrow\ &\{u_1 \mid (\{u_1\}(\texttt{b.next}))\texttt{.next} := \{u_1\}\texttt{b}\}\\ \rightsquigarrow\ &\{u_1 \mid \big(\mathit{if}((\{u_1\}\texttt{b}) \doteq \texttt{self})\mathit{then}(\texttt{b.next.next})\mathit{else}(\{u_1\}\texttt{b})\big)\texttt{.next} := \texttt{b}\}\\ \rightsquigarrow\ &\{\underbrace{u_1 \mid \big(\mathit{if}(\texttt{b} \doteq \texttt{self})\mathit{then}(\texttt{b.next.next})\mathit{else}(\texttt{b})\big)\texttt{.next} := \texttt{b}}_{u_{12}}\}\end{aligned}$$

Basically, the sequential execution $\{u_1\}\{u_2\}$ is converted into a parallel execution $\{u_1 \mid u_2'\}$, where u_2' is the result of applying the substitution expressed by u_1

to u_2. Applying this substitution proceeds by recursively shifting $\{u_1\}$ from a term to its subterms, thereby replacing the term's operator if it is affected by the update. As an exception, the top-level operator of the left hand side of u_2 is never affected, because it specifies the symbol to be updated by u_2, and not a value. While it is obvious that $\{u_1\}$ cannot affect `b`, it is not possible to determine syntactically whether $\{u_1\}$ affects `b.next`: this depends on whether `b` and `self` are aliased or not. Therefore, a syntactical case distinction in the form of an *if-then-else* term is introduced.

Based on the same principles, we can merge $\{u_{12}\}\{u_3\}$ into $\{u_{123}\}$:

$$\begin{aligned}\{u_{12}\}\{u_3\} &\rightsquigarrow \{u_{12} \mid (\{u_{12}\}\texttt{self}).\texttt{i} := \{u_{12}\}(\texttt{self.i} + 1)\}\\ &\rightsquigarrow \{u_{12} \mid \texttt{self.i} := (\{u_{12}\}(\texttt{self.i})) + \{u_{12}\}1\}\\ &\rightsquigarrow \{u_{12} \mid \texttt{self.i} := (\{u_{12}\}\texttt{self}).\texttt{i} + 1\}\\ &\rightsquigarrow \{\underbrace{u_{12} \mid \texttt{self.i} := \texttt{self.i} + 1}_{u_{123}}\}\end{aligned}$$

Since $\{u_{12}\}$ cannot affect $\{u_3\}$ at all, $\{u_{123}\}$ is equal to $\{u_{12} \mid u_3\}$. For the same reason, the result of combining $\{u_{123}\}\{\mathcal{V}^{pre}\}$ is simply $\{u_{123} \mid \mathcal{V}^{pre}\}$:

$$\{u_{123}\}\{\mathcal{V}^{pre}\} \quad \rightsquigarrow \quad \{u_{123} \mid \mathcal{V}^{pre}\}$$

Now, there is no modal operator or update left which separates *DefPre* and $\mathcal{V}^{pre}$. This allows us to use the equations in *DefPre* to rewrite g_1 to `self`, g_2 to `b.next`, and g_3 to `b.next.next` in $\mathcal{V}^{pre}$. After these rewriting steps, the left hand sides of the atomic updates in u_{123} all occur in $\mathcal{V}^{pre}$ as well. This means that the effect of u_{123} is completely overridden by $\mathcal{V}^{pre}$ (because the rightermost update "wins" in case of a clash, Def. 8), and we can safely get rid of u_{123}:

$$\{u_{123} \mid \mathcal{V}^{pre}\} \quad \rightsquigarrow \quad \{\mathcal{V}^{pre}\}$$

Overall, we have now transformed (4) into:

$$\varphi \wedge \mathit{DefPre} \wedge \{\mathcal{V}^{pre}\}P[*] \rightarrow \{\mathcal{V}^{pre}\}P[*] \tag{5}$$

Since (5) is obviously valid, we have successfully finished our proof of (3). The implementation in the KeY tool performs the described steps automatically within a few seconds.

As the example shows, verifying *RespectsModifies* amounts to proving that the following two updates are equivalent with respect to $P[*]$: (1) the anonymising update $\mathcal{V}^{pre}$ on the left side of the implication, and (2) an update $\{u\}\{\mathcal{V}^{pre}\}$ resulting from symbolic execution of p on the right side of the implication. In this example and many other cases, the update rewriting rules are strong enough to make this equivalence show as *syntactical* equality of the updates. If this is not the case, the problem of proving update equivalence can be reduced to the problem of proving a first-order formula: for showing the equivalence of $\mathcal{V}^{pre}$ and $\{u\}\{V^{pre}\}$ with respect to $P[*]$, it is sufficient to show the validity of formulas

$$\forall \bar{x}.\{\mathcal{V}^{pre}\}f(\bar{x}) \doteq \{u\}\{\mathcal{V}^{pre}\}f(\bar{x})$$

for all function symbols f not representing local program variables which occur as the top-level operator on the left hand side in one of the involved updates. The updates occurring in these formulas can always be transformed away.

7 Modifies Clauses for Loops

Besides constraining the locations which may be modified by a *method*, another useful application of modifies clauses—and thus of our technique for ensuring their correctness—lies in the verification of *loops* with the help of *invariants*. The classical loop invariant rule in dynamic logic with updates looks as follows (using sequent calculus notation):

$$\frac{\begin{array}{c}\Gamma \;\Rightarrow\; \{u\}\mathit{Inv},\; \Delta \\ \mathit{Inv},\; \mathtt{e} \;\Rightarrow\; [\mathtt{p}]\mathit{Inv} \\ \mathit{Inv},\; \neg\mathtt{e} \;\Rightarrow\; [\omega]\psi\end{array}}{\Gamma \;\Rightarrow\; \{u\}[\mathtt{while(e)\ p;}\ \omega]\psi,\; \Delta}$$

The first premise of this rule expresses that the invariant *Inv* holds before the loop, the second that it is preserved by the loop body p, and the third that *if* it holds after the loop, *then* the remaining program ω establishes the postcondition ψ. Together, the validity of these three premises implies the validity of the rule's conclusion, namely that after executing the loop and subsequently ω, the postcondition ψ is satisfied. The initial states in which the loop is to be considered are described by the sets of formulas Γ and Δ, and by the update u.

This rule has the disadvantage that the second and third premise do not contain the context information encoded in Γ, Δ and u. The reason is that we cannot assume that this information still holds at the beginning of an arbitrary iteration of the loop, since it may have been invalidated by the previous iterations. This makes it necessary to put all information from the context that is preserved by the loop (and that is needed for further reasoning) into the loop invariant, which is very inconvenient. To overcome this deficiency, an improved loop invariant rule making use of modifies clauses has been proposed in [6]:

$$\frac{\begin{array}{l}\Gamma \;\Rightarrow\; \{u\}\mathit{Inv},\; \Delta \\ \boldsymbol{\Gamma} \;\Rightarrow\; \boldsymbol{\{u\}\{\mathcal{V}\}}(\mathit{Inv} \land \mathtt{e} \to [\mathtt{p}]\mathit{Inv}),\; \boldsymbol{\Delta} \\ \boldsymbol{\Gamma} \;\Rightarrow\; \boldsymbol{\{u\}\{\mathcal{V}\}}(\mathit{Inv} \land \neg\mathtt{e} \to [\omega]\psi),\; \boldsymbol{\Delta}\end{array}}{\Gamma \;\Rightarrow\; \{u\}[\mathtt{while(e)\ p;}\ \omega]\psi,\; \Delta}$$

A modifies clause *mod* for the loop is specified in addition to a loop invariant, and an anonymising update $\mathcal{V} := \mathcal{V}(\mathit{mod})$ is used to selectively "erase" only the necessary parts of the context. Since Γ, Δ and u are kept, it is now sufficient to encode in the invariant only properties concerning locations which are part of the modifies clause. However, this rule *assumes* that the given modifies clause is respected by the loop; it is unsound if this is not the case. Our technique allows to extend it so that this assumption is *proven*:

$$\frac{\begin{array}{l}\Gamma \;\Rightarrow\; \{u\}\mathit{Inv},\ \Delta \\ \Gamma \;\Rightarrow\; \{u\}\big(\boldsymbol{DefPre} \rightarrow \\ \qquad\qquad \{\mathcal{V}\}(\mathit{Inv} \wedge \mathtt{e} \wedge \boldsymbol{\{\mathcal{V}^{pre}\}P[*,\bar{v}]} \rightarrow [\mathtt{p}](\mathit{Inv} \wedge \boldsymbol{\{\mathcal{V}^{pre}\}P[*,\bar{v}]}))\big),\ \Delta \\ \Gamma \;\Rightarrow\; \{u\}\{\mathcal{V}\}(\mathit{Inv} \wedge \neg\mathtt{e} \rightarrow [\omega]\psi),\ \Delta\end{array}}{\Gamma \;\Rightarrow\; \{u\}[\mathtt{while(e)\ p;}\ \omega]\psi,\ \Delta}$$

The changed second premise states that in addition to *Inv*, the loop body also preserves $\{\mathcal{V}^{pre}\}P[*,\bar{v}]$. As local variables $\bar{v}$ declared in the part of the program before the loop can be changed by the loop and these changes are visible in the program ω, those local variables can consequently be part of the modifies clause. This also necessitates that we adjust our notion of the correctness of modifies clauses (see Def. 4) accordingly, and use the location dependent predicate $P[*,\bar{v}]$ which in addition to the heap also depends on the relevant local variables $\bar{v}$.

One particularity of this invariant rule is that we show that $\{\mathcal{V}^{pre}\}P[*,\bar{v}]$ is preserved by `p` if `p` is executed in a state in which we assumed that no other locations than the ones contained in *mod* have changed (which is what we actually want to prove by showing the preservation of $\{\mathcal{V}^{pre}\}P[*,\bar{v}]$).

On a first glance this might look like circular reasoning. This is, however, not the case which can be motivated by the following inductive argument: Before the first loop iteration no location was yet changed (by the loop) which trivially entails the induction hypothesis that in this state *mod* is a correct modifies clause for the preceding loop iterations. This forms our induction basis which does not need to be proven explicitly. In the second premise we do the induction step by proving: *If, before an arbitrary iteration of the loop, mod is a correct modifies clause for all the preceding iterations (this assumption is expressed by the anonymising update $\mathcal{V}$) then mod will also be a correct modifies clause for the next iteration (expressed by the preservation of $\{\mathcal{V}^{pre}\}P[*,\bar{v}]$).* Together with the induction basis this obviously implies that *mod* is a correct modifies clause for the entire loop.

Fig. 2 shows a Java loop which is specified by a loop invariant and a modifies clause (modifies clauses for loops are not currently part of JML, but they are supported by KeY). Since the `DataBase` object `d` cannot be changed by the loop according to its specification, the invariant merely needs to constrain the values of locations that *can* change, namely those of the local variables `i` and `result`. In other approaches, where no modifies clauses are used for loops, it is necessary to also encode in the invariant every property of `d` which can, for instance, affect the result of the call to `processQuery`.

8 Conclusions

We have presented an approach for the verification of modifies clauses in the framework of dynamic logic. In contrast to a straightforward encoding of modifies clause correctness into a dynamic logic formula, where all locations not in the

— Java + JML —

```
DataBase d = readDataBase();
int i = 0;
Object result = null;
//@ loop_invariant result == null && i >= 0;
//@ assignable result, i;
while(i < c){
    result = processQuery(i++, d);
    if(result != null) return result;
}
...
```

— Java + JML —

Fig. 2. A loop annotated with an invariant and a modifies clause.

modifies clause have to be considered explicitly, our approach leads to proof obligations that are compact in size and remain the same if the program context is changed or extended, which is important for making the verification modular. We have proven that our encoding expresses the desired property (App. A), and presented a useful variation for invariant based loop verification.

Our encoding is based on two extensions to classical dynamic logic, namely updates (in particular anonymising updates) and location dependent symbols. Both of these concepts have been developed independently of our technique [5, 24, 9, 10] and are useful for a variety of other purposes, e.g. for the verification of depends clauses [10].

The technique has been implemented as a part of the KeY tool, where the target language of the verification is Java, and where modifies clauses are e.g. expressed in JML. The implementation has been successfully applied to a number of examples, such as a Java Card implementation of the Mondex case study [25], for which all modifies clauses have been proven[3]. The degree of automation for these proofs was high.

For fully modular specification and verification, modifies clauses must support some form of data abstraction, such as data groups [17], ownership [21] or dynamic frames [15]. As an important line of future work, we intend to extend our approach with a data abstraction mechanism, based on our concept of location dependent symbols. We expect this extension to be mostly orthogonal to the results presented in this paper.

References

1. W. Ahrendt, T. Baar, B. Beckert, R. Bubel, M. Giese, R. Hähnle, W. Menzel, W. Mostowski, A. Roth, S. Schlager, and P. H. Schmitt. The KeY tool. *Software and System Modeling*, 4(1):32–54, 2005.

[3] Source code, specifications and proofs for this case study can be found here: `http://i12www.ira.uka.de/~engelc/mondex/mondex-mod.tgz`

2. M. Barnett, B.-Y. E. Chang, R. DeLine, B. Jacobs, and K. R. M. Leino. Boogie: A modular reusable verifier for object-oriented programs. In *4th International Symposium on Formal Methods for Components and Objects (FMCO 2005)*, LNCS 4111, pages 364–387. Springer, 2006.
3. M. Barnett, M. Fähndrich, D. Garbervetsky, and F. Logozzo. Annotations for (more) precise points-to analysis. In *International Workshop on Aliasing, Confinement and Ownership in Object-Oriented Programming (IWACO)*, 2007.
4. B. Beckert, R. Hähnle, and P. H. Schmitt, editors. *Verification of Object-Oriented Software: The KeY Approach*. LNCS 4334. Springer, 2007.
5. B. Beckert and A. Platzer. Dynamic logic with non-rigid functions: A basis for object-oriented program verification. In *3rd International Joint Conference on Automated Reasoning (IJCAR 2006)*, LNCS 4130, pages 266–280. Springer, 2006.
6. B. Beckert, S. Schlager, and P. H. Schmitt. An improved rule for while loops in deductive program verification. In *7th International Conference on Formal Engineering Methods (ICFEM 2005)*, LNCS 3785, pages 315–329. Springer, 2005.
7. B. Beckert and P. H. Schmitt. Program verification using change information. In *1st International Conference on Software Engineering and Formal Methods (SEFM 2003)*, pages 91–99. IEEE Press, 2003.
8. A. Borgida, J. Mylopoulos, and R. Reiter. On the frame problem in procedure specifications. *IEEE Transactions on Software Engineering*, 21(10):785–798, 1995.
9. R. Bubel. *Formal Verification of Recursive Predicates*. PhD thesis, University of Karlsruhe, 2007.
10. R. Bubel, R. Hähnle, and P. H. Schmitt. Specification predicates with explicit dependency information. In *5th International Verification Workshop (VERIFY'08)*, volume 372 of *CEUR Workshop Proceedings*, pages 28–43. CEUR-WS.org, 2008.
11. N. Cataño and M. Huisman. ChAsE: A static checker for JML's assignable clause. In *4th International Conference on Verification, Model Checking, and Abstract Interpretation (VMCAI 2003)*, LNCS 2575, pages 26–40. Springer, 2002.
12. D. R. Cok and J. R. Kiniry. ESC/Java2: Uniting ESC/Java and JML. In *International Workshop on Construction and Analysis of Safe, Secure and Interoperable Smart devices (CASSIS 2004)*, LNCS 3362, pages 108–128. Springer, 2005.
13. D. Harel, D. Kozen, and J. Tiuryn. *Dynamic Logic*. MIT Press, 2000.
14. C. A. R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–580, 1969.
15. I. T. Kassios. Dynamic frames: Support for framing, dependencies and sharing without restrictions. In *14th International Symposium on Formal Methods (FM 2006)*, LNCS 4085, pages 268–283. Springer, 2006.
16. G. T. Leavens, A. L. Baker, and C. Ruby. Preliminary design of JML: A behavioral interface specification language for Java. *SIGSOFT Software Engineering Notes*, 31(3):1–38, 2006.
17. K. R. M. Leino. Data groups: Specifying the modification of extended state. In *13th ACM Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA'98)*, pages 144–153. ACM Press, 1998.
18. C. Marché and C. Paulin-Mohring. Reasoning about Java programs with aliasing and frame conditions. In *18th International Conference on Theorem Proving in Higher Order Logics (TPHOLs 2005)*, LNCS 3603, pages 179–194. Springer, 2005.
19. B. Meyer. Applying 'Design by Contract'. *Computer*, 25(10):40–51, 1992.
20. C. Morgan. *Programming from specifications*. Prentice-Hall, 1990.
21. P. Müller. *Modular Specification and Verification of Object-Oriented Programs*. LNCS 2262. Springer, 2002.

22. A. Roth. *Specification and Verification of Object-Oriented Software Components.* PhD thesis, University of Karlsruhe, 2006.
23. P. Rümmer. Proving and disproving in dynamic logic for Java. Licentiate Thesis 2006–26L, Chalmers University of Technology, 2006.
24. P. Rümmer. Sequential, parallel, and quantified updates of first-order structures. In *13th International Conference on Logic for Programming, Artificial Intelligence and Reasoning (LPAR 2006)*, LNCS 4246, pages 422–436. Springer, 2006.
25. P. H. Schmitt and I. Tonin. Verifying the Mondex case study. In *5th IEEE International Conference on Software Engineering and Formal Methods (SEFM 2007)*, pages 47–56. IEEE Press, 2007.
26. F. Spoto and E. Poll. Static analysis for JML's assignable clauses. In *10th Int. Workshop on Foundations of Object-Oriented Languages (FOOL-10)*, 2003.
27. A. D. Sălcianu and M. C. Rinard. Purity and side effect analysis for Java programs. In *6th International Conference on Verification, Model Checking and Abstract Interpretation (VMCAI 2005)*, LNCS 3385, pages 199–215. Springer, 2005.

A Proof of Soundness and Completeness

Let *mod* be a modifies clause, φ a formula, p a program, and let mod^{pre}, $\mathcal{V}^{pre}$, *DefPre*, $P[*]$ be chosen as in Def. 11. Before we start to prove Theorem 1, we need some helper lemmas. The correctness of these lemmas is rather obvious, so we refrain from giving a formal proof for them.

Lemma 1. *For all states s and all variable assignments β with $s, \beta \models DefPre$:*

$$val_{s,\beta}(mod^{pre}) = val_{s,\beta}(mod).$$

This lemma is a direct consequence of Def. 10: *DefPre* is chosen so that it links mod^{pre} and mod.

Lemma 2. *For all locations l, all s, β: $l \in val_{s,\beta}(mod^{pre})$ iff there is a $v \in \mathbf{U}$ with $(l, v) \in val_{s,\beta}(\mathcal{V}^{pre})$.*

This lemma follows directly from Defs. 3, 8 and 9: the anonymising update for mod^{pre} affects exactly the locations in mod^{pre}.

Lemma 3. *Let (S, ρ) be a Kripke structure. Then for all $s_1, s_2 \in S$ and all β:*

$$val_{s_1,\beta}(\mathcal{V}^{pre}) = val_{s_2,\beta}(\mathcal{V}^{pre}).$$

Lemma 3 holds since $\mathcal{V}^{pre}$ contains only rigid symbols, with the exception of the top level symbols on its left-hand sides (which are not evaluated). Thus its evaluation is not state dependent.

Proof (of Theorem 1). We first show that if p respects the modifies clause *mod*, this implies the validity of $RespectsModifies(p, mod, \varphi)$, and then that conversely an incorrect modifies clause makes the formula invalid.

Let (S, ρ) be an arbitrary Kripke structure and let s_{pre}, s_{post} be states such that $(s_{pre}, s_{post}) \in \rho(p)$ and such that

$$s_{pre} \models \varphi \wedge DefPre \wedge \{\mathcal{V}^{pre}\}P[*]$$

(for all other states s_{pre}, $\mathit{RespectsModifies}(p, \mathit{mod}, \varphi)$ is trivially true).

We now assume that p respects mod under the precondition φ. Then for all $(f, \bar{v})$ with

$$s_{pre}(f)(\bar{v}) \neq s_{post}(f)(\bar{v})$$

$(f, \bar{v}) \in \mathit{val}_{s_{pre},\beta}(\mathit{mod})$ holds. We also know that, according to Lemma 1, since $s_{pre} \models \mathit{DefPre}$ also $(f, \bar{v}) \in \mathit{val}_{s_{pre},\beta}(\mathit{mod}^{pre})$ holds.

Due to Lemma 3 we know that:

$$\mathit{val}_{s_{pre},\beta}(\mathcal{V}^{pre}) = \mathit{val}_{s_{post},\beta}(\mathcal{V}^{pre})$$

This implies that for every location $(f, \bar{v})$ where f is not a program variable

$$\begin{aligned} \mathit{val}_{s_{pre},\beta}(\mathcal{V}^{pre})(s_{pre})(f)(\bar{v}) &= \mathit{val}_{s_{post},\beta}(\mathcal{V}^{pre})(s_{pre})(f)(\bar{v}) \\ &= \mathit{val}_{s_{post},\beta}(\mathcal{V}^{pre})(s_{post})(f)(\bar{v}) \end{aligned} \tag{6}$$

The explanation for equation (6) is: if

$$s_{pre}(f)(\bar{v}) = s_{post}(f)(\bar{v})$$

then equation (6) holds trivially irrespective of the form of $\mathcal{V}^{pre}$. If

$$s_{pre}(f)(\bar{v}) \neq s_{post}(f)(\bar{v})$$

then, because p respects mod under the precondition φ and because of Lemma 2, there is a $v \in \mathbf{U}$ with $((f, \bar{v}), v) \in \mathit{val}_{s_{post},\beta}(\mathcal{V}^{pre})$ and

$$\begin{aligned} \mathit{val}_{s_{post},\beta}(\mathcal{V}^{pre})(s_{pre})(f)(\bar{v}) &= v \\ &= \mathit{val}_{s_{post},\beta}(\mathcal{V}^{pre})(s_{post})(f)(\bar{v}) \end{aligned}$$

From this we get that

$$\mathit{val}_{s_{pre},\beta}(\mathcal{V}^{pre})(s_{pre})(f) = \mathit{val}_{s_{post},\beta}(\mathcal{V}^{pre})(s_{post})(f)$$

holds for every location function symbol f that is not a local program variable and every variable assignment β. Together with $s_{pre} \models \{\mathcal{V}^{pre}\}P[*]$ this entails that $s_{post} \models \{\mathcal{V}^{pre}\}P[*]$ making $\mathit{RespectsModifies}(p, \mathit{mod}, \varphi)$ valid.

We now assume that mod is not respected by p under the precondition φ. Then, there is a location $(h, \bar{v})$ where h is not a local program variable with $s_{pre}(h)(\bar{v}) \neq s_{post}(h)(\bar{v})$ and $(h, \bar{v}) \notin \mathit{val}_{s_{pre},\beta}(\mathit{mod})$ and consequently (due to Lemma 1) $(h, \bar{v}) \notin \mathit{val}_{s_{pre},\beta}(\mathit{mod}^{pre})$. Together with Lemmas 2 and 3 this entails

$$\begin{aligned} \mathit{val}_{s_{pre},\beta}(\mathcal{V}^{pre})(s_{pre})(h)(\bar{v}) &= s_{pre}(h)(\bar{v}) \\ \mathit{val}_{s_{post},\beta}(\mathcal{V}^{pre})(s_{post})(h)(\bar{v}) &= s_{post}(h)(\bar{v}) \end{aligned}$$

which implies

$$\mathit{val}_{s_{pre},\beta}(\mathcal{V}^{pre})(s_{pre})(h) \neq \mathit{val}_{s_{post},\beta}(\mathcal{V}^{pre})(s_{post})(h)$$

It is thus "admissible" for the predicate symbol $P[*]$ to be interpreted differently in the two states $val_{s_{pre},\beta}(\mathcal{V}^{pre})(s_{pre})$ and $val_{s_{post},\beta}(\mathcal{V}^{pre})(s_{post})$. We can now choose a Kripke structure with states s'_{pre}, s'_{post} such that $s'_{pre}(f) = s_{pre}(f)$, $s'_{pre}(q) = s_{pre}(q)$, $s'_{post}(f) = s_{post}(f)$ and $s'_{post}(q) = s_{post}(q)$ for all function symbols f and all predicate symbols q with $q \neq P[*]$. In addition we require that

$$val_{s_{pre'},\beta}(\mathcal{V}^{pre})(s'_{pre}) \models P[*] \tag{7}$$

$$val_{s_{post'},\beta}(\mathcal{V}^{pre})(s'_{post}) \not\models P[*] \tag{8}$$

Obviously $(s'_{pre}, s'_{post}) \in \rho(p)$ and (due to (7)) $s'_{pre} \models \varphi \wedge \mathit{DefPre} \wedge \{\mathcal{V}^{pre}\}P[*]$ holds. Together with (8) we get:

$$s'_{pre} \not\models \varphi \wedge \mathit{DefPre} \wedge \{\mathcal{V}^{pre}\}P[*] \rightarrow [p]\{\mathcal{V}^{pre}\}P[*]$$

□

B Treatment of Instance Creation

In this appendix we elaborate on how object creation can be taken into account by modifies clauses and how this can be handled by our approach.

Like many other dynamic logics, and modal logics in general, our dynamic logic operates under the constant domain semantics, which means that all states in a Kripke structure have the same universe **U**. The implications this carries for the modeling of object creation are that all objects that can ever be created by a program have to exist in *every* program state.

For indicating whether objects are created (in the sense of the programming language) and which objects are to be created next, we make use of *implicit fields* and object *repositories*. An object repository Rep_C contains all instances of (exact) type C existing in **U** that are already or will be created. Rep_C is accessed via the function symbol get_C which is interpreted as a bijective function

$$val_{s,\beta}(get_C) : \mathbb{N} \rightarrow Rep_C$$

Implicit fields behave like normal fields but are not user declared. We augment objects with the implicit boolean instance field *<created>* and classes with the implicit static field *<nextToCreate>* where

$$val_{s,\beta}(x.\mathit{<created>}) = \mathit{TRUE}$$

iff the object $val_{s,\beta}(x)$ is created in s, and where $C.\mathit{<nextToCreate>}$ denotes the smallest non-negative index such that the object

$$val_{s,\beta}(get_C(C.\mathit{<nextToCreate>}))$$

is not yet created in s. If a new instance of type C is created, then the repository object $get_C(C.\mathit{<nextToCreate>})$ is taken for this purpose and the attribute $C.\mathit{<nextToCreate>}$ is afterwards increased by 1.

The predicate $P[*]$ depends, the way we defined it, also on implicit fields and "normal" instance fields of objects that are not created in the pre-state of a program p. When specifying a modifies clause for a program p which creates new objects, we thus have to make the necessary locations of $<\!created\!>$ and $<\!nextToCreate\!>$ part of the modifies clause.

This is, however, a contrast to a more common semantics of modifies clauses that permits object creation and initialisation of fields of newly-created objects, without having to specify this in modifies clauses explicitly. Fortunately, this semantics can be adopted by our approach easily, by using a location dependent predicate $P[*']$ depending (i) only on those locations that exist already in the pre-state of p and (ii) not on $<\!nextToCreate\!>$ locations. The equivalence of two updates $\{u_1\}$, $\{u_2\}$ with respect to $P[*']$ can be shown by proving the validity of the formula

$$\forall \bar{x}.(\vartheta[\bar{x}] \rightarrow \{u_1\}f(\bar{x}) \doteq \{u_2\}f(\bar{x}))$$

for all function symbols f which occur as the top-level operator on the left hand side in $\{u_1\}$ or $\{u_2\}$ (where f is neither a local program variable nor a $<\!nextToCreate\!>$ attribute). We define $\vartheta[x_1, \ldots, x_n]$ as

$$\bigwedge_{\substack{i\,: \\ i \in \{1,\ldots,n\}, \\ x_i \textit{ has object type}}} x_i.<\!created\!> \doteq \mathtt{true}$$

Thus $\vartheta[\bar{x}]$ indicates that all reference type arguments of each function symbol f occuring as the top-level operator on the left hand side in $\{u_1\}$ or $\{u_2\}$ are already created.