Static analysis of unbounded structures in object-oriented programs

Grabe, I.

Part I

Object Creation

Chapter 2

Abstract Object Creation in Dynamic Logic[1]

In this chapter we give a representation of a weakest precondition calculus for abstract object creation in dynamic logic, the logic underlying the KeY theorem prover. This representation allows to both specify and verify properties of objects at the abstraction level of the (object-oriented) programming language. Objects which are not (yet) created never play any role, neither in the specification nor in the verification of properties. Further, we show how to symbolically execute abstract object creation.

2.1 Introduction

In object-oriented programming languages like Java, objects can be dynamically created by the constructor methods provided by their class. Using constructors for object creation is an abstraction from the underlying representation of objects and the implementation of object creation. At the abstraction level of the programming language, objects are described as instances of their classes, i.e., the classes provide the only operations which can be performed on objects. For practical purposes it is important to be able to specify and verify properties of objects at the abstraction level of the programming language. Specification languages like the Java Modeling Language (JML) [73] and the Object Constraint Language (OCL) [83] abstract from the underlying representation of objects. In [40], a Hoare logic is presented to verify properties of

[1] The work presented in this chapter was published as [10].

an object-oriented programming language at the abstraction level of the programming language itself. This Hoare logic is based on a weakest precondition calculus for object creation which abstracts from the implementation of object creation.

In this chapter we give a representation of a weakest precondition calculus for abstract object creation in dynamic logic, the logic underlying the KeY theorem prover [19]. This representation allows to both specify and verify properties of objects at the abstraction level of the programming language. Objects which are not (yet) created never play any role, neither in the specification nor in the verification of properties.

The generalization of Hoare logic to dynamic logic is of particular interest because it allows for the specification of properties of dynamic object structures which cannot be expressed in first-order logic, like reachability. In Hoare logic such properties require quantification over (finite) sequences or recursively defined predicates in the specification language which seriously complicates both the weakest precondition calculus and the underlying logic. In dynamic logic we can restrict to first-order quantification and use the modalities to express for example reachability properties.

An interesting consequence of the abstraction level of the specification language studied in this chapter is the *dynamic scope* of the quantification over objects because it is restricted to the created objects and as such is also affected by object creation. However, we show that the standard logic of first-order quantification also applies in the presence of (object) quantifiers with a dynamic scope.

Further, we show how to symbolically execute abstract object creation in KeY. In general, symbolic execution in KeY accumulates in a simultaneous substitution of the assignments generated by a computation. This accumulation involves a pre-processing of the substitution which in general simplifies its actual application. However, we cannot simply accumulate abstract object creation because its side-effects can only be processed by the actual application of the corresponding substitution. We show how to solve this problem by the introduction of fresh logical variables which are used as temporary place holders for the newly created objects. The use of these place holders together with the fact that we can always anticipate object creation allows to symbolically execute abstract object creation.

Related work

Most formalisations of object-oriented programs, like embeddings into the logic of higher-order theorem provers PVS [100] and Isabelle [70], or dynamic logic as employed in the KeY theorem prover, use an explicit representation of objects. Object creation is then formalized in terms of the information about which objects are in fact created. Such an explicit representation of objects additionally requires an axiomatization of certain consistency requirements, e.g., the global invariant that the values of the fields of created objects only refer to created objects. These requirements pervade the correctness proofs with the basic case distinction between "to be or not to be created" and adds considerably to the length of the proofs, as we illustrate in Section 2.5.

The contribution of this chapter is the formalization of object creation in dynamic logic which abstracts from an explicit representation of objects and the corresponding implementation of object creation. Proofs in this formalization only refer to created objects and as such are not pervaded by irrelevant implementation details.

Outline

In Section 2.2 we introduce a dynamic logic for a simple WHILE-language with object creation. This language allows us to focus on object creation. We present the axiomatization of the language in terms of the sequent calculus given in Section 2.3. Please observe that this calculus can be extended to other programming constructs of existing object-oriented languages like Java as described in [21]. With the calculus at hand symbolic execution of programs is described in Section 2.4. After a discussion of the state of the art in symbolic execution with respect to object creation and a look into the expressiveness of our approach in Section 2.5 we conclude with Section 2.6.

2.2 Dynamic Logic

To focus on the abstract object creation we restrict ourselves to a simple WHILE-language as our object-oriented programming language. The language contains data of three types Object, Integer, and Boolean. In [21] Becker and Platzer present a similar dynamic logic for Java Card called ODL. ODL covers the type system of Java. Besides the type system, dynamic dispatch, side-effects of expressions, and exception handling are presented in terms of program transformations. However ODL models object creation in terms of an

explicit representation of objects. To obtain a logic covering Java that follows our theory of abstract object creation this representation can be replaced by our theory or our theory can be extended analogous to [21].

2.2.1 Syntax

We assume the sets F of fields and GVar of global variables to be given. Fields are the instance variables of objects. We assume a partitioning of the global variables into a set PVar of program variables and a set LVar of logical variables. Logical variables do not change during program execution, i.e. there are no assignments to logical variables. Logical variables are used to express invariant properties and for (first-order) quantification. All fields and variables are typed. As mentioned before we restrict to the types Object, Integer, and Boolean. We omit explicit declarations. The grammar for statements and expressions of the simple WHILE-language are presented in Figure 2.1.

s	::=	$\mathsf{while}\ e\ \mathsf{do}\ s\ \mathsf{od} \mid \mathsf{if}\ e_1\ \mathsf{then}\ s_2\ \mathsf{else}\ s_3\ \mathsf{fi} \mid s_1; s_2 \mid \mathsf{skip} \mid$	
		$u := \mathsf{new} \mid e_1.x := e_2 \mid u := e$	statements
e	::=	$u \mid e.x \mid \mathsf{null} \mid e_1 = e_2 \mid \mathsf{if}\ e_1\ \mathsf{then}\ e_2\ \mathsf{else}\ e_3\ \mathsf{fi} \mid f(e_1, ..., e_n)$	expressions

Figure 2.1: Grammar rules for the simple WHILE-language

The statement while denotes the usual looping. Conditional branching is denoted by if–then–else. The condition for both looping and branching is given by a Boolean expression. A semicolon denotes sequential composition. By skip we denote the empty statement. Object creation is denoted by $u := \mathsf{new}$, where u is a program variable. An assignment to a program variable is denoted by $u := e$. A dot denotes dereferencing, i.e., $e_1.x := e_2$ denotes an assignment to the field x of the object referenced by e_1. For technical convenience only we do not have assignments $e.x := \mathsf{new}$. In order to separate object creation from the aliasing problem we reason about such assignments in terms of the statement $u := \mathsf{new}; e.x := u$, where u is a fresh program variable.

The expression null of type Object denotes the undefined reference. The Boolean expression $e_1 = e_2$ denotes the test for equality between the values of the expressions e_1 and e_2, e.g., e_1 and e_2 refer to the same object in case e_1 and e_2 are variables of type Object. A conditional expression is denoted by if–then–else. The function $f(e_1, ..., e_n)$ denotes an arithmetic or Boolean operation of arity n. We assume every statement and expression to be well-typed. It is important to note that object expressions, i.e., expressions of type Object,

can only be compared for equality, dereferenced, or appear as argument of a conditional expression.

Formulas. Dynamic logic is a variant of *modal logic*. Different parts of a formula are evaluated in different worlds (states), which vary in the interpretation of, in our case, program variables and fields. Dynamic logic extends full first-order logic with two additional (mix-fix) operators: $\langle\,.\,\rangle\,.$ (diamond) and $[\,.\,]\,.$ (box). In both cases, the first argument is a *program* (fragment), whereas the second argument is another dynamic logic formula. A formula $\langle p\rangle\phi$ is true in a state s if execution of p terminates when started in s and results in a state where ϕ is true. As for the box-operator, a formula $[p]\phi$ is true in a state s if execution of p, when started in s, does *either* not terminate *or* results in a state where ϕ is true. In other words, the difference between the operators is the difference between total and partial correctness.[2] Dynamic logic is closed under all logical connectives.

For instance, the formula $\forall\, l.\,(\langle p\rangle\ (l = u)\ \leftrightarrow\ \langle q\rangle\ (l = u))$ states equivalence of p and q w.r.t. the program variable u.

Example 2.2.1 (Object Creation) *We give an example of a formula involving object creation:* $\forall l.\langle u := \mathsf{new}\rangle\neg(u = l)$. *It states that every new object indeed is new because the logical variable* l *ranges over all the objects that exist* before *the object creation* $u := \mathsf{new}$. *Consequently, after the execution of* $u := \mathsf{new}$ *we have that the new object is not equal to any object that already existed before, i.e.,* $\neg(u = l)$, *when* l *refers to an "old" object. Note that the formula* $\langle u := \mathsf{new}\rangle\forall l.\neg(u = l)$ *has a completely different meaning. In fact the formula is false (cf. Section 2.3.3). These examples illustrate a further advantage of dynamic logic over Hoare logic: the presence of explicit quantifiers in both formulas clarify the difference in meaning.*

All major program logics (Hoare logic, weakest precondition calculus, dynamic logic) have in common that the resolving of assignments requires substitutions in the formula, in one way or the other. In the KeY approach, the effect of substitutions is delayed, by having *explicit substitutions* in the logic, called *'updates'*. In this chapter, elementary updates have the form $u := \mathsf{new}$, $e_1.x := e_2$, or $u := e$. Updates are introduced to the logic via the update

[2] Just as in standard modal logic, the diamond resp. box operators quantify existentially resp. universally over states (reached by the program). In case of deterministic programs, however, the only difference between the two is whether termination is claimed or not.

modality $\{\,.\,\}\,.$, connecting arbitrary updates with arbitrary formulas, like in $0 < v \rightarrow \{u := v\}\, 0 < u$.

A full account of KeY style dynamic logic can be found in [20].

2.2.2 Semantics

To define the semantics of our DL we assume given an arbitrary (infinite) set O of *object identities*, with typical element o. We define null itself to be an element of O, i.e., the value of the expression null is null itself. By $dom(T)$ we denote the domain of values of type T, e.g., $dom(\text{Object})=O$.

States. A state $\Sigma = (\sigma, \tau)$ is a pair consisting of a heap σ and an environment τ. The heap σ is a partial function such that $\sigma(o)$ for every $o \in O$, if defined, denotes the internal state of object o. That is, the value of a field x of an object o, for which $\sigma(o)$ is defined, is given by $\sigma(o)(x) \in dom(T)$. The domain $dom(\sigma)$ of objects that exist in a heap σ is given by the set of objects o for which $\sigma(o)$ is defined. In order to describe unbounded object creation we require the domain of a heap to be finite. The environment τ assigns values to the global variables. The value of a variable v is given by $\tau(v)$.

We require every state $\Sigma = (\sigma, \tau)$ to be consistent, i.e.,

- $\mathsf{null} \in dom(\sigma)$,
- $\sigma(o)(x) \in dom(\sigma)$ for every $o \in dom(\sigma)$ and field x of type Object,
- $\tau(v) \in dom(\sigma)$ for every global variable v of type Object.

In words, null is an existing object, the fields of type Object of existing objects refer to existing objects and all global variables of type Object refer to existing objects.

Semantics of Expressions and Statements. The semantics of an expression e of type T is a partial function $[\![e]\!] : \Sigma \rightharpoonup dom(T)$. As an example, if $[\![e]\!]$ is defined and does not evaluate to null then

$$[\![e.x]\!](\sigma, \tau) = \sigma([\![e]\!](\sigma, \tau))(x),$$

otherwise $[\![e.x]\!]$ is undefined. For a general treatment of failures we assume given a predicate $def(e)$ which defines the conditions under which the expression e is defined. For example, we have that $def(u.x) \equiv \neg(u = \mathsf{null})$.

The semantics of a statement s is a partial function $[\![s]\!] : \Sigma \rightharpoonup \Sigma$. We focus on the semantics of object creation. In order to formally describe the initialisation of newly created objects, we first introduce for each type T an initial value of type T, i.e., $init_{\text{Object}} = \mathsf{null}$, $init_{\text{Integer}} = 0$, and $init_{\text{Boolean}} = \mathsf{false}$. We define *init* to be the initial state, i.e., the state that assigns to each field x of type T its initial value $init_T$. For the selection of a new object we use a choice function ν on heaps to get a fresh object, i.e., $\nu(\sigma) \notin dom(\sigma)$.

We now define

$$[\![u := \mathsf{new}]\!](\sigma, \tau) = (\sigma[o := init], \tau[u := o]),$$

where $o = \nu(\sigma)$. The heap $\sigma[o := init]$ assigns the local state *init* to the new object o and the environment $\tau[u := o]$ assigns this object to the program variable u.

Semantics of Formulas. A formula ϕ in dynamic logic is valid if $\Sigma \models \phi$ holds for every consistent state Σ. For a logical variable l of type Object, we have the following semantics of universal quantification

$$(\sigma, \tau) \models \forall l.\phi \text{ iff for all } o \in dom(\sigma) : (\sigma, \tau[l := o]) \models \phi,$$

where the consistency of $(\sigma, \tau[l := o])$ implies that the object o exists in σ. Consequently, quantification is restricted to the existing objects. Note that null is always included in the scope of the quantification (i.e., the scope of the quantification is non-empty).

Returning to the above example, we have

$$\begin{array}{l}(\sigma, \tau) \models \forall l.\langle u := \mathsf{new}\rangle\neg(u = l)\\ \text{iff}\\ (\sigma, \tau[l := o]) \models \langle u := \mathsf{new}\rangle\neg(u = l)\end{array}$$

for all $o \in dom(\sigma)$. Let $o' = \nu(\sigma)$. By the semantics of the diamond modality of dynamic logic and the above semantics of object creation we conclude that

$$\begin{array}{l}(\sigma, \tau[l := o]) \models \langle u := \mathsf{new}\rangle\neg(u = l)\\ \text{iff}\\ (\sigma[o' := init], \tau[l := o]) \models \neg(u = l)\\ \text{iff}\\ o \neq o'\end{array}$$

Note that since $o' \notin dom(\sigma)$ by definition of $\nu(\sigma)$ indeed $o \neq o'$ for all $o \in dom(\sigma)$.

2.3 Axiomatization

In this section, we introduce a proof system for dynamic logic with object creation which abstracts from the explicit representation of objects in the semantics defined above. As a consequence the rules of the proof system are purely defined in terms of the logic itself and do not refer to the semantics. It is characteristic for dynamic logic, in contrast to Hoare logic or weakest precondition calculi, that program reasoning is fully interleaved with first-order logic reasoning, because diamond, box or update modalities can appear both outside and inside the logical connectives and quantifiers. It is therefore important to realise that in the following proof rules, ϕ, ψ and alike, match *any* formula of our logic, possibly containing programs or updates.

2.3.1 Sequent Calculus

We follow [21, 19] in presenting the proof system for dynamic logic as a sequent calculus. A sequent is a pair of sets of formulas (each formula closed for logical variables) written as $\phi_1, ..., \phi_m \vdash \psi_1, ..., \psi_n$. The intuitive meaning is that, given all of $\phi_1, ..., \phi_m$ hold, at least one of $\psi_1, ..., \psi_n$ must hold. We use capital Greek letters to denote (possibly empty) sets of formulas. For instance, by $\Gamma \vdash \phi \to \psi, \Delta$ we mean a sequent containing at least an implication formula on the right side. Sequent calculus rules always have one sequent as conclusion and zero, one or many sequents as premises:

$$\frac{\Gamma_1 \vdash \Delta_1 \ \ldots \ \Gamma_n \vdash \Delta_n}{\Gamma \vdash \Delta}$$

Semantically, a rule states that the validity of all n premises implies the validity of the conclusion ("top-down"). Operationally, rules are applied bottom-up, reducing the provability of the conclusion to the provability of the premises, starting from the initial sequent to be proved. Rules with no premise close the current proof branch. In Figure 2.2 we present some of the rules dealing with propositional connectives and quantifiers (see [55] for the full set). We omit the rules for the left hand side, the rules to deal with negation and the rule to cover conditional expressions. $\phi[l/e]$ denotes standard substitution of l with e in ϕ.

When it comes to the rules dealing with programs, most of them are not sensitive to the side of the sequent and can moreover be applied to subformulas even. For instance, $\langle s_1; s_2 \rangle \phi$ can be split up into $\langle s_1 \rangle \langle s_2 \rangle \phi$ regardless of where

$$\textsf{impRight}\ \frac{\Gamma, \phi \vdash \psi, \Delta}{\Gamma \vdash \phi \to \psi, \Delta} \qquad \textsf{andRight}\ \frac{\Gamma \vdash \phi, \Delta \qquad \Gamma \vdash \psi, \Delta}{\Gamma \vdash \phi \wedge \psi, \Delta}$$

$$\textsf{allRight}\ \frac{\Gamma \vdash \phi[l/c], \Delta}{\Gamma \vdash \forall l.\phi, \Delta}\ \text{with } c \text{ a new constant} \qquad \textsf{allLeft}\ \frac{\Gamma, \forall l.\phi, \phi[l/e] \vdash \Delta}{\Gamma, \forall l.\phi \vdash \Delta}\ \text{with } e \text{ an expression}$$

$$\textsf{close}\ \frac{}{\Gamma, \phi \vdash \phi, \Delta}$$

$$\textsf{ind}\ \frac{\Gamma \vdash \phi[l/0], \Delta \qquad \Gamma \vdash \forall l.(\phi \to \phi[l/l+1]), \Delta}{\Gamma \vdash \forall l.\phi, \Delta}\ \text{with } l \text{ of type Integer}$$

Figure 2.2: Sequent rules - first-order logic rules

it occurs. For that we introduce the following syntax

$$\frac{\lfloor\, \phi' \,\rfloor}{\lfloor\, \phi \,\rfloor}$$

for a schema rule where the premise is constructed from the conclusion via replacing an occurrence of ϕ by ϕ'.

In Figure 2.3 we present the rules dealing with statements. The schematic modality $\langle\!|\cdot|\!\rangle$ can be instantiated with both $[\cdot]$ and $\langle\cdot\rangle$, though consistently within a single rule application. The extension of these rules with the predicate $def(e)$ to reason about failures is standard and therefore omitted.

Total correctness formulas of the form $\langle \textsf{while} \ldots \rangle \phi$ are proved by first applying the induction rule ind (possibly after generalising the formula) and applying the unwind rule within the induction step. For space reasons, we omit the invariant rule dealing with formulas of the form $[\textsf{while} \ldots]\phi$ (see [21, 20]).

$$\text{split}\ \frac{\lfloor\, \langle\!\langle s_1 \rangle\!\rangle \langle\!\langle s_2 \rangle\!\rangle \phi \,\rfloor}{\lfloor\, \langle\!\langle s_1; s_2 \rangle\!\rangle \phi \,\rfloor} \qquad \text{if}\ \frac{\lfloor\, (e \rightarrow \langle\!\langle s_1 \rangle\!\rangle \phi) \wedge (\neg e \rightarrow \langle\!\langle s_2 \rangle\!\rangle \phi) \,\rfloor}{\lfloor\, \langle\!\langle \mathsf{if}\ e\ \mathsf{then}\ s_1\ \mathsf{else}\ s_2\ \mathsf{fi} \rangle\!\rangle \phi \,\rfloor}$$

$$\text{unwind}\ \frac{\lfloor\, \langle\!\langle \mathsf{if}\ e\ \mathsf{then}\ s; \mathsf{while}\ e\ \mathsf{do}\ s\ \mathsf{od}\ \mathsf{else}\ \mathsf{skip}\ \mathsf{fi} \rangle\!\rangle \phi \,\rfloor}{\lfloor\, \langle\!\langle \mathsf{while}\ e\ \mathsf{do}\ s\ \mathsf{od} \rangle\!\rangle \phi \,\rfloor}$$

$$\text{assignVar}\ \frac{\lfloor\, \{u := e\}\phi \,\rfloor}{\lfloor\, \langle\!\langle u := e \rangle\!\rangle \phi \,\rfloor} \qquad \text{assignField}\ \frac{\lfloor\, \{e_1.x := e_2\}\phi \,\rfloor}{\lfloor\, \langle\!\langle e_1.x := e_2 \rangle\!\rangle \phi \,\rfloor}$$

$$\text{createObj}\ \frac{\lfloor\, \{u := \mathsf{new}\}\phi \,\rfloor}{\lfloor\, \langle\!\langle u := \mathsf{new} \rangle\!\rangle \phi \,\rfloor}$$

Figure 2.3: Sequent rules - dynamic logic rules

2.3.2 Application of General Updates

Updates are essentially delayed substitutions.[3] They are resolved by application to the succeeding formula, e.g., $\{u := e\}(u > 0)$ leads to $e > 0$. Update application is only allowed on formulas *not* starting with either a diamond, box or update modality. The last restriction is dropped for symbolic execution, see Section 2.4.

We now define update application on formulas in terms of a rewrite relation $\{\mathcal{U}\}\phi \rightsquigarrow \phi'$ on formulas. As a technical vehicle, we extend the update operator to expressions, such that $\{\mathcal{U}\}e$ is an expression, for all updates $\mathcal{U}$ and expressions e. Accordingly, the rewrite relation $\rightsquigarrow$ carries over to such expressions: $\{\mathcal{U}\}e \rightsquigarrow e'$.

In Figure 2.4 we define $\rightsquigarrow$ for all standard cases (see also [92, 19]). The symbol $\mathcal{U}$ matches all updates, whereas $\mathcal{U}_{nc}$ ('*n*on-*c*reating') excludes statements of the form $u := \mathsf{new}$. Furthermore, Lit is the set of literals of all types, in our context $\{\mathsf{null}, \mathsf{true}, \mathsf{false}\} \cup \{\ldots, -1, 0, 1, \ldots\}$. (Recall LVar is the set of logical variables.)

The aliasing analysis performed by the last rule is the motivation to add

[3] The benefit of delaying substitutions in the context of symbolic execution is illustrated in Section 2.4.

$$\frac{\{\mathcal{U}\}\phi_1 * \{\mathcal{U}\}\phi_2 \rightsquigarrow \phi'}{\{\mathcal{U}\}(\phi_1 * \phi_2) \rightsquigarrow \phi'} \quad \text{with } * \in \{\wedge, \vee, \rightarrow\}$$

$$\frac{\neg\{\mathcal{U}\}\phi \rightsquigarrow \phi'}{\{\mathcal{U}\}(\neg\phi) \rightsquigarrow \phi'}$$

$$\frac{Q\, l.\, \{\mathcal{U}_{nc}\}\phi \rightsquigarrow \phi'}{\{\mathcal{U}_{nc}\}(Q\, l.\, \phi) \rightsquigarrow \phi'} \quad \text{with } Q \in \{\forall, \exists\},\ l \text{ not in } \mathcal{U}_{nc}$$

$$\{\mathcal{U}\}\alpha \rightsquigarrow \alpha \quad \text{with } \alpha \in \mathrm{LVar} \cup \mathrm{Lit}$$

$$\frac{\{\mathcal{U}_{nc}\}e_1 = \{\mathcal{U}_{nc}\}e_2 \rightsquigarrow e'}{\{\mathcal{U}_{nc}\}(e_1 = e_2) \rightsquigarrow e'} \qquad \frac{f(\{\mathcal{U}\}e_1, ..., \{\mathcal{U}\}e_n) \rightsquigarrow e'}{\{\mathcal{U}\}f(e_1, ..., e_n) \rightsquigarrow e'}$$

$$\frac{(\{u := e_1\}e_2).x \rightsquigarrow e'}{\{u := e_1\}(e_2.x) \rightsquigarrow e'} \qquad \frac{(\{e.x := e_1\}e_2).y \rightsquigarrow e'}{\{e.x := e_1\}(e_2.y) \rightsquigarrow e'} \quad x, y \text{ different fields}$$

$$\{u_1 := e\}u_2 \rightsquigarrow u_2 \quad u_1, u_2 \text{ different variables} \qquad \{u := e\}u \rightsquigarrow e$$

$$\frac{\mathsf{if}\ (\{e.x := e_1\}e_2) = e\ \mathsf{then}\ e_1\ \mathsf{else}\ (\{e.x := e_1\}e_2).x\ \mathsf{fi}\ \rightsquigarrow\ e'}{\{e.x := e_1\}(e_2.x) \rightsquigarrow e'}$$

Figure 2.4: Application of Updates - standard cases

conditional expressions to our language. Object creation of the form $u := \mathsf{new}$ is only covered as far as it behaves like any other update. The cases where object creation makes a difference are discussed separately in Section 2.3.3. The relation $\rightsquigarrow$ is defined in a big-step manner, such that updates are resolved completely in a single $\rightsquigarrow$ step.

Note that $\rightsquigarrow$ is not defined for formulas of the form $\{\mathcal{U}\}\langle s\rangle\phi$, $\{\mathcal{U}\}[s]\phi$ or $\{\mathcal{U}\}\{\mathcal{U}'\}\phi$, i.e., they are not subject to update application. We return to formulas with nested updates, like $\{\mathcal{U}\}\{\mathcal{U}'\}\phi$, in Section 2.4.

The following rule links the rewrite relation $\rightsquigarrow$ with the sequent calculus:

$$\mathsf{applyUpd}\quad \frac{\lfloor\, \phi' \,\rfloor}{\lfloor\, \{\mathcal{U}\}\phi \,\rfloor} \quad \text{with } \{\mathcal{U}\}\phi \rightsquigarrow \phi'$$

2.3.3 Contextual Application of Object Creation

To define update application on expressions $\{u := \mathsf{new}\}e$, simple substitution is not sufficient, i.e., replacing u in e by some expression, because we cannot refer to the newly created object in the state prior to its creation. However, since object expressions can only be compared for equality, or dereferenced, and do not appear as arguments of any other function, we define update application by a contextual analysis of the occurrences of u in e.

We define application of $u := \mathsf{new}$ inductively. Some cases are already covered in Section 2.3.2, Figure 2.4 (the rules dealing with unrestricted $\mathcal{U}$). The other cases are discussed in the following.

If u_1 and u_2 are different variables, then

$$\{u_1 := \mathsf{new}\}u_2 \rightsquigarrow u_2$$

Since the fields of a newly created object are initialised we have

$$\{u := \mathsf{new}\}u.x \rightsquigarrow init_T$$

where T is the type of x.

If e is neither u nor a conditional expression then

$$\frac{(\{u := \mathsf{new}\}e).x \rightsquigarrow e'}{\{u := \mathsf{new}\}(e.x) \rightsquigarrow e'}$$

Otherwise, if e is a conditional expression then

$$\frac{\mathsf{if}\ \{u := \mathsf{new}\}b\ \mathsf{then}\ \{u := \mathsf{new}\}(e_1.x)\ \mathsf{else}\ \{u := \mathsf{new}\}(e_2.x)\ \mathsf{fi} \rightsquigarrow e'}{\{u := \mathsf{new}\}(\mathsf{if}\ b\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2\ \mathsf{fi}\ .x) \rightsquigarrow e'}$$

Note that we use here the valid equation:

$$\mathsf{if}\ b\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2\ \mathsf{fi}\ .x = \mathsf{if}\ b\ \mathsf{then}\ e_1.x\ \mathsf{else}\ e_2.x\ \mathsf{fi}.$$

The only other possible context of u is that of an equality $e = e'$. We distinguish the following cases.

If neither e nor e' is u or a conditional expression then they cannot refer to the newly created object and we define[4]

$$\frac{(\{u := \mathsf{new}\}e) = (\{u := \mathsf{new}\}e') \rightsquigarrow e''}{\{u := \mathsf{new}\}(e = e') \rightsquigarrow e''}$$

If e is u and e' is neither u nor a conditional expression (or vice versa) then after $u := \mathsf{new}$ the expressions e and e' cannot denote the same object (because one of them refers to the newly created object and the other one refers to an already existing object) and so we define

$$\{u := \mathsf{new}\}(e = e') \rightsquigarrow \mathsf{false}$$

On the other hand if both the expressions e and e' equal u we obviously have

$$\{u := \mathsf{new}\}(e = e') \rightsquigarrow \mathsf{true}$$

If e is a conditional expression of the form $\mathsf{if}\ b\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2\ \mathsf{fi}$ then

$$\frac{\mathsf{if}\ \{u := \mathsf{new}\}b\ \ \mathsf{then}\ \{u := \mathsf{new}\}(e_1 = e')\ \mathsf{else}\ \{u := \mathsf{new}\}(e_2 = e')\ \mathsf{fi} \rightsquigarrow e''}{\{u := \mathsf{new}\}(e = e') \rightsquigarrow e''}$$

And similarly for $e' = e$. Note that we use here the valid equation:

$$(\mathsf{if}\ b\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2\ \mathsf{fi} = e') = \mathsf{if}\ b\ \mathsf{then}\ e_1 = e'\ \mathsf{else}\ e_2 = e'\ \mathsf{fi}$$

Since object expressions can only be compared for equality, dereferenced or appear as argument of a conditional expression, it is easy to see that for every boolean expression e there exists an expression e' such that $\{u := \mathsf{new}\}e \rightsquigarrow e'$.

The following lemma states the semantic correctness of the rewrite relation $\{u := \mathsf{new}\}e \rightsquigarrow e'$: The value of e' in the state *before* the assignment $u := \mathsf{new}$ equals the value of e *after* the assignment.

Lemma 2.3.1
If $\{u := \mathsf{new}\}e \rightsquigarrow e'$ *and* $[\![u := \mathsf{new}]\!](\sigma, \tau) = (\sigma', \tau')$ *then* $[\![e']\!](\sigma, \tau) = [\![e]\!](\sigma', \tau')$.

The proof of this lemma involves a further elaboration of proofs given in [16].

Now we define the rewriting of $\{u := \mathsf{new}\}\phi$, where ϕ is a first-order formula in predicate logic (which does not contain modalities). The rules for this generalization are standard.

[4] To see why the shifting inwards of $\{u := \mathsf{new}\}$ is necessary, consider the case $\{u := \mathsf{new}\}(u.x = u.x)$.

Example 2.3.2 *We present a rule for quantification as an example:*

$$\frac{(\{u := \mathsf{new}\}\phi[l/u]) \wedge \forall l.(\{u := \mathsf{new}\}\phi) \rightsquigarrow \psi}{\{u := \mathsf{new}\}\forall l.\phi \rightsquigarrow \psi}$$

where l is a logical variable. This rewrite rule takes care of the changing scope *of the quantified variable l by distinguishing the following cases: p holds for the new object is expressed by the first conjunct $\{u := \mathsf{new}\}\phi[l/u]$ which is obtained by application of the update to $\phi[l/u]$ and p holds for all 'old' objects is expressed by the second conjunct $\forall l.(\{u := \mathsf{new}\}\phi)$.*

Example 2.3.3 *As an example, we derive $\{u := \mathsf{new}\}\forall l.\neg(u = l) \rightsquigarrow \neg(\mathsf{true}) \wedge \forall l.\neg\mathsf{false}$:*

$$\dfrac{\dfrac{\dfrac{\{u := \mathsf{new}\}(u = u) \rightsquigarrow \mathsf{true}}{\{u := \mathsf{new}\}\neg(u = u) \rightsquigarrow \neg(\mathsf{true})} \quad \dfrac{\dfrac{\{u := \mathsf{new}\}(u = l) \rightsquigarrow \mathsf{false}}{\{u := \mathsf{new}\}\neg(u = l) \rightsquigarrow \neg\mathsf{false}}}{\forall l.\{u := \mathsf{new}\}\neg(u = l) \rightsquigarrow \forall l.\neg\mathsf{false}}}{\{u := \mathsf{new}\}\neg(u = u) \wedge \forall l.\{u := \mathsf{new}\}\neg(u = l) \rightsquigarrow \neg(\mathsf{true}) \wedge \forall l.\neg\mathsf{false}}}{\{u := \mathsf{new}\}\forall l.\neg(u = l) \rightsquigarrow \neg(\mathsf{true}) \wedge \forall l.\neg\mathsf{false}}$$

The resulting formula is equivalent to false. *We use this to prove the formula $\langle u := \mathsf{new}\rangle\forall l.\neg(u = l)$, which states that u is different from all objects existing* after *the update (including u itself), invalid. In fact we have the following derivation for $\neg\langle u := \mathsf{new}\rangle\forall l.\neg(u = l)$.*

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\forall l.\neg\mathsf{false} \vdash \mathsf{true}}\,\mathsf{closeTrue}}{\neg(\mathsf{true}), \forall l.\neg\mathsf{false} \vdash}\,\mathsf{notLeft}}{\neg(\mathsf{true}) \wedge \forall l.\neg\mathsf{false} \vdash}\,\mathsf{andLeft}}{\{u := \mathsf{new}\}\forall l.\neg(u = l) \vdash}\,\mathsf{applyUpd}}{\langle u := \mathsf{new}\rangle\forall l.\neg(u = l)) \vdash}\,\mathsf{assignVar}}{\vdash \neg\langle u := \mathsf{new}\rangle\forall l.\neg(u = l)}\,\mathsf{notRight}$$

On the other hand, we have the following derivation of

$$\forall l.\langle u := \mathsf{new}\rangle\neg(u = l)$$

which expresses in an abstract and natural way that u indeed is a new object different from objects existing before *the update.*

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\mathsf{false} \vdash}\,\mathsf{closeFalse}}{\vdash \neg\mathsf{false}}\,\mathsf{notRight}}{\vdash \{u := \mathsf{new}\}\neg(u = c)}\,\mathsf{applyUpd}}{\vdash \langle u := \mathsf{new}\rangle\neg(u = c)}\,\mathsf{assignVar}}{\vdash \forall l.(\langle u := \mathsf{new}\rangle\neg(u = l))}\,\mathsf{allRight}$$

The second example shows that the standard rules for quantification apply to the quantification over the existing objects.

2.4 Symbolic Execution

2.4.1 Simultaneous Updates for Symbolic State Representation

The proof system presented so far allows for classical backwards reasoning, in a weakest precondition manner. We now generalise the notion of updates, to allow for the *accumulation* of substitutions, thereby delaying their application. In particular, this can be done in a *forward manner*, giving the proofs a *symbolic execution* nature. We illustrate this principle by example:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{u<v\vdash u<v}\,\mathsf{close}}{u<v\vdash\{w:=u\,|\,u:=v\,|\,v:=u\}v<u}\,\mathsf{applyUpd}}{u<v\vdash\{w:=u\,|\,u:=v\}\{v:=w\}v<u}\,\mathsf{mergeUpd}}{u<v\vdash\{w:=u\,|\,u:=v\}\langle v:=w\rangle v<u}\,\mathsf{assignVar}}{u<v\vdash\{w:=u\}\{u:=v\}\langle v:=w\rangle v<u}\,\mathsf{mergeUpd}}{u<v\vdash\{w:=u\}\langle u:=v;v:=w\rangle v<u}\,\mathsf{split},\mathsf{assignVar}}{u<v\vdash\langle w:=u;u:=v;v:=w\rangle v<u}\,\mathsf{split},\mathsf{assignVar}$$

The first application of the update rule mergeUpd introduces what is called the simultaneous update $w:=u\,|\,u:=v$. After applying the second mergeUpd, note that the w from the inner update was turned into a u in the simultaneous update. This is achieved by *applying* the outer update to the inner one:

$$\mathsf{mergeUpd}\quad\dfrac{\lfloor\{\mathcal{U}_1\,|\,\ldots\,|\mathcal{U}_n\,|\mathcal{U}'\}\phi\rfloor}{\lfloor\{\mathcal{U}_1\,|\,\ldots\,|\mathcal{U}_n\}\{U\}\phi\rfloor}\ \text{with}\ \{\mathcal{U}_1\,|\,\ldots\,|\mathcal{U}_n\}\mathcal{U}\rightsquigarrow\mathcal{U}'$$

For this, we need to extend the rewrite relation $\rightsquigarrow$ towards defining application of updates to updates:

$$\dfrac{u:=\{\mathcal{U}_{nc}\}e\rightsquigarrow\mathcal{U}'}{\{\mathcal{U}_{nc}\}(u:=e)\rightsquigarrow\mathcal{U}'}\qquad\dfrac{(\{\mathcal{U}_{nc}\}e_1).x:=\{\mathcal{U}_{nc}\}e_2\rightsquigarrow\mathcal{U}'}{\{\mathcal{U}_{nc}\}(e_1.x:=e_2)\rightsquigarrow\mathcal{U}'}$$

What remains is the definition of the application of simultaneous updates to *expressions*. For space reasons, we will not include the full definition here,

but only one interesting special case, where two left-hand sides both write the field x which is accessed in $e.x$.

$$\frac{\mathsf{if}\ ((\mathcal{U}e_2) = e)\ \mathsf{then}\ e_2'\,\mathsf{else}\ \mathsf{if}\ ((\mathcal{U}e_1) = e)\ \mathsf{then}\ e_1'\,\mathsf{else}\ \mathcal{U}(e).x\ \mathsf{fi}\ \mathsf{fi}\ \rightsquigarrow\ e'}{\mathcal{U}(e.x) \rightsquigarrow e'}$$
$$\text{with } \mathcal{U} = \{e_1.x := e_1' \mid e_2.x := e_2'\}$$

This already illustrates two principles: a recursive alias analysis has to be performed on all left-hand sides, and moreover, in case of a clash, the rightmost update will 'win'. The latter is exactly what reflects the destructive semantics of imperative programming. Most cases are, however, much simpler. Most of the time, it is sufficient to think of an application of a simultaneous update as an application of a standard substitution (of more than one variable). For a full account on simultaneous updates, see [92].

The idea to use simultaneous updates for symbolic execution was developed in the KeY project [19], and turned out to be a powerful concept for the validation of real world (Java) programs. A simultaneous update forms a representation of the symbolic state which is reached by "executing" the program in the proof up to the current proof node. The program is "executed" in a forward manner, avoiding the backwards execution of (pure) weakest precondition calculi, thereby achieving better readability of proofs. The simultaneous update is only applied to the post-condition as a final, single step. The KeY tool uses these updates not only for verification, but also for test case generation with high code based coverage [46] and for symbolic debugging.

2.4.2 Symbolic Execution and Abstract Object Creation

A motivation to choose the setting of dynamic logic with updates is to allow for abstract object creation in symbolic execution style verification. To do so, we have to answer the question of how symbolic execution and abstract object creation can be combined. The problem is that there is no natural way of merging object creation $\{u := new\}$ with other updates. Consider, for instance, the following formulas, only the first of which is valid.

$$\langle u := \mathsf{new}; v := u\rangle(u = v) \qquad \langle u := \mathsf{new}; v := \mathsf{new}\rangle(u = v)$$

Symbolic execution generates the following formulas:

$$\{u := \mathsf{new}\}\{v := u\}(u = v) \qquad \{u := \mathsf{new}\}\{v := \mathsf{new}\}(u = v)$$

Merging the updates naively results in both cases in:

$$\{u := \mathsf{new} \mid v := \mathsf{new}\}(u = v)$$

Whichever semantics one gives to a simultaneous update with two object creations, the formula cannot be both valid and invalid.

The proposed solution is twofold: not to merge an object creation with other updates at all, but to create a second reference to the new object, to be used for merging. For this, we introduce a *fresh* auxiliary variable to store the newly created object, and generate *two* updates according to the following rule:

$$\mathsf{createObj}\quad \frac{\lfloor\, \{a := \mathsf{new}\}\{u := a\}\phi \,\rfloor}{\lfloor\, \langle u = \mathsf{new}\rangle\phi \,\rfloor} \quad \text{with } a \text{ a fresh program variable}$$

The inner update $\{u := v\}$ can be merged with other updates resulting from the analysis of ϕ. The next point to address is the "disruption" of the symbolic state, caused by object creation being unable to merge with their "neighbours", thereby strictly separating state changes happening before and after object creation. The key idea to overcome this is to gradually move all object creations to the very front (as if all objects were allocated up front) and perform standard symbolic execution on the remaining updates. We achieve this by the following rule:

$$\mathsf{pullCreation}\quad \frac{\lfloor\, \{u := \mathsf{new}\}\mathcal{U}_{nc}\phi \,\rfloor}{\lfloor\, \mathcal{U}_{nc}\{u := \mathsf{new}\}\phi \,\rfloor} \quad \text{with } u \text{ not appearing in } \mathcal{U}_{nc}$$

We illustrate symbolic execution with abstract object creation by an example.

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\vdash \neg\mathsf{false}}\,\mathsf{notRight},\mathsf{closeFalse}}{\vdash \{a := \mathsf{new}\}\neg(v = a)}\,\mathsf{applyUpd}}{\vdash \{a := \mathsf{new}\}\{u := v \mid v := a \mid w := u\}\neg(w = v)}\,\mathsf{applyUpd}}{\vdash \{a := \mathsf{new}\}\{u := v \mid v := a\}\{w := u\}\neg(w = v)}\,\mathsf{mergeUpd}}{\vdash \{a := \mathsf{new}\}\{u := v\}\{v := a\}\langle w := u\rangle\neg(w = v)}\,\mathsf{mergeUpd},\mathsf{assignVar}}{\vdash \{u := v\}\{a := \mathsf{new}\}\{v := a\}\langle w := u\rangle\neg(w = v)}\,\mathsf{pullCreation}}{\vdash \{u := v\}\langle v := \mathsf{new}; w := u\rangle\neg(w = v)}\,\mathsf{split},\mathsf{createObj}}{\vdash \langle u := v; v := \mathsf{new}; w := u\rangle\neg(w = v)}\,\mathsf{split},\mathsf{assignVar}$$

2.5 Discussion

2.5.1 Object Creation vs. Object Activation

Proof systems for object-oriented languages(cf. [2]) usually achieve the uniqueness of objects via an injective mapping, here called obj, from the natural numbers to object identities. Only the object identities $\mathsf{obj}(i)$ up to a maximum index i are considered to stand for actually created objects. In each state, the successor of this maximum index is stored in a ghost variable, here called next. (In case of Java, next would be a `static` field, for each class). Object creation increases the value of next, which conceptually is more an activation than a creation. Quantifiers cover the entire co-domain of obj, including "not yet created" objects. In order to restrict a certain property ϕ to the "created" objects, the following pattern is used: $\forall l.(\psi \rightarrow \phi)$, where ψ restricts to the created objects. Formulas of the form $\exists n.\,(n < \mathsf{next} \wedge \mathsf{obj}(n) = l)$ are the approach taken in ODL [21]. To avoid the extra quantifier, ghost instance variable of boolean type, here called $\mathsf{created}$, can be used to indicate for each object whether or not it has already been "created", see [20]. In this case we set the $\mathsf{created}$ status of the "new" object (identified by next) and increase next. The assertion $\forall n.(\mathsf{obj}(n).\mathsf{created} \leftrightarrow n < \mathsf{next})$ retains the relation between the $\mathsf{created}$ status and the object counter next on the level of the proofs. In both case, we need further assertions to state that fields of created objects always refer to created objects.

To state in this setting that a new object indeed is new we need to argument the formula introduced in Section 2.3, i.e. $\forall l.\,(l.\mathsf{created} \rightarrow \langle u := \mathsf{new}\rangle \neg(u = l))$. In fact the formula in Section 2.3 is not valid in this setting. An object activation style proof of this is given in Figure 2.5 (abbreviating $\mathsf{created}$ by cr). Many steps in this proof are caused by the particular details of the explicit representation of objects and the simulation of object creation by object activation.

2.5.2 Expressiveness

Many interesting properties of dynamic object structures, like reachability in dynamic linked data structures, cannot be expressed in first-order predicate logic. There are approaches to simulate reachability by an overapproximation of the reachable states [74]. In first-order dynamic logic however we can use the modalities to express such properties. For example, if a linked list is given in terms of a field *next* and the data is stored in a field *data* then the following formula in dynamic logic states that the object denoted by v is reachable from

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{c.\mathsf{cr}, \mathsf{obj}(\mathsf{next}) = c \vdash c.\mathsf{cr}}\,\mathsf{close}}{c.\mathsf{cr}, \mathsf{obj}(\mathsf{next}) = c \vdash \mathsf{obj}(\mathsf{next}).\mathsf{cr}}\,\mathsf{equality}}{\neg\mathsf{obj}(\mathsf{next}).\mathsf{cr}, c.\mathsf{cr}, \mathsf{obj}(\mathsf{next}) = c \vdash}\,\mathsf{notLeft}}{(\mathsf{obj}(\mathsf{next}).\mathsf{cr} \leftrightarrow \mathsf{next} < \mathsf{next}), c.\mathsf{cr}, \mathsf{obj}(\mathsf{next}) = c \vdash}\,(2\ \mathsf{rules})}{\forall n.(\mathsf{obj}(n).\mathsf{cr} \leftrightarrow n < \mathsf{next}), c.\mathsf{cr}, \mathsf{obj}(\mathsf{next}) = c \vdash}\,\mathsf{allLeft}}{c.\mathsf{cr}, \mathsf{obj}(\mathsf{next}) = c \vdash}\,\mathsf{assumption}(1)}{c.\mathsf{cr} \vdash \neg(\mathsf{obj}(\mathsf{next}) = c)}\,\mathsf{notRight}}{c.\mathsf{cr} \vdash \{u := \mathsf{obj}(\mathsf{next}) \mid \mathsf{obj}(\mathsf{next}).\mathsf{cr} := \mathsf{true} \mid \mathsf{next} := \mathsf{next}+1\}\neg(u = c)}\,\mathsf{applyUpd}\,\mathsf{createObj}$$

$$\dfrac{\dfrac{c.\mathsf{cr} \vdash \langle u := \mathsf{new}\rangle\neg(u = c)}{\vdash c.\mathsf{cr} \rightarrow \langle u := \mathsf{new}\rangle\neg(u = c)}\,\mathsf{impRight}}{\vdash \forall l.\,(l.\mathsf{cr} \rightarrow \langle u := \mathsf{new}\rangle\neg(u = l))}\,\mathsf{allRight}$$

Figure 2.5: Object activation style proof

the object denoted by u:

$$\langle \mathsf{while}\ u \neq v\ \mathsf{do}\ u := u.next\ \mathsf{od}\rangle(\mathsf{true})$$

Note that in DL such formulas can be used to express properties themselves.

2.6 Conclusion

In this chapter we gave a representation of a weakest precondition calculus for abstract object creation in dynamic logic and the KeY theorem prover. Abstract object creation is formalized in terms of an inductively defined rewrite relation. The standard sequent calculus for dynamic logic is extended with a schema rule which allows to substitute formulas in sequents and thus provides a general mechanism to import for example specific rewrite relations. The resulting logic abstracts from an explicit representation of objects and the corresponding implementation of object creation. As such it abstracts from irrelevant implementation details which in general complicate proofs. Moreover, it treats the dynamic scope of quantified object variables in a standard manner. Finally, we have shown how to symbolically execute abstract object creation in KeY.