Softw Syst Model (2016) 15:1117–1140
DOI 10.1007/s10270-014-0446-9

THEME SECTION PAPER

Integrating deductive verification and symbolic execution for abstract object creation in dynamic logic

Stijn de Gouw · Frank de Boer · Wolfgang Ahrendt · Richard Bubel

Received: 15 November 2013 / Revised: 15 August 2014 / Accepted: 3 December 2014 / Published online: 25 December 2014

Abstract We present a fully abstract weakest precondition calculus and its integration with symbolic execution. Our assertion language allows both specifying and verifying properties of objects at the abstraction level of the programming language, abstracting from a specific implementation of object creation. Objects which are not (yet) created never play any role. The corresponding proof theory is discussed and justified formally by soundness theorems. The usage of the assertion language and proof rules is illustrated with an example of a linked list reachability property. All proof rules presented are fully implemented in a version of the KeY verification system for Java programs.

Communicated by Prof. Einar Broch Johnsen and Luigia Petre.

This research is partly funded by the EU project FP7-231620 HATS: Highly Adaptable and Trustworthy Software using Formal Models and the EU project FP7-610582 ENVISAGE.

S. de Gouw · F. de Boer
CWI, Amsterdam, The Netherlands

S. de Gouw
SDL Fredhopper, Amsterdam, The Netherlands
e-mail: cdegouw@cwi.nl

F. de Boer
Leiden University, Amsterdam, The Netherlands
e-mail: frb@cwi.nl

W. Ahrendt
Chalmers University of Technology, Göteborg, Sweden
e-mail: ahrendt@chalmers.se

R. Bubel (✉)
Technische Universität Darmstadt, Darmstadt, Germany
e-mail: bubel@cs.tu-darmstadt.de

Keywords Specification · Verification · Program logic · Dynamic logic · Object creation

1 Introduction

Program verification is often based on a weakest precondition calculus, which computes for a given post-condition the weakest requirements on the initial state of a program. The original formulation given by Hoare [32] does not provide an algorithmic description of the computation. This issue was resolved by Dijkstra in [25], where he proposes a weakest precondition calculus amenable to be cast into algorithmic form. The suggested approach takes a program and its post-condition as input. The weakest precondition is computed by going backward through the program and transforming the desired post-condition step-by-step until the beginning of the program is reached.

The approach has some drawbacks because of its backward processing order. Used in a semiautomatic or interactive verification environment, it has a negative impact on the understandability of the proof situation. We claim that a calculus which follows the normal execution order of the program is easier to understand whether interaction is required. To remedy this situation, we integrate a weakest precondition calculus with symbolic execution. This allows us to compute the weakest precondition in a forward manner and makes the calculus behave like a symbolic interpreter. Integrating symbolic execution and the computation of a weakest precondition calculus allows us to use our framework in a number of areas that use symbolic execution as basis like automatic test generation and offers additional benefits like improved precision.

Symbolic execution is a static analysis technique that goes back to the seminal work of [18,36]. Symbolic execution

allows us to explore all possible execution paths of a program up to a fixed finite depth (to deal with unbounded loops or recursions, loop invariants or method contracts/summaries can be used). This is achieved by executing the program on symbolic instead of concrete input values. In the last decade, symbolic execution underwent a revival resulting in the creation of efficient symbolic execution engines for real world programming languages [11,19,24]. Symbolic execution has been applied to a wide variety of application scenarios such as test case generation or debugging [36] and formal verification of programs against a functional property [18]. For the latter, program annotations in the form of loop invariants and method contracts are necessary.

A systematic comparison of symbolic execution versus verification condition generation has been presented in [35]. The authors identified significant performance advantages for symbolic execution approaches in case of larger programs. Although the results cannot directly be transferred to our approach, we expect that several of the observations concerning performance still hold. Our integration of weakest precondition computation and symbolic execution is seamless, i.e., we perform the logical reasoning and symbolic execution within the same logic framework. Symbolic execution is performed in terms of symbolic execution rules of the logic calculus. The symbolic execution rules compute basically the path conditions, which is highlighted as one advantage of symbolic execution in the cited paper. In addition, the authors identified as a drawback that their symbolic execution engine dealt with implications using case distinctions only, while the theorem prover could apply more advanced techniques. Again in our case, the theorem prover itself performs the symbolic execution. Hence, we can continuously and seamlessly interleave first-order reasoning to rule out infeasible code branches and perform first-order reasoning to simplify path conditions and to achieve a more compact heap representation during symbolic execution. Thus, our approach is able to combine the advantages of both techniques. On the other side, as we basically interpret the program in a more general framework, we might not be as fast as a specialized source or even byte code symbolic interpreter.

Our integration is unique in that we achieve a fully abstract calculus, which allows us to verify properties at the abstraction level of the programming language. In object-oriented programming languages such as Java, objects as first-class citizens in the domain of values introduce a general mechanism of indirection. This high-level mechanism of indirection abstracts 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 user-defined operations (i.e., methods) that can be performed on objects. The Java language itself provides only the following built-in operations on objects:

Object Operators in Java	
"instanceof"	To test whether an object is of a specified type
"new"	To create a new instance of a specified type
"."	To obtain the value of the specified field of the specified object
"=="	To test equality between object references
"!="	To test inequality between object references
"?:"	Conditional expression
"(Type)"	Cast operator to covert the type of the specified object

Moreover, these operations can only be performed on the created objects; the objects not (yet) created do not exist and therefore can also not be referred to by *any* programming construct. This ensures memory safety, relieves the programmer of (error-prone) manual memory management, helps portability and allows compiler optimizations to freely move objects in memory. For practical purposes, it is important to be able to specify and verify properties of objects at the abstraction level of the programming language, following Wittgenstein [53]:

Whereof one cannot speak, thereof one must be silent.

In [22], a Hoare logic is presented to verify properties of 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. This abstraction requires new techniques for computing the weakest precondition of object creation statements because in the state *prior* to the creation of the object this new object does *not* exist so that we cannot refer to it. Note that we thus cannot obtain simply the weakest precondition as usual by substitution in the post-condition of the variable to which the new object is assigned because there does not exist a term for it in the state prior to the creation of the object! Moreover, because of the abstraction level in the assertion language, *quantification* over objects only involves *created* objects. Consequently, the *scope* of the quantifiers is also affected by object creation which has therefore to be taken into account by the weakest precondition calculus.

The main contributions of this paper are (i) the introduction of a new formalization of a weakest precondition calculus for abstract object creation in first-order dynamic logic *and* (ii) its integration with symbolic execution to achieve a forward reasoning style in the sense that the calculus behaves like a symbolic interpreter of the program to be verified.

This logic allows for the specification and verification of object-oriented programs at an abstraction level which coincides with that of the object-oriented programming language Java. That is, besides the above operations on objects,

the logic only supports quantification over created objects, including array objects. Consequently, objects not (yet) created cannot be referred to by *any* construct in the logic. We show that the standard *first-order* sequent calculus of this logic that forms the basis of the KeY theorem prover [11] adequately captures the abstract data type of objects and allows us to generate the verification conditions in standard first-order logic.

Furthermore, we extend the dynamic logic with auxiliary variables. Since arrays in Java are also objects, this indirection allows for the first-order specification of properties of object structures which cannot be expressed directly in first-order logic such as reachability.

The KeY verification system has been applied to a number of industrial case studies: In [45] KeY was used to reason about security properties of a realistic commercial demonstration application of an electronic purse (*Demoney*), which has been provided by Trusted Logics. The paper [46] describes how KeY has been successfully used to fully verify a reference implementation of the JavaCard API. Recently, the test generation facility of KeY, which generates test cases from proof attempts, was able to reveal a bug in a commercial real-time Java library (see [4]).

In [3], co-authored by some of the authors of this paper, a program logic using abstract object creation instead of activation has already been introduced. We extend this initial work to classes, recursive methods, arrays, dynamic binding and failures. Furthermore, we provide and discuss tool support—all proof rules presented in this paper have been fully implemented in a special version of KeY—by means of an example of a queue. The new tool is evaluated based on a comparison with a traditional object activation style proof of the same example. Finally, we show how to symbolically execute abstract object creation in KeY.

Related work

Specification languages like the Java Modeling Language (JML) [40] and the Object Constraint Language (OCL) [47] also abstract from the underlying representation of objects. In contrast, all known tools (to the best of our knowledge) for the (deductive) verification of object-oriented programs are based on some explicit representation of objects, e.g., objects are represented by natural numbers and counters are used to model object creation. Such an explicit representation can be useful in the context of programming languages that support for example pointer arithmetic, but it does not comply with the abstract data type of objects as provided by object-oriented languages such as Java. In fact, it complicates formal proofs of soundness because of the complex relation between the heap and its logical representation. Furthermore, we show in this paper that also the verification engine itself does not require such an explicit (internal) representation.

Our basic approach to abstract object creation provides a solid basis for the integration and extension of many other important proof-theoretical object-oriented concepts like invariants [10,33], modularity [39], dynamic frames [50] and behavioral subtyping [5,44]. Furthermore, in [2], our basic approach to abstract object creation has been integrated in a proof theory of multi-threaded programs.

Pure first-order logic without auxiliary variables is not expressive enough to assert for example reachability properties. To partially remedy this shortcoming, in [6], a formalization of abstract object creation is given using inductively defined predicates (so-called *pure* methods). Such predicates are also typically used in specifying the footprint of a program, which is crucial in tools based on separation logic (VeriFAST [34], jStar [26], Slayer [15] and Smallfoot [14]) to facilitate local reasoning. Separation logic adds several non-standard logical connectives to reason about heap properties, such as the separating conjunction and the points-to predicate. These connectives support modularity; however, in general, they seriously complicate proof theory: Calcagno et al. prove in [20] that the points-to predicate cannot be axiomatized. Another approach is taken by Why3 [28], PVS [52] and Isabelle [38], which generate verification conditions in standard higher-order logic (without introducing non-standard connectives). However, in general, higher-order logic also complicates proof theory (in contrast to first-order logic, the validities are not recursively enumerable), and thus, a high degree of automation is harder to obtain.

Chalice [42,43] is a programming language for the specification and verification of multi-threaded programs. In Chalice, reasoning about linked data structures is done via data abstraction, i.e., the internal linked data structure is mapped via model fields/ghost fields to a suitable abstract data type. The primary focus of Chalice is on concurrency properties like absence of data races and deadlocks.

Concerning object creation, Chalice uses object activation underneath, though certain aspects have been abstracted away like the order of activation. In contrast to our approach, their domain stays constant and object creation is modeled as follows: The semantics maintains a partial function Ω which maps object references to environment lists. An environment is a message, and an object is called allocated if it is in the domain of Ω. On the semantics level, they always need to quantify over the objects which are in the domain, i.e., $\forall o.(o \in dom(\Omega) \rightarrow \phi)$, to assert that a property ϕ holds for all allocated objects. These guards are not necessary in our approach as unallocated objects do not exist at all and thus cannot be quantified over in the first place.

TACO [29] is a tool for bounded program verification of JML annotated Java programs based on SAT solving. They also use object activation to model object creation.

To the best of our knowledge, our extension of KeY is the first tool which supports abstract object creation as described

above. A detailed description of other tools is beyond the scope of this paper; however, the following summary of the results of recent competitions is of interest.

The general structure of the competitions was to provide a number of assignments with each assignment consisting of

- a description of the algorithm in pseudocode or a general problem description, which had to be implemented in a programming language of the teams choice, and
- a natural language description of the desired properties to be specified and verified.

The focus of the competitions was slightly different. While the emphasis for the competitions VSTTE'11 [37] and FoVeOOS'11 [16] was on verifying program correctness, VSTTE'12 moved the focus more toward algorithm verification. The competitions showed that verifiers using abstract programming languages did best as they could use abstract data types directly in the programming language or at least could map linked data structures easier to abstract data types than systems for languages like Java. KIV, which won VSTTE'12 (together with ACL2), is also based on dynamic logic like KeY, but is based on abstract data types. They provide support for several languages including Java and an ASM style programming language. The Java back-end was, e.g., only used for the first problem in VSTTE'12 (which was also solved by KeY). While for the remaining problems (complex heap structures), the more abstract ASM style language was used. Another tool that did very well was Dafny [41], which uses its own (more abstract) object-oriented language and whose specification and reasoning system implement dynamic frames. KeY was competitive in VSTTE'11 and FoVeOOS'11, but it did not so well in VSTTE'12 due to its heavier focus on algorithm verification. KeY was able to prove all problems of the first two competitions after the competition using a pre-release version of KeY 2.0 with explicit support for dynamic frames. Using that version, most (except one) problems of VSTTE'12 could be solved. The only time a system based on separation logic participated was VeriFAST [34] at VSTTE'11. Their specification of some of the problems was elegant, but automation was problematic. They managed to solve the problems in the aftermath but indicated that labor-intensive work was necessary.

Outline

In Sect. 2, we introduce a dynamic logic for an object-oriented language with recursive methods and object creation. Other features are not part of this core language, but in Sect. 3, we demonstrate how to handle them using a transformational approach. We present the axiomatization of the language in terms of the sequent calculus given in Sect. 4. To reason about formulas containing updates (which arise from assignments and object creations in the sequent calculus), we define in Sect. 5 an inductive rewrite relation which simplifies such formulas to standard first-order logic formulas (without updates). With the calculus at hand, symbolic execution of programs is described in Sect. 6. Finally, we illustrate our approach on a typical case: inserting an element in a queue in Sect. 7. After a discussion of our approach in Sect. 8, we conclude with Sect. 9.

2 Dynamic logic

In this section, we give the syntax and semantics of our programming language and specification language (dynamic logic or DL). We start here with an informal introduction to DL. The formal definitions, including those for the program semantics and concepts such as model and domain, follow in the subsequent subsections.

DL is a variant of *modal logic* which extends and generalizes Hoare logic, e.g., it allows the direct expression of program equivalence and weakest preconditions. Different parts of a formula are evaluated in different worlds (models), which vary in the interpretation of, in our case, program variables, fields and the underlying domain of created objects (which changes due to execution of object creations). DL extends full first-order logic with two additional (mix-fix) operators: $\langle\,.\,\rangle\,.$ (diamond) and $[\,.\,]\,.$ (box). In both cases, the first argument is a *statement*, whereas the second argument is another DL formula. A formula $\langle p\rangle\phi$ is true in a model M if execution of p (starting from the model M) can terminate in some model where ϕ is true. The formula $[p]\phi$ is true in a model M if execution of p, when started in M, *either* does not terminate *or* always results in a model in which ϕ is true. For deterministic programs, the only difference between the two operators is whether termination is claimed (i.e., the difference between partial and total correctness). DL 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.

An example formula involving object creation is $\forall l.\ \langle u := \mathsf{new}\,\rangle\neg(u = l)$. 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 $u := \mathsf{new}\,$. We do not have the constant domain assumption here: Object creations modify the domain by adding the new object (see Sect. 2.1.3 for a detailed formal semantics of object creations). 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, that formula is false (cf. Sect. 5.2). These examples also illustrate a further advantage of DL over Hoare logic: In the presence of abstract creation, it allows for

a direct logical expression of the dynamic scope of the object quantifiers (as illustrated above).

All major program logics (Hoare logic, wp calculus, DL) 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* (called *"updates"*) in the logic. Updates are of the form $\{loc := r\}$, where loc is a location in the given language and the right-hand side r determines the value to be stored. The semantics of updates are the same as those for the corresponding assignments. The presence of explicit substitutions in the logic has a number of advantages. First of all, it allows us to disentangle the resolving of an assignment and the application of the resulting substitution. Here, both steps can happen at different times during the process of building a proof. Second, our approach to object creation demands a non-standard way of applying substitutions, and updates are a vehicle to treat this issue explicitly on the level of the syntax (see Sect. 5). And at last, to enable symbolic execution in a forward direction, we will in Sect. 6 generalize the notion of updates, allowing the parallel composition of elementary updates to accumulate the effect of (a prefix of) a program. With that, updates become the cornerstone in integrating deductive verification and symbolic execution for abstract object creation.

Therefore, our DL is extended with updates. We close expressions and formulas under update prefixing as follows: If e' is an expression, then so is $\{loc := r\}e'$, and if ϕ is a DL formula, then so is $\{loc := r\}\phi$. Note that this construction is part of the overall recursive definition of terms and formulas, such that updates can appear not only at the top level of formulas or expressions, but also nested inside. For instance, $0 < v \rightarrow \{u := v\}\,(0 < u)$ is a formula, and $e_1 + \{u := v\}e_2$ is an expression.

In the language considered in this paper, assignments are "simple enough" to be turned into updates directly, once the assignment is in the focus of the proof step to be performed. (This is not the case for many real world languages, where expression evaluation may have side effects, may pass control to a different method and may throw exceptions. In KeY style logics for such languages, like DL for Java [12], complex assignments are flattened to simple assignments before turned into updates). The language considered in this paper allows the following forms of assignments: $u := \mathsf{new}$, $u := e, e_1.x := e_2$ and $e_1[e_2] := e_3$. They can all be turned into explicit substitutions without modifying them, for instance when transforming the formula $\langle loc := r; s\rangle\phi$ into the formula $\{loc := r\}\langle s\rangle\phi$ during symbolic execution. We therefore use the same syntactic category for both, $loc := r$ within the $\langle\rangle$ modality and within the $\{\}$ modality, see the next section. Calling updates "explicit substitutions" refers to the intuition that they are substitutions which have not been applied yet, but will be later in the proof. However, locally within individual formulas, the updates act more like an additional modality, both syntactically and semantically. In the following, we will use the terminology "update" in an overloaded manner by referring to both, the modality $\{loc := r\}$ and the $loc := r$ contained in the modality. A full account of KeY style DL can be found in [12].

2.1 Language

We introduce here a core class-based object-oriented language with support for recursive methods, unbounded arrays and object creation. A corresponding assertion language is presented which is based on first-order dynamic logic.

2.1.1 Types and declarations

The language is strongly typed and contains the primitive types Nat and Boolean. Additionally, there are user-defined class types C, predefined class types $T[\,]$ of *unbounded* arrays in which the elements are of type T and a union type Object. Arrays are dynamically allocated and are indexed by natural numbers. Multi-dimensional arrays are modeled (as in Java) as arrays in which the elements themselves are arrays. For instance, Nat[ ] is the type of one-dimensional arrays of natural numbers, and Nat[ ][ ] is the type of two-dimensional arrays of natural numbers. We assume a transitive reflexive subtype relation between types. Object is the super type of any class type. We will not be concerned with type checking, but only note that it can be done statically and is orthogonal to the semantics of the language.

There are three kinds of declarations: variable declarations, field declarations and method declarations. A variable declaration associates a type to a name. If the type is not an array type, we call the variable a simple variable. Arrays are assumed to be unbounded, but we show how bounded arrays can be handled by a transformation in Sect. 3.

Declarations of fields of a class C associate a type $\mathrm{C} \rightarrow T$ to a field name. Fields can thus be seen as mappings from objects of C to values.

A method declaration $m(\mathsf{this}, u_1, \ldots, u_n) :: s$ associates the method parameters as a list of variables $\mathsf{this}, u_1, \ldots, u_n$ and a statement s (its *body*) to a method name m. Unlike in Java, methods do not return anything. Return values can be simulated using variables. The first formal parameter of each method is the special variable this. It stores the currently executing object and is special since it is read-only (assignments to this are not allowed).

We do not represent declarations explicitly in the language syntax, which we are about to introduce. Instead, we assume to be given a set Var of variable declarations and for each class C a set F_{C} of fields and M_{C} of methods. Var is partitioned into a set PVar of program variables and a set LVar of logical variables. Logical variables do not occur in pro-

grams. They are used in dynamic logic formulas to express properties of programs and can be quantified over. The set of program variables consists of both local and global variables. For technical convenience, we restrict local variables to formal parameters (for a treatment of blocks, we refer to [9]).

2.1.2 Syntax

Expressions of our language are side effect free. The following grammar generates the language of expressions:

$$e ::= u \mid e.x \mid \mathsf{null} \mid e_1 = e_2 \mid \mathsf{if}\, b\, \mathsf{then}\, e\, \mathsf{fi} \mid \mathsf{if}\, b\, \mathsf{then}\, e_1\, \mathsf{else}\, e_2\, \mathsf{fi} \mid e_1[e_2] \mid f(e_1, \ldots, e_n) \mid C(e) \mid \{\mathcal{U}\}e$$

Variables are indicated by the typical element u, and x is a typical field. Since u and x can be of an array type, this implies that arrays are also expressions. A dot denotes dereferencing, i.e., $e.x$ is the value of the field x of the object to which e points. This is syntactic sugar for $x(e)$ since fields will be considered to be functions from objects to values (this reflects the fact that field types have the form $\mathrm{C} \to T$). The expression null 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. For object expressions, this means that we use the Java reference semantics: The expressions must denote the same object. It is therefore possible for two expressions of type object to be unequal even if all fields of the objects they denote have the same value. The expression $\mathsf{if}\, b\, \mathsf{then}\, e\, \mathsf{fi}$ has value e if the Boolean expression b is true; otherwise, it has an arbitrary value. This expression allows a systematic approach to prove properties about partial functions. A conditional expression is denoted by $\mathsf{if}\, b\, \mathsf{then}\, e_1\, \mathsf{else}\, e_2\, \mathsf{fi}$. The motivation for including it in our core language is that it significantly simplifies treatment of failures (Sect. 3) and aliasing (Sect. 5). If e_1 is an expression of type $T[\,]$ and e_2 is an expression of type Nat, then $e_1[e_2]$ is an expression of type T. Here, T itself can be an array type. For example, if a is an array variable of type Nat[ ][ ], then the expression $a[0]$ denotes an array of type Nat[ ]. The function $f(e_1, \ldots, e_n)$ denotes an arithmetic or Boolean operation of arity n. For class types C, the Boolean expression $C(e)$ is true if and only if the (dynamic) type of e is exactly C. We restrict ourselves to the following operations on expressions of a class type: comparing for equality, dereferencing, accessing as an array if the object is of an array type or used as argument of a class predicate, if-then expression or conditional expression. The last expression form ($\{\mathcal{U}\}e$) is not allowed in programs, i.e., it appears only in formulas outside programs (This form is mainly used for defining the application of updates on formulas recursively via the application of updates on expressions, see Sect. 5).

The language of statements is generated by the following grammar:

$$s ::= \mathsf{skip} \mid s_1; s_2 \mid \mathsf{if}\, b\, \mathsf{then}\, s_1\, [\,\mathsf{else}\, s_2\,]\, \mathsf{fi} \mid \mathsf{while}\ e\ \mathsf{do}\ s\ \mathsf{od} \mid m(e_1, \ldots, e_n) \mid \mathcal{U}$$
$$\mathcal{U} ::= u := \mathsf{new} \mid u := e \mid e_1[e_2] := e \mid e_1.x := e$$

By skip, we denote the empty statement. A semicolon denotes sequential composition. Conditional branching is denoted by $\mathsf{if}-\mathsf{then}-\mathsf{else}-\mathsf{fi}$. If the optional else is not present and the condition b holds, s_1 is executed. If b is false, this statement causes a failure. The statement while denotes the usual looping. The condition for both looping and branching is given by a Boolean expression. In a method call $m(e_1, \ldots, e_n)$, the first actual parameter e_1 denotes the callee. Method calls provide the only way to transfer control from the current object to another (the callee). Updates are generated by the productions for $\mathcal{U}$. The first $u := \mathsf{new}$ assigns to the program variable u a newly created object of the declared type (possibly an array type) of u. Objects are never destroyed. The next three updates denote assignments of an expression e to a program variable u, a *subscripted* variable $e_1[e_2]$ (where e_1 must have an array type and e_2 a natural number) and a field x, respectively. For technical convenience only, we do not have assignments $e_1[e_2] := \mathsf{new}$ and $e_1.x := \mathsf{new}$. We reason about such assignments in terms of the statement $u := \mathsf{new}; e_1[e_2] := u$, where u is a fresh program variable, to separate object creation from the aliasing problem. Following standard practice, assignments are well-typed when the type of the value is a subtype of the type of the variable to which it is assigned. We assume that every statement and expression to be well-typed. A *program* in our language consists of a statement (referred to as the main statement) together with sets for variable declarations, field declarations and method declarations.

Dynamic logic formulas are built from the Boolean expressions and statements defined above. Formally, formulas are generated by the following grammar:

$$\phi ::= b \mid \phi_1 \wedge \phi_2 \mid \phi_1 \vee \phi_2 \mid \phi \to \phi_2 \mid \exists l : \phi \mid \forall l : \phi \mid \{\mathcal{U}\}\phi \mid [s]\phi \mid \langle s \rangle \phi$$

In this grammar, b is a Boolean expression, s is a statement, $\mathcal{U}$ is an update, and l is a logical variable of any type of our core language. The meaning of such formulas was already described informally in the beginning of this section and will be formalized below.

Conventions We use $\equiv$ for *syntactic* equality. For expressions e and statements s, $var(e)$ and $var(s)$ denote the sets of variables that appear in them. This set is extended to sets of method declarations in a point-wise manner: For a set of method declarations M_{C}, $var(M_{\mathrm{C}})$ is the set of all variables that appear in some method body of the methods in M_{C}. The set change(s) contains the variables that appear on the left-hand side of an assignment in s. For a method declaration $m(u_1, \ldots, u_n) :: s$, change(m) is defined as change(s)$\backslash\{\mathsf{this}, u_1, \ldots, u_n\}$. This reflects the fact that

the formal parameters are not changed by method calls, since they are reset to their initial values. For a set M_C of method declarations, change(M_C) is the union of all sets change(m) with $m \in M_C$. We abbreviate the set of all method declarations of all classes by D and extend var and $change$ to D in the obvious way. The set $free(\phi)$ contains all variables that occur free in the formula ϕ.

2.1.3 Semantics

Before the semantics of statements and expressions of our programming language can be defined, a meaning must be assigned to the non-logical symbols of our language. For reference, we list all non-logical symbols here:

1. The constants (functions of arity 0): null_C (of type C), 0 (Nat) and true and false (Boolean).
2. Arithmetical operations. We assume at least the successor function, addition and multiplication.
3. Boolean operations (at least conjunction, disjunction and negation).
4. For each type T the conditional $(\,\mathsf{if}\, b\, \mathsf{then}\, e_1\, \mathsf{else}\, e_2\, \mathsf{fi}\,)_T$.
5. For each type T: $(\,\mathsf{if}\, b\, \mathsf{then}\, e_1\, \mathsf{fi}\,)_T$.
6. For each array type $T[\,]$ an access function $[\,]_{T[\,]}$ of type Nat $\to (T[\,] \to T)$.
7. For every class C a unary predicate C of type Object $\to$ Boolean.
8. Variables declared in Var.
9. For every class C, its fields declared in F_C.

The basic notion underlying the semantics of both the programming language and the assertion language is that of a many-sorted structure of the form

$$\left(\bigcup_T dom(T),\, I\right)$$

consisting of a disjoint union of domains $dom(T)$ for each type T and an interpretation I. The interpretation assigns a total function (a meaning) to each function symbol and a relation to each relation symbol. Thus, for example, for the constant null_C, the interpretation assigns an element of dom(C) to it. We omit explicit type annotations of null and the conditional expression if no confusion occurs. What we call a model is sometimes called a model plus a valuation (i.e., an assignment of the variables).

A *model* M for our language is a structure that satisfies the axioms:

- Arithmetical operations satisfy the theory of Peano arithmetic.
- Boolean operations satisfy the axioms for Boolean algebras.
- Each class predicate C satisfies $C(\mathsf{null})$.
- If o_1 and o_2 have the same type T, then the conditional satisfies $(\,\mathsf{if}\, \mathsf{true}\, \mathsf{then}\, o_1\, \mathsf{else}\, o_2\, \mathsf{fi}\,)_T = o_1$ and $(\,\mathsf{if}\, \mathsf{false}\, \mathsf{then}\, \mathsf{else}\, \mathsf{fi}\, o_1 o_2)_T = o_2$.
- If o and o' have the same type T, then the if-then expression obeys $\mathsf{if}\, \mathsf{true}\, \mathsf{then}\, o\, \mathsf{fi}_T = o$ and $\mathsf{if}\, \mathsf{false}\, \mathsf{then}\, \mathsf{fi}\, o_T = \mathsf{If false then}\, o'\, \mathsf{fi}_T$.
  This axiom expresses that if the first argument is false, the whole expression always has the same (arbitrary) value, regardless of the value of the second argument.

The above first-order axioms are not categorical: There are multiple structures in which all of the above axioms are true. We stipulate that *any* structure that satisfies the axioms is a model for our language, including those structures in which for example the arithmetical operations have a non-standard interpretation. For notational convenience, we write $M(s)$ instead of $I(s)$ for the interpretation of the non-logical symbol s in the model M and use the abbreviation $M(T)$ for the phrase "$dom(T)$ in M." If u is declared in Var as a variable of type T, it is interpreted as an individual of the domain $M(T)$. A field $x \in F_C$ of type C $\to T$ is interpreted as a unary function M(C) $\to M(T)$. Array access functions $[\,]_{T[\,]}$ are interpreted as binary functions M(Nat) $\to (M(T[\,]) \to M(T))$. Thus, array indices can be seen as fields.

Semantics of expressions and statements The meaning of an expression e of type T is a (total) function $[\![e]\!]$ that maps a model M to an individual of $M(T)$. This function is defined by induction on e. Here are the main cases:

- $e \equiv e_1.x$: $[\![e]\!](M) = M(x)([\![e_1]\!](M))$.
- $e \equiv e_1[e_2]$: $[\![e]\!](M) = M([\,]_{T[\,]})([\![e_2]\!](M))([\![e_1]\!](M))$ where e_1 has the array type $T[\,]$ and e_2 has type Nat.
- $e \equiv C(e_1)$: $[\![e]\!](M) = \mathsf{true}$ iff $[\![e_1]\!](M) \in M(C)$
- $e \equiv \mathsf{if}\, b\, \mathsf{then}\, e_1\, \mathsf{else}\, e_2\, \mathsf{fi}$:

$$[\![e]\!](M) = \begin{cases} [\![e_1]\!](M), & \text{if } [\![b]\!](M) \\ [\![e_2]\!](M), & \text{otherwise} \end{cases}$$

- $e \equiv \mathsf{if}\, b\, \mathsf{then}\, e_1\, \mathsf{fi}$:

$$[\![e]\!](M) = \begin{cases} [\![e_1]\!](M), & \text{if } [\![e_1]\!](M) \\ def_T, & \text{otherwise} \end{cases}$$

Note that for the last two cases, the left-hand side denotes a syntactical expression and the right-hand side denotes the corresponding (semantic) function as interpreted in the given model. It is easily checked that this semantics satisfies the axioms for models. One feature of our semantics is that since the meaning function is total, some meaning is also

assigned to the expression $\mathsf{null}.x$. In other words, no failures occur at the expression level; instead, execution continues with unpredictable behavior. A different semantics of expressions, which can cause failures, can be simulated by means of a program transformation (see Sect. 3).

Statements in our language are deterministic and can fail ($\mathsf{if} - \mathsf{then} - \mathsf{fi}$) or diverge. We define the meaning of a statement in terms of a small-step operational semantics and use the (quite common) notation

$$\langle s, M\rangle \to \langle s', M'\rangle$$

to express that executing s in the model M reduces to executing s' in the model M'. We use $\to^*$ for the reflexive and transitive closure of $\to$. By

$$\langle s, M\rangle \to M',$$

we express that executing s in the model M terminates in the model M'. Since calls can appear in statements, the definition of the transition relation depends in general on the method declarations. For an $\mathsf{if}\, b\, \mathsf{then} - \mathsf{fi}$-statement, there is no next step if b is false: The execution "hangs" and there are no s' and M' (not even an empty s') such that $\langle \mathsf{if}\, b\, \mathsf{then}\, s\, \mathsf{fi}, M\rangle \to \langle s', M'\rangle$. Throughout execution, assignments to variables and fields change the model in the interpretation of the variables and fields, respectively. The interpretation of the array access function changes due to assignments to subscripted variables. Moreover, during execution, the domain $dom(\mathrm{C})$ for instances of C is extended with new objects by object creations $u := \mathsf{new}$. Consequently, all functions that take an argument of C (including conditional expressions and $\mathsf{if} - \mathsf{then} - \mathsf{fi}$) must be extended appropriately for the new object. Statements do not affect the domains of natural numbers and Booleans, and the interpretation of other non-logical symbols. In summary, statements map models to models, potentially changing variables, array access functions, fields and the domains for class types, leaving the interpretation of functions and relations that do not take arguments of a class type fixed.

The meaning of normal assignments, conditional statements and while loops is defined in the standard way. Therefore, we focus on the semantics of object creation, field assignments and method calls. Since object creation involves initialization of the fields of the new object, we first define for each type a default value: $init_{\mathrm{Nat}} = 0$, $init_{\mathrm{Boolean}} = \mathsf{false}$ and $init_{\mathrm{C}} = \mathsf{null}$. Furthermore, for the selection of a new object of class C, we use a choice function ν on a model M and class C to get a fresh object $\nu(M, \mathrm{C})$ of class C which satisfies $\nu(M, \mathrm{C}) \notin M(T)$ for any type T (in particular, $\nu(M, \mathrm{C}) \notin M(\mathrm{C})$). Clearly, without loss of generality, we may assume that $\nu(M, \mathrm{C})$ only depends on $M(\mathrm{C})$ in the sense that $\nu(M, \mathrm{C}) = \nu(M', \mathrm{C})$ if $M(\mathrm{C}) = M'(\mathrm{C})$. It is worthwhile to observe that this choice function preserves the deterministic nature of our core language. Non-deterministic (or random) selection of a fresh object would require reasoning semantically up to a notion of isomorphic models which would unnecessarily complicate soundness proofs. Note furthermore that the above definition of the choice function follows the general and standard approach in formal semantics of the introduction of fresh variables in language like the π-calculus or Prolog, where these variables are selected from some domain which does not appear in the actual configuration. The semantics of an object creation statement $u := \mathsf{new}$, where u has type C, can now be defined as follows:

$$\langle u := \mathsf{new}, M\rangle \to M'$$

where, for $o = \nu(M, \mathrm{C})$, M and M' only differ with respect to the following clauses:

1. $M'(\mathrm{C}) = M(\mathrm{C}) \cup \{o\}$.
2. $M'(u) = o$.
3. $M'(x)(o) = init_T$ for any field x of type $\mathrm{C} \to T$.
4. $M'(f) = M(f)$, for every arithmetic and Boolean operation f.
5. $\mathsf{if\ false\ then}\, o\, \mathsf{fi} = \mathsf{if\ false\ then}\, o'\, \mathsf{fi}$, for every $o' \in M(\mathrm{C})$.

This semantics for object creation corresponds to the intuition that an object creation first adds a new object to the domain for the class C, then sets the values of the fields in the new object to their default value and finally assigns the new object to u. Note that the above semantics ensures that the if-then with false as its first argument and any arbitrary object as its second argument (even the newly created object) always denotes the same arbitrary object as the one in the model before executing the object creation.

The semantics of an array creation

$$\langle u := \mathsf{new}, M\rangle \to M'$$

where u has type $T[\,]$ is similar to the above: for $o = \nu(M, T[\,])$, M and M' only differ with respect to the following clauses:

1. $M'(T[\,]) = M(T[\,]) \cup \{o\}$.
2. $M'(u) = o$.
3. $M'([\,]_{T[\,]})(n)(o) = init_T$ for all $n \in M(\mathrm{Nat})$.
4. $M'(f) = M(f)$, for every arithmetic and Boolean operation f.
5. $\mathsf{if\ false\ then}\, o\, \mathsf{fi} = \mathsf{if\ false\ then}\, o'\, \mathsf{fi}$, for every $o' \in M(T[\,])$.

The third clause states that all elements in the array are initialized to their default value.

Assignments to fields are executed as follows:

$$\langle e.x := e_1, M\rangle \to M'$$

where $M'(x)([\![e]\!](M)) = [\![e_1]\!](M)$ and the domains and the interpretations of the other non-logical symbols in M' are the same as in M. Assignments to an array element are defined analogously and therefore omitted.

For method calls, we use a copy rule: Whenever a call is encountered, the program proceeds by executing the method body. The method body is chosen statically based on the class type of the parameter this (we show in Sect. 3 how dynamic binding can be supported). The values of the actual parameters are evaluated in parallel and assigned to the formal parameters before the method body executes. Directly after the call succeeds, the formal parameters are restored to their initial values. This leads to the following rule:

$$\frac{\langle s, M'\rangle \to \langle s', M''\rangle}{\langle m(e_1, \ldots, e_{n+1}), M\rangle \to \langle s, M'''\rangle}$$

where

- the method m is declared as $m(\mathsf{this}, u_1, \ldots, u_n) :: s$,
- M' differs from M only as follows:
  - $M'(\mathsf{this}) = [\![e_1]\!](M)$
  - $M'(u_i) = [\![e_{i+1}]\!](M)$, for $1 \le i \le n$
- and M''' differs from M'' only as follows:
  - $M'''(\mathsf{this}) = M(\mathsf{this})$
  - $M'''(u_i) = M(u_i)$, for $1 \le i \le n$.

Intuitively, this corresponds to defining the operational meaning of calls in three steps: In the first step, M' is obtained by assigning the actual parameters to the formal parameters. In the second step, the actual method body is executed in M', resulting in M''. In the final step, M''' is obtained by resetting the formal parameters to their initial values.

Semantics of formulas Since calls may occur inside the modal operators of a dynamic logic formula, the truth of a dynamic logic formula ϕ depends in general on a set of method declarations to bind calls to the right method body. Given a set of method declarations, we write $M \models \phi$ if the formula ϕ is true in the model M. A formula ϕ is valid if $M \models \phi$ holds for every model M.

The modal operators have their usual semantics:

$M \models [s]\phi$ iff $M' \models \phi$ for all M' such that $\langle s, M\rangle \to M'$

and

$M \models \langle s\rangle\phi$ iff $M' \models \phi$ for some M' such that $\langle s, M\rangle \to M'$.

Interestingly, even though we allow quantification over arrays, all assertions are *first-order* dynamic logic formulas because arrays are objects which are used to represent sequences, but are not sequences themselves. For example, if s has an array type, $\exists s : s[0] = 0$ expresses that there exists an array object which points to a sequences whose first element is 0. This modeling of arrays as pointers (to sequences) is also taken by Java.

In general, for a logical variable l of type T, we have the following semantics of existential quantification

$M \models \exists l.\phi$ iff for some $o \in M(T) : M' \models \phi$.

where M' differs from M only in $M'(l) = o$. The semantics of universal quantification can be derived using the equivalence $(\forall l : \phi) \leftrightarrow (\neg\exists l : \neg\phi)$. Note that the *extensionality* axiom $\forall a, b, n : a[n] = b[n] \to a = b$ for arrays is not valid. In contrast, this axiom clearly holds if a and b range over sequences.

As an illustration of the semantics for object creation and quantification, we show that the scope of quantification over objects is dynamic:

$M \models \forall l.\langle u := \mathsf{new}\rangle\neg(u = l)$
iff
for all $o' \in M(T) : M' \models \langle u := \mathsf{new}\rangle\neg(u = l)$

where M' differs from M only in $M'(l) = o'$ and l has type T. Let $o = \nu(M[l := o'], \mathrm{C})$ and suppose $\langle u := \mathsf{new}, M'\rangle \to M''$. Then, by the semantics of the diamond modality of dynamic logic and the above semantics of object creation, we conclude that

$M' \models \langle u := \mathsf{new}\rangle\neg(u = l)$
iff (semantics of object creation)
$M'' \models \neg(u = l)$
iff (compute values of u, l in the model M'')
$o \neq o'$

In the last step of the proof, we have used properties of the semantics of object creation to deduce that $M''(l) = o'$ and $M''(u) = o$. This is justified by the following line of reasoning. By definition of $o = \nu(M, \mathrm{C})$, we have $o \notin M(\mathrm{C})$. Furthermore, since $M(\mathrm{C}) = M'(\mathrm{C})$, we also have $o \notin M'(\mathrm{C})$. On the other hand, $o' \in M'(\mathrm{C})$; hence, we indeed have $o \neq o'$ for all $o' \in M(\mathrm{C})$.

3 Transformations

We show here that our core language contains suitable primitive constructs from which more intricate features can be handled by a completely mechanical translation. The features to which this transformational approach applies include failures, *bounded* arrays, inheritance and dynamic binding.

Failures A failure normally occurs (but not in the semantics described previously) in the execution of statements when executing $\mathsf{if}\, b\, \mathsf{then}\, s\, \mathsf{fi}$ if b is false, calling a method or accessing a field of the null object and when evaluating an

undefined expression (such as division by zero). To achieve this in principle, checks for definedness have to be added to all expressions and statements. This seriously obfuscates both proof rules for and semantics of programs. We solved this problem by assuming that only an execution of $\mathsf{if}\, b\, \mathsf{then}\, s\, \mathsf{fi}$ can fail.

We now define a transformation which guarantees that whenever such an "undefined" expression is evaluated in the original program, $\mathsf{if\ false\ then}\, s\, \mathsf{fi}$ is executed in the transformed program. In the definition of the transformation, we will make use of a Boolean expression $\mathrm{def}(e)$ with the following property:

$[\![\mathrm{def}(e)]\!](M)$ if and only if no failure would occur when evaluating e in M.

In principle, any definition of def which satisfies the above property will suffice, but as an example, we define here a few representative cases of the definition. A more complete treatment can be found in [21].

- $\mathrm{def}(u) \equiv \mathsf{true}$
- $\mathrm{def}(e.x) \equiv \mathrm{def}(e) \wedge e \neq \mathsf{null}$
- $\mathrm{def}(e_1[e_2]) \equiv \mathrm{def}(e_1) \wedge \mathrm{def}(e_2) \wedge e_1 \neq \mathsf{null}$
- $\mathrm{def}(f(e_1, \ldots, e_n)) \equiv \mathrm{def}(e_1) \wedge \ldots \wedge \mathrm{def}(e_n) \wedge \mathrm{def}(f)(e_1, \ldots, e_n)$
- $\mathrm{def}(\,\mathsf{if}\, b\, \mathsf{then}\, e_1\, \mathsf{else}\, e_2\, \mathsf{fi} \equiv \mathrm{def}(b) \wedge ((b \wedge \mathrm{def}(e_1)) \vee (\neg b \wedge \mathrm{def}(e_2)))$

The fourth case covers arithmetical operations which can fail, such as division. For such operations, we assume to have a condition $\mathrm{def}(f)(v_1, \ldots, v_n)$ on the arguments on which f is defined. For division, this condition is $\mathrm{def}(div)(x, y) \equiv y \neq 0$.

We are now in the position to define the transformation $\Theta(s)$ by induction on the structure of s:

- $\Theta(u{:=}e) \equiv \mathsf{if}\, \mathrm{def}(e)\, \mathsf{then}\, u{:=}e\, \mathsf{fi}$
- $\Theta(s_1; s_2) \equiv \Theta(s_1); \Theta(s_2)$
- $\Theta(\,\mathsf{if}\, e\, \mathsf{then}\, s_1\, \mathsf{else}\, s_2\, \mathsf{fi}\,) \equiv \mathsf{if}\, \mathrm{def}(e)\, \mathsf{then}\, (\,\mathsf{if}\, e\, \mathsf{then}\, \Theta(s_1)\, \mathsf{else}\, \Theta(s_2)\, \mathsf{fi})\, \mathsf{fi}$
- $\Theta(m(e_1, \ldots, e_n)) \equiv \mathsf{if}\, (\mathrm{def}(e_1) \wedge \cdots \wedge \mathrm{def}(e_n) \wedge e_1 \neq \mathsf{null}\,)\, \mathsf{then}\, m(e_1, \ldots, e_n)\, \mathsf{fi}$

The cases not covered here are trivial adaptations of the above cases. The final result is a transformed program where the above transformation is applied to the main statement of the program and to each method body.

In addition to transformations of expressions and programs, it is also possible to define a transformation on formulas, depending on the intended semantics of assertions. For instance, in the proof system we shall use in this paper, arrays will be treated in the assertion language as unbounded. Consequently, if an array of natural numbers is created (more on bounded arrays follows below) and assigned to the variable a, then afterward $a[a.length + 1] = 0$ will be provable (because natural numbers are initialized to 0), despite the fact that $a.length + 1$ lies "outside the array bounds." If instead a semantics of assertions is desired in which this formula is not provable, a transformation on assertions could be defined as: $\Theta(a[e]) \equiv \mathsf{if}\, 0 \leq e < a.length\, \mathsf{then}\, a[e]\, \mathsf{fi}$.

To describe the relation between a given program and its transformed counterpart, we define the following fail predicate on statements.

- $\mathrm{fail}(u{:=}e) \equiv \neg\mathrm{def}(e)$
- $\mathrm{fail}(s_1; s_2) \equiv \mathrm{fail}(s_1)$
- $\mathrm{fail}(\,\mathsf{if}\, e\, \mathsf{then}\, s_1\, \mathsf{else}\, s_2\, \mathsf{fi}\,) \equiv \neg\mathrm{def}(e)$
- $\mathrm{fail}(m(e_1, \ldots, e_n)) \equiv \neg(\mathrm{def}(e_1) \wedge \ldots \wedge \mathrm{def}(e_n) \wedge e_1 \neq \mathsf{null}\,)$

The intention is that $\mathrm{fail}(s)$ holds in a given model, if the execution of s in that model immediately fails. The next theorem summarizes the outcome of the above transformations:

Theorem 1 *Let $\Rightarrow$ be the transition relation restricted to non-failing models in the computation of a statement s (i.e., models in which fail(s) is false). Then, for all statements s and models M:*
if then

1. $\langle\Theta(s), M\rangle \rightarrow^* \langle s', M'\rangle$ *if and only if* $\langle s, M\rangle \Rightarrow^* \langle s'', M'\rangle$, *where* $\Theta(s'') = s'$.
2. *If there is no transition step for* $\langle s', M'\rangle$, *then* $[\![fail(s')]\!](M')$ *is* false.

Proof The proof of this theorem proceeds by induction on s.

Besides failures, other features such as dynamic binding, return values, constructors and bounded arrays (which are not strictly part of the core language) are handled by the below shallow embeddings that allow to view them as syntactic sugaring.

Dynamic binding in closed programs Note that the language already contains an inheritance relation, but we now use it to handle dynamic binding. In our semantics (described in Sect. 2.1.3), subclasses do not correspond to subsets: If C is a subclass of D, then their sets of instances $dom(C)$ and $dom(D)$ are disjoint. This, for example, allows us to define the e instanceof C operator of Java as the Boolean expression $n \neq null \wedge (C(e) \vee C_1(e) \vee \cdots \vee C_n(e)$ of all the superclasses $C_1, \ldots, C_n$ of C (note that this uses our restriction to closed programs).

To add support for dynamic binding, we first rename each method uniquely by prefixing its class: If m is a method of

class C, then its new name is $C.m$. Dynamic binding is then achieved by transforming each method call $m(s, e_1, \ldots, e_k)$ to the statement S_n, defined inductively as follows:

$S_0 \equiv \mathsf{skip}$

$S_{i+1} \equiv \mathsf{if}\ C_{i+1}(s)\ \mathsf{then}\ C_{i+1}.m(s, e_1, \ldots, e_k)\ \mathsf{else}\ S_i\ \mathsf{fi}$

where $\{C_1, \ldots, C_n\}$ is the set of subclasses of the type of s. In this scheme, we make essential use of the fact that $C(s)$ returns true only if s is exactly of type C, not if the type of s is merely a subclass of C.

Return values and constructor methods Java constructors are special methods that initialize the fields of a newly created object, do not return anything and are named after their class. In Java, the statement new C(e) initializes the fields of the new object by executing the constructor C (with actual parameters e). Constructors can be handled in our core language by the following simple transformation (the argument of Θ is a Java code fragment; the result is a code fragment of our core language):

– $\Theta(\mathsf{new\ C(e)}) \equiv u := new;\ u.C'(e)$

where u is a fresh variable of type C. Intuitively, this first creates the new object and then invokes the constructor (which is renamed to C') on it; treating it like a normal method. The renaming avoids name clashes with class predicates. Return values of methods can be simulated by introducing a fresh (global) variable *result* and assigning the return value to *result* whenever the return-statement would be executed in the original Java program. This results in the following transformation:

– $\Theta(\mathsf{return}\ e) \equiv result := e$

Bounded arrays An unbounded array a is of "unlimited" length, and hence, the expression $a[n]$ is well-defined for each natural number n. Bounded arrays are defined only on an initial segment $\{0, \ldots, b\}$ of the natural numbers. The number b is called the bound of the array. In Java, bounded arrays are created with the statement new T[n]. We model the bound by adding a read-only field $length$ of type Nat to each array type and defining the following transformation from Java to our core language:

– $\Theta(\mathsf{new\ T[n]}) \equiv a := new;\ a.length := n.$

where a is a fresh array variable of type T. The intention is that $a[k]$ is defined if and only if $0 \leq k < a.length$. If an element outside of the bounds of the array is accessed, a failure occurs. This requires only the following revision in the definition of def(e):

– $\mathrm{def}(e_1[e_2]) \equiv \mathsf{if}\ (\mathrm{def}(e_1) \wedge \mathrm{def}(e_2))\ \mathsf{then}\ (e_1 \neq \mathsf{null} \wedge 0 \leq e_2 < e_1.length)\ \mathsf{else\ false\ fi}$

Note that the length of the array is not part of the type of an array variable. This follows the usual practice in Java where the number of dimensions is part of the type of an array variable, but the length is not. Consequently arrays with different lengths can be assigned to the same array variable during a program execution.

4 Sequent calculus

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 realize that in the following proof rules, ϕ, ψ and alike match *any* formula of our logic, possibly containing programs or updates.

We follow [11,13] in presenting the proof system for dynamic logic as a Gentzen-style sequent calculus. A sequent is a pair of sets of formulas (each formula closed for logical variables) written as $\phi_1, \ldots, \phi_m \vdash \psi_1, \ldots, \psi_n$. The formulae ϕ_i on the left of the *sequent arrow* $\vdash$ are called the *antecedent*, the formulae ψ_j on the right the *succedent* of the sequent. The meaning of such a sequent is identical to the meaning of the logic formula

$$\bigwedge_{i=1\ldots m} \phi_i \rightarrow \bigvee_{i=1\ldots n} \psi_i$$

where an empty antecedent (or succedent) is the neutral element of the conjunction ($true$), respectively, disjunction ($false$).

We use capital Greek letters to denote (possibly empty) sets of formulas. For instance, by $\Gamma \vdash \phi \rightarrow \psi, \Delta$, we mean a sequent containing at least an implication formula on the right side. Sequent calculus rules always have one sequent as conclusion and finitely many sequents as premises:

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

Semantically, a rule states that the validity of all n premises implies the validity of the conclusion ("top-down").

We use the sequent calculus analytically, i.e., we start with the sequent to be proven valid. A sequent calculus proof is a tree in which each node is labeled with a sequent. The root

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

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

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

Fig. 1 Some first-order rules

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

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

$$\text{assignToUpd}\ \frac{\lfloor \{\mathcal{U}\} \phi \rfloor}{\lfloor \langle\!\langle \mathcal{U} \rangle\!\rangle \phi \rfloor}\ \text{where } \mathcal{U} \text{ is any update} \qquad \text{Invariance}\ \frac{\lfloor \phi_1 \to \langle\!\langle s \rangle\!\rangle \phi_1' \rfloor}{\lfloor (\phi_1 \wedge \phi_2) \to \langle\!\langle s \rangle\!\rangle (\phi_1' \wedge \phi_2) \rfloor}\ \text{where } free(\phi_2) \cap change(s) = \emptyset \text{ and } \phi_2 \text{ contains no fields that are changed by } s$$

Fig. 2 Dynamic logic rules

node is labeled with the initial sequent to be proven valid. Rules are applied on the leaves of the proof tree in a bottom-up manner. This means that a rule is applied to the sequent seq_n of a leaf node n by matching its conclusion against seq_n, instantiating the premises with the obtained instantiation and adding them as new children of n. This reduces the provability of the conclusion to the provability of the premises step-wise until the obtained sequents are trivially valid (same formula on both sides, $false$ in the antecedent or $true$ in the succedent) and can be closed by applying rules with no premise. A proof is closed if and only if all its branches are closed.

In Fig. 1, we present some of the rules dealing with propositional connectives and quantifiers (see [30] 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. We write $\phi[e/l]$ to denote the standard substitution of l with e in ϕ.

Several sequent rules resemble more (conditional) rewrite rules, which replace a formula ϕ by an equivalent formula ϕ' or a term t by a semantically equal term t'. Most rules dealing with programs are actually of this kind as most of them are not sensitive to the side of the sequent and can moreover even be applied to sub-formulas. 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 it occurs. For ease of presentation, we introduce the following syntax

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

to express these kind of rules where the premise is constructed from the conclusion via replacing an occurrence of ϕ by ϕ'.

Example 1 A simple example for a rewrite rule is for instance

$$\text{concreteAnd}\ \frac{\lfloor \phi \rfloor}{\lfloor \phi \wedge true \rfloor}$$

which can be applied on all occurrences of $A \wedge true$ in the sequent

$$B \to (A \wedge true), A \wedge true \vdash B \wedge (A \wedge true)$$

reducing it step-wise to the sequent

$$B \to A, A \vdash B \wedge A$$

In Fig. 2, we present the rules dealing with statements. The schematic modality $\langle\!\langle \cdot \rangle\!\rangle$ can be instantiated with both $[\cdot]$ and $\langle \cdot \rangle$, though consistently within a single rule application.

Total correctness formulas of the form $\langle \text{while} \ldots \rangle \phi$ are proved by first applying the induction rule ind (possibly after generalizing the formula), followed by the unwind rule in the induction step. To deal with formulas of the form $[\text{while} \ldots]\phi$, we use the standard loop invariant rules, e.g., [12,13]. The invariance rule (Sect. 2) is an adaptation rule. This name originates from the fact that it allows adapting a contract to a specific context. For while programs, adaptation rules are unnecessary, but they are essential for a relative complete proof system with recursive procedures (see [7]).

Method calls In the specification of methods, we employ auxiliary variables, sometimes also called ghost variables. Such variables are used to specify and verify properties that would otherwise be difficult or even impossible to specify (see the case study Sect. 7 for an example of such a property). One aspect in which our approach differs from the seminal work of Owicki and Gries [48] is that due to object creation, their rule for deleting assignments to ghost variables does not hold in our approach. For instance, the formula $\{u := \text{new}\}(\exists x.x \neq null)$ might not be valid if the update is deleted.

From a pure logic point of view, ghost variables (or ghost fields) are nothing else than normal local variables (or fields) and their semantics are exactly the same. The syntactic distinction is made to distinguish cleanly between program code and specification. This allows us to strip the whole specification code out of the source code without altering its semantics; in particular, the production program code can be compiled such that it does not contain variable declarations and other statements which are used for specification purposes only.

To ease specification, we restrict the usage of ghost variables as follows: In our approach, ghost variables do not occur in method bodies and are used only to express properties of methods. The value of ghost variables will be set in a special block of code that executes directly after the

Fig. 3 Update application, general cases

$$\text{R1}\ \frac{\{\mathcal{U}\}\phi_1 * \{\mathcal{U}\}\phi_2 \rightsquigarrow \phi'}{\{\mathcal{U}\}(\phi_1 * \phi_2) \rightsquigarrow \phi'} \quad \text{with } * \in \{\wedge, \vee, \rightarrow\} \qquad \text{R2}\ \frac{\{\mathcal{U}\}\phi \rightsquigarrow \phi'}{\{\mathcal{U}\}(\neg\phi) \rightsquigarrow \neg\phi'} \qquad \text{R3}\ \frac{\{\mathcal{U}_{nc}\}\phi \rightsquigarrow \phi'}{\{\mathcal{U}_{nc}\}(Q\,l.\,\phi) \rightsquigarrow Q\,l.\,\phi'} \quad \text{with } Q \in \{\forall, \exists\},\ l \text{ not in } \mathcal{U}_{nc}$$

$$\text{R4}\ \frac{\{\mathcal{U}_{nc}\}e_1 = \{\mathcal{U}_{nc}\}e_2 \rightsquigarrow e'}{\{\mathcal{U}_{nc}\}(e_1 = e_2) \rightsquigarrow e'} \qquad \text{R5}\ \frac{f(\{\mathcal{U}\}e_1, ..., \{\mathcal{U}\}e_n) \rightsquigarrow e'}{\{\mathcal{U}\}f(e_1, ..., e_n) \rightsquigarrow e'} \qquad \text{R6}\ \frac{C(\{\mathcal{U}_{nc}\}e) \rightsquigarrow e'}{\{\mathcal{U}_{nc}\}C(e) \rightsquigarrow e'} \quad \text{where } C \text{ is a class predicate}$$

$$\text{R7}\ \frac{\begin{array}{l}(\mathsf{if}(\{\mathcal{U}\}b)\ \mathsf{then}(\{\mathcal{U}\}\phi_1)\\ \mathsf{else}(\{\mathcal{U}\}\phi_2)\ \mathsf{fi}) \rightsquigarrow e'\end{array}}{\{\mathcal{U}\}(\mathsf{if}\ b\ \mathsf{then}\ \phi_1\ \mathsf{else}\ \phi_2\ \mathsf{fi}) \rightsquigarrow e'} \qquad \text{R14}\ \frac{\begin{array}{l}\{\mathcal{U}\}(\mathsf{if}\ b\ \mathsf{then}\ op(e_1, \ldots, e_i, \ldots, e_n)\\ \mathsf{else}\ op(e_1, \ldots, e_i', \ldots, e_n)\ \mathsf{fi}) \rightsquigarrow e'\end{array}}{\{\mathcal{U}\}op(e_1, \ldots, \mathsf{if}\ b\ \mathsf{then}\ e_i\ \mathsf{else}\ e_i'\ \mathsf{fi}, \ldots, e_n) \rightsquigarrow e'} \quad \text{where } op \text{ is any function or relation symbol}$$

$$\text{R15}\ \frac{\{\mathcal{U}\}(\mathsf{if}\ b\ \mathsf{then}\ e\ \mathsf{else}(\mathsf{if\ false\ then}\ e\ \mathsf{fi})\ \mathsf{fi}) \rightsquigarrow e'}{\{\mathcal{U}\}(\mathsf{if}\ b\ \mathsf{then}\ e\ \mathsf{fi}) \rightsquigarrow e'} \qquad \text{R16}\ \frac{\mathsf{if\ false\ then}\ e\ \mathsf{fi} \rightsquigarrow e'}{\{\mathcal{U}\}(\mathsf{if\ false\ then}\ e\ \mathsf{fi}) \rightsquigarrow e'}$$

where op is any function or relation symbol and $b \not\equiv \mathsf{false}$

method body. This code block is required to terminate for obvious reasons. In the block, fields and normal variables are *read-only* and only ghost variables may be assigned to. To avoid complicated analyses of the control flow, we disallow method calls in the code block: The block is a while program.

We associate two dynamic logic formulas to each method: a precondition and a post-condition. Following the design by contract paradigm, the partial correctness *contract* of a method m with $u_1, \ldots, u_n$ as its formal parameters, a precondition p and a post-condition q is given by the formula $p \rightarrow [m(t_1, \ldots, t_n)]\langle s\rangle q$. Intuitively, the formula states that when precondition p is satisfied then the post-condition q must hold after execution of method m (or m must diverge). In more detail, in the contract we require $free(q) \cap \{u_1, \ldots, u_n\} = \emptyset$. The while program s is the code block for ghost variables (note that the modality ensures termination of this statement), and $t_1, \ldots t_n$ is a sequence of fresh variables representing the actual parameters (which can be used in the post-condition to refer to the actual parameters' value). The freshness of the variables ensures that the contract specifies a property of a *generic* call. The restriction that the formal parameters do not occur in the post-condition is necessary for the soundness of the verification conditions we are about to introduce. To prove for example that the value of the formal parameters before and after the call is the same (they are reset to their initial values according to the semantics of method calls), other rules such as the invariance rule must be used.

Proving correctness of a program where the main statement has the contract $p \rightarrow [s_{\text{main}}]q$ involving (possibly recursive) methods with contracts of the form $p_i \rightarrow [m_i(t_1, \ldots, t_n)]\langle s_i\rangle q_i$ and method body B_i amounts to proving the following *verification conditions*:

1. $\vdash p \rightarrow [s_{\text{main}}]q$
2. $\vdash (p_i \wedge u_1 = t_1 \wedge \ldots \wedge u_n = t_n) \rightarrow [B_i]\langle s_i\rangle q_i$

These verification conditions are inspired by the (partial correctness) rules for Hoare logic given in [8].

5 Applications of updates

In this section, we define a rewrite relation on dynamic logic formulas *with* updates, to standard first-order logic formulas *without* updates. This relation is necessary to reason about formulas containing updates. Updates are essentially delayed substitutions.[1] 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 Sect. 6.

We now define update application on formulas in terms of a rewrite relation $\{\mathcal{U}\}\phi \rightsquigarrow \phi'$ on formulas. It is here where we also need, as a technical vehicle, the update applied to expressions, $\{\mathcal{U}\}e$ (see the syntax of expressions, Sect. 2).

5.1 General updates

Figure 3 defines $\rightsquigarrow$ for general updates. The symbol $\mathcal{U}$ matches all updates, whereas $\mathcal{U}_{nc}$ ('*n*on-creating') excludes the form $u := \mathsf{new}$. Most rules are standard (see also [11,49]); the non-standard rules R14, R15 and R16 are discussed separately below. 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.

Object creation of the form $u := \mathsf{new}$ is only covered in so far as it behaves like any other update. The cases where object creation differs are discussed separately in Sect. 5.2. The relation $\rightsquigarrow$ is defined in a big-step manner, such that updates are resolved completely in a single $\rightsquigarrow$ step.

[1] The benefit of delaying substitutions in the context of symbolic execution is illustrated in Sect. 6.

Fig. 4 Update application, special cases

$$\text{R8}\ \frac{}{\{u := e\}u \rightsquigarrow e}$$

$$\text{R9}\ \frac{}{\{\mathcal{U}\}u \rightsquigarrow u}\ \text{where } \mathcal{U} \equiv u' := e \text{ or } \mathcal{U} \equiv e.x := e_1 \text{ or } \mathcal{U} \equiv e_1[e_2] := e_3 \text{ or } \mathcal{U} \equiv u' := \mathsf{new}$$

$$\text{R10}\ \frac{(\{\mathcal{U}\}e_2).x \rightsquigarrow e'}{\{\mathcal{U}\}(e_2.x) \rightsquigarrow e'}\ \text{where } \mathcal{U} \equiv u := e_1 \text{ or } \mathcal{U} \equiv e.y := e_1 \text{ or } \mathcal{U} \equiv e_1[e_2] := e_3$$

$$\text{R11}\ \frac{\mathsf{if}(e = \{\mathcal{U}\}e_2)\ \mathsf{then}\ e_1\ \mathsf{else}(\{\mathcal{U}\}e_2).x\ \mathsf{fi} \rightsquigarrow e'}{\{\mathcal{U}\}(e_2.x) \rightsquigarrow e'}\ \text{where } \mathcal{U} \equiv e.x := e_1$$

$$\text{R12}\ \frac{(\{\mathcal{U}\}e_4)[\{\mathcal{U}\}e_5] \rightsquigarrow e'}{\{\mathcal{U}\}(e_4[e_5]) \rightsquigarrow e'}\ \text{where } \mathcal{U} \equiv u := e \text{ or } \mathcal{U} \equiv e.x := e_1 \text{ or } \mathcal{U} \equiv e_1[e_2] := e_3 \text{ and } e_1, e_4 \text{ have a different type}$$

$$\text{R13}\ \frac{\mathsf{if}(e_1 = \{\mathcal{U}\}e_4 \wedge e_2 = \{\mathcal{U}\}e_5)\ \mathsf{then}\ e_3\ \mathsf{else}(\{\mathcal{U}\}e_4)[\{\mathcal{U}\}e_5]\ \mathsf{fi} \rightsquigarrow e'}{\{\mathcal{U}\}(e_4[e_5]) \rightsquigarrow e'}\ \text{where } \mathcal{U} \equiv e_1[e_2] := e_3 \text{ and } e_1, e_4 \text{ have the same type}$$

For rule R14, observe that the axioms for models (Sect. 2.1.3) imply that conditional expressions satisfy the following identity:

$$op(e_1, \ldots, \mathsf{if}\, b\, \mathsf{then}\, e_i\, \mathsf{else}\, e_i'\, \mathsf{fi}, \ldots, e_n) = \mathsf{if}\, b\, \mathsf{then}\, op(e_1, \ldots, e_i, \ldots, e_n)\, \mathsf{else}\, op(e_1, \ldots, e_i', \ldots, e_n)\, \mathsf{fi}$$

where op is any function symbol or relation symbol in our language (including dereferencing a field and taking subscripts of an array). Rule R14 formalizes this outward shifting of the conditional expression. The shifting continues until the conditionals reach the formula level, at which rule R7 is applicable.

Such "shifting" is not sound for $\mathsf{if}\, b\, \mathsf{then}\, e\, \mathsf{fi}$-expressions. For example, if $proj$ is a binary function defined by $proj(x, y) = x$, then $proj(x, \mathsf{if\ false\ then}\, e\, \mathsf{fi}) = x$, but $\mathsf{if\ false\ then}\, proj(x, e)\, \mathsf{fi} = \mathsf{if\ false\ then}\, x\, \mathsf{fi}$, which denotes an arbitrary object (of the same type as x). Thus, clearly $proj(x, \mathsf{if\ false\ then}\, e\, \mathsf{fi}) = \mathsf{if\ false\ then}\, proj(x, e)\, \mathsf{fi}$ is not valid. However, from the axioms for models, we *can* infer the following identity:

$$\mathsf{if}\, b\, \mathsf{then}\, e\, \mathsf{fi} = \mathsf{if}\, b\, \mathsf{then}\, e\, \mathsf{else}\, (\mathsf{if\ false\ then}\, e\, \mathsf{fi})\, \mathsf{fi}$$

Hence, as a first step to treating updates on $\mathsf{if}\, b\, \mathsf{then}\, e\, \mathsf{fi}$-expressions, we introduce the rule R15. The resulting $\mathsf{if} - \mathsf{then} - \mathsf{else}$-expression can then be shifted outward using the previously discussed rule R14. Updates applied to an expression $\mathsf{if\ false\ then}\, e\, \mathsf{fi}$ can simply be discarded due to the following two reasons. First, updates only affect the interpretation of the variables or fields, but not the interpretation of $\mathsf{if} - \mathsf{then} - \mathsf{fi}$ (see Sect. 2.1.3). Second, if the condition is false, applying the update to e is not needed since the value of the expression then does not depend on the second argument e (again, see Sect. 2.1.3): $\mathsf{if\ false\ then}\, e\, \mathsf{fi}_T = \mathsf{if\ false\ then}\, e'\, \mathsf{fi}_T$. This justifies rule R16.

$$\text{R13/R12}\ \frac{\text{R9}\ \dfrac{}{\{\mathcal{U}\}u \rightsquigarrow u} \qquad \text{R9}\ \dfrac{}{\{\mathcal{U}\}i \rightsquigarrow i}}{\{\mathcal{U}\}(u[i]) \rightsquigarrow \mathsf{if}(s.a = u \wedge 1 = i)\ \mathsf{then}\ v\ \mathsf{else}\ u[i]\ \mathsf{fi}} \qquad \text{R9}\ \frac{}{\{\mathcal{U}\}j \rightsquigarrow j}$$
$$\{\mathcal{U}\}(u[i][j]) \rightsquigarrow \mathsf{if}(s.a = u \wedge 1 = i)\ \mathsf{then}\ v\ \mathsf{else}\ u[i]\ \mathsf{fi}[j]$$

Fig. 5 Illustration of the array rules. a and u are two-dimensional integer arrays, v a one-dimensional integer array, i, j are integers and $\mathcal{U} \equiv s.a[1] := v$

Figure 4 shows the update application rules for special cases. Rule R8 is the simple case where an update is applied to a term which is syntactically equal to the update's left-hand side. Rule R9 eliminates an update application in cases where it is syntactically decidable that the update has no effect. Rule R10 propagates the update over a field access expression if the left-hand side of the update is not identical to the accessed field. The conditional expressions used in the rules R11 and R13 take into account possible *aliases*. For example, in R11, we have to check whether the object denoted by e_2 *after* the update $\{\mathcal{U}\}$ equals that of e (*before* the update), because in that case its values obviously also change by the update.

Figure 5 contains an example illustrating the use of the rules. Note that in the derivation, both R12 and R13 are applied to subscripted expressions, but that R12 is applied in case $s.a$ and $u[i]$ have a different type, and R13 is applied if $s.a$ and u have the same array type, and consequently may be the same array.

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

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

The soundness of the rewrite relation for non-creating updates is stated in the next substitution lemma:

Lemma 1 (Substitution lemma)
Suppose $\langle \mathcal{U}_{nc}, M\rangle \to^* M'$.

1. *For all expressions e: if* $\{\mathcal{U}_{nc}\}e \rightsquigarrow e'$ *then* $[\![e']\!](M) = [\![e]\!](M')$.
2. *For all formulas* ϕ*: if* $\{\mathcal{U}_{nc}\}\phi \rightsquigarrow \phi'$ *then* $M \models \phi'$ *iff* $M' \models \phi$.

Proof All cases are standard and can be found in [8], except for the subscripted assignment. Let $\mathcal{U} \equiv e_1[e_2] := e_3$ and suppose $\langle \mathcal{U}, M\rangle \to^* M'$.

$[\![e_4[e_5]]\!](M')$

$=$ (semantics of array access)

$M'([\,])([\![e_5]\!](M'))([\![e_4]\!](M'))$

$=$ (induction hypothesis)

$M'([\,])([\![\{\mathcal{U}\}e_5]\!](M))([\![\{\mathcal{U}\}e_4]\!](M))$

$=$ (definition of M', semantics of subscripted assignment)

$$\begin{cases} [\![e_3]\!](M) & \text{if } [\![e_1]\!](M) = [\![\{\mathcal{U}\}e_4]\!](M) \wedge \\ & \quad [\![e_2]\!](M) = [\![\{\mathcal{U}\}e_5]\!](M) \\ M([\,])([\![\{\mathcal{U}\}e_5]\!](M))([\![\{\mathcal{U}\}e_4]\!](M)) & \text{otherwise} \end{cases}$$

$=$ (semantics of array access)

$$\begin{cases} [\![e_3]\!](M) & \text{if } [\![e_1]\!](M) = [\![\{\mathcal{U}\}e_4]\!](M) \wedge \\ & \quad [\![e_2]\!](M) = [\![\{\mathcal{U}\}e_5]\!](M) \\ [\![(\{\mathcal{U}\}e_4)[\{\mathcal{U}\}e_5]]\!](M)) & \text{otherwise} \end{cases}$$

$=$ (semantics of formulas)

$$\begin{cases} [\![e_3]\!](M) & \text{if } [\![e_1 = \{\mathcal{U}\}e_4 \wedge e_2 = \{\mathcal{U}\}e_5]\!](M) \\ [\![(\{\mathcal{U}\}e_4)[\{\mathcal{U}\}e_5]]\!](M)) & \text{otherwise} \end{cases}$$

$=$ (semantics of conditional expression)

$[\![\,\mathsf{if}\,(e_1 = \{\mathcal{U}\}e_4 \wedge e_2 = \{\mathcal{U}\}e_5)\,\mathsf{then}\,e_3\,\mathsf{else}\,(\{\mathcal{U}\}e_5)[\{\mathcal{U}\}e_4]\,\mathsf{fi}\,]\!](M) =$ (definition of update application on array expressions) $[\![\{\mathcal{U}\}(e_4[e_5])]\!](M)$

□

5.2 Contextual application of object creation

To define update application on $\{u := \mathsf{new}\}e$, simple substitution, i.e., replacing u in e by some expression, is not sufficient because we cannot refer to the newly created object in the model prior to its creation. However, object expressions can only be compared for equality, dereferenced, accessed as an array if the object is of an array type or appeared as arguments of a class predicate or conditional expression. Since object expressions do not appear as arguments of any other function, we define update application by a contextual analysis of the occurrences of u in e. Some cases are already covered in Sect. 5.1 and Fig. 3 (the rules dealing with unrestricted $\mathcal{U}$). The other cases are discussed below.

Since the fields of a new object are initialized to their default value, we have the following rules

$$\mathsf{C1}\ \frac{}{\{u := \mathsf{new}\}(u.x) \rightsquigarrow init_T}$$
where C $\to T$ is the type of x

$$\mathsf{C2}\ \frac{(\{u := \mathsf{new}\}e).x \rightsquigarrow e'}{\{u := \mathsf{new}\}(e.x) \rightsquigarrow e'}$$
where e is neither u nor a conditional expression

The next cases states that all elements in a newly created array are initialized to their default value:

$$\mathsf{C3}\ \frac{}{\{u := \mathsf{new}\}u[e] \rightsquigarrow init_T}$$
where $T[\,]$ is the type of u

$$\mathsf{C4}\ \frac{(\{u := \mathsf{new}\}e)[\{u := \mathsf{new}\}(e_1)]) \rightsquigarrow e'}{\{u := \mathsf{new}\}(e[e_1]) \rightsquigarrow e'}$$
where e is neither u nor a conditional expression

Another possible context in which u can occur 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[2]

$$\mathsf{C5}\ \frac{(\{u := \mathsf{new}\}e) = (\{u := \mathsf{new}\}e') \rightsquigarrow e''}{\{u := \mathsf{new}\}(e = e') \rightsquigarrow e''}$$
where neither e nor e' is u or a conditional expression

If both the expressions e and e' are u, we obviously have

$$\mathsf{C6}\ \frac{}{\{u := \mathsf{new}\}(u = u) \rightsquigarrow \mathsf{true}}$$

On the other hand, 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); so we define

$$\mathsf{C7}\ \frac{}{\{u := \mathsf{new}\}(e = e') \rightsquigarrow \mathsf{false}}$$
where e is u and e' is neither u nor a conditional expression (or vice versa)

The final context in which u can occur is that of a class predicate, where the following three rules apply:

$$\mathsf{C8}\ \frac{}{\{u := \mathsf{new}\}(\mathrm{C}(u)) \rightsquigarrow \mathsf{true}}$$
where u is of type C

[2] To see why the shifting inward of $\{u := \mathsf{new}\}$ is necessary, consider the case $\{u := \mathsf{new}\}(u.x = 0)$. Neither side of the equality denotes the new object (at the top level), but the new object occurs in a subexpression (namely $u.x$). With shifting, $\{u := \mathsf{new}\}(u.x = 0)$ results in $(0 = 0)$. Without shifting, the (incorrect) result is $u.x = 0$.

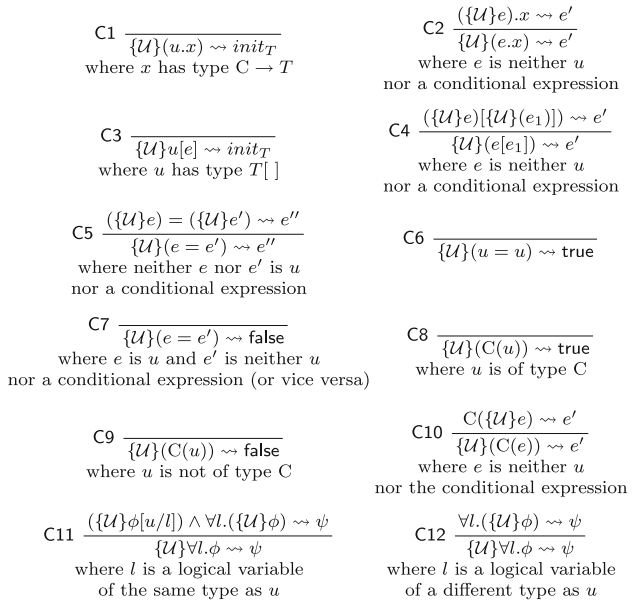

Fig. 6 Simplification of object creation with $\mathcal{U} \equiv u := \mathsf{new}$

$$\mathsf{C9}\ \frac{}{\{u := \mathsf{new}\}(\mathrm{C}(u)) \rightsquigarrow \mathsf{false}}$$
where u is not of type C

$$\mathsf{C10}\ \frac{\mathrm{C}(\{u := \mathsf{new}\}e) \rightsquigarrow e'}{\{u := \mathsf{new}\}(\mathrm{C}(e)) \rightsquigarrow e'}$$
where e is neither u nor the conditional expression

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 universal quantification are:

$$\mathsf{C11}\ \frac{(\{u := \mathsf{new}\}\phi[u/l]) \wedge \forall l.(\{u := \mathsf{new}\}\phi) \rightsquigarrow \psi}{\{u := \mathsf{new}\}\forall l.\phi \rightsquigarrow \psi}$$
where l is a logical variable of the same type as u

$$\mathsf{C12}\ \frac{\forall l.(\{u := \mathsf{new}\}\phi) \rightsquigarrow \psi}{\{u := \mathsf{new}\}\forall l.\phi \rightsquigarrow \psi}$$
where l is a logical variable of a different type as u

The first rewrite rule takes care of the *changing scope* of the quantified variable l by distinguishing two cases: ϕ holds for the new object is expressed by the first conjunct $\{u := \mathsf{new}\}\phi[u/l]$ (which is obtained by application of the update to $\phi[u/l]$) and ϕ holds for all "old" objects is expressed by the second conjunct $\forall l.(\{u := \mathsf{new}\}\phi)$. The rules for existential quantification can be derived from the rules for universal quantification and the equivalence $\exists l.\phi$ iff $\neg\forall l.\neg\phi$. For easy reference, the object creation rules are summarized in Fig. 6.

Abstract rewrite systems in general can obey two properties: termination and confluence. Together these two properties ensure the existence of a unique normal form. In our case, the normal form is an expression or formula without updates and uniqueness of the normal form is clearly only important up to logical equivalence (i.e., a sort of "semantic version" of confluence). The next lemma characterizes the expressions on which our rewrite relation terminates[3]

Lemma 2 (Termination of the rewrite relation)
Let ϕ' be any pure first-order formula (no modal operators) and let e' be any expression not generated by the grammar

$$e_{nnf} ::= u \mid \mathsf{if}\ b\ \mathsf{then}\ e_{nnf}\ \mathsf{fi} \mid \mathsf{if}\ b\ \mathsf{then}\ e_{nnf}\ \mathsf{else}\ e\ \mathsf{fi} \mid \mathsf{if}\ b\ \mathsf{then}\ e\ \mathsf{else}\ e_{nnf}\ \mathsf{fi}$$

Then

1. $\{u := \mathsf{new}\}e' \rightsquigarrow e$ *where e contains no updates.*
2. $\{u := \mathsf{new}\}\phi' \rightsquigarrow \phi$ *where ϕ contains no updates.*

Proof Sketch For formulas ϕ with quantifiers, it suffices to note that in the derivation of $\{u := \mathsf{new}\}\phi \rightsquigarrow \phi'$, the quantifier rule applies in its premises $\{u := \mathsf{new}\}$ to formulas with one less quantifier. This proves the second case, by induction on the number of quantifiers. For the first case, note that there is a rewrite rule for each expression $e' \not\equiv u$. In all rules without premises, the resulting expression contains no updates. The lemma now follows from the observation that for rules with premises, but not involving the offending forms of conditional expressions excluded by the grammar above, the update is applied to expressions different from u, whose parse trees are of a lesser height. □

The following theorem extends Lemma 1 to object creations. It guarantees that the normal forms obtained from applying the rewrite relation are unique up to logical equivalence. Intuitively, the second case of the theorem together with the second case of the previous lemma states that we can compute weakest preconditions of abstract object creation for any pure first-order formula (i.e., not involving modalities).

Theorem 2 Semantic correctness of the rewrite relation
Suppose $\langle u := \mathsf{new}, M\rangle \rightarrow^* M'$.

1. *For any expression e, if* $\{u := \mathsf{new}\}e \rightsquigarrow e'$ *then* $[\![e']\!](M) = [\![e]\!](M')$.
2. *For any formula ϕ, if* $\{u := \mathsf{new}\}\phi \rightsquigarrow \phi'$ *then* $M \models \phi'$ *iff* $M' \models \phi$.

Proof Suppose $\mathcal{U} \equiv u := \mathsf{new}$, $\langle \mathcal{U}, M\rangle \rightarrow^* M'$, and o is the newly created object.
We first prove the case for $e \equiv \mathsf{if\ false\ then}\ e\ \mathsf{fi}$:

$[\![\{\mathcal{U}\}(\mathsf{if\ false\ then}\ e\ \mathsf{fi})]\!](M)$
$=$ (rule R16)
$[\![\mathsf{if\ false\ then}\ e\ \mathsf{fi}]\!](M)$

[3] As a counterexample, the term $\{u := \mathsf{new}\}u$ cannot be simplified further.

$=$ (Operational semantics of object creation (pg. 13) and last axiom for models)

$[\![\text{ if false then } e \text{ fi }]\!](M')$

The other cases are proven by induction on the structure of expressions and formulas. We illustrate two representative cases, namely when the rewrite relation for object creation is applied to an expression $e'.x$ and when it is applied to a formula $\forall l : \phi$. Other cases follow analogously to these.

– If $e \equiv e'.x$, where x is a field of type $\mathrm{C} \to T$. We distinguish three sub-cases.

1. $e \equiv u.x$:

$[\![\{\mathcal{U}\}(u.x)]\!](M)$
$=$ (rule C1 and value of $init_T$ in the model M)
$init_T$
$=$ (Since fields of new objects are initialized their default value)
$M(x)(o)$
$=$ (Operational semantics of object creation; pg. 13)
$M(x)([\![u]\!](M'))$
$=$ (Semantics of field access)
$[\![u.x]\!](M')$

2. $e \equiv (\text{ if } b \text{ then } e_1 \text{ else } e_2 \text{ fi }).x$:

$[\![\{\mathcal{U}\}((\text{ if } b \text{ then } e_1 \text{ else } e_2 \text{ fi }).x)]\!](M)$
$=$ (rule R14)
$[\![(\{\mathcal{U}\} \text{ if } b \text{ then } e_1.x \text{ else } e_2.x \text{ fi }]\!](M)$
$=$ (rule R7)
$[\![\text{ if } (\{\mathcal{U}\}b) \text{ then } (\{\mathcal{U}\}(e_1.x)) \text{ else } (\{\mathcal{U}\}(e_2.x)) \text{ fi }]\!](M)$
$=$ (Semantics of conditional expression)
$\begin{cases} [\![\{\mathcal{U}\}(e_1.x)]\!](M) & \text{if } [\![\{\mathcal{U}\}b]\!](M) \\ [\![\{\mathcal{U}\}(e_2.x)]\!](M) & \text{otherwise} \end{cases}$
$=$ (Induction hypothesis)
$\begin{cases} [\![e_1.x]\!](M') & \text{if } [\![b]\!](M') \\ [\![e_2.x]\!](M') & \text{otherwise} \end{cases}$
$=$ (Semantics of conditional expression)
$[\![\text{ if } b \text{ then } e_1.x \text{ else } e_2.x \text{ fi }]\!](M')$

3. $e \equiv \text{ if } b \text{ then } e \text{ fi }.x$ where $b \not\equiv \text{ false }$:

$[\![\{\mathcal{U}\}(\text{ if } b \text{ then } e \text{ fi }.x)]\!](M)$
$=$ (rule R15)
$[\![\{\mathcal{U}\}(\text{ if } b \text{ then } e \text{ else if false then } e \text{ fi fi }.x)]\!](M)$
$=$ (rule R14)
$[\![\{\mathcal{U}\}(\text{ if } b \text{ then } e.x \text{ else } (\text{ if false then } e \text{ fi }).x \text{ fi })]\!](M)$
$=$ (rule R7)
$[\![(\text{ if } \{\mathcal{U}\}b \text{ then } \{\mathcal{U}\}(e.x) \text{ else }$
$\{\mathcal{U}\}((\text{ if false then } e \text{ fi }).x) \text{ fi })]\!](M)$
$=$ (Semantics of conditional expression)
$\begin{cases} [\![\{\mathcal{U}\}(e.x)]\!](M) & \text{if } [\![\{\mathcal{U}\}b]\!](M) \\ [\![\{\mathcal{U}\}((\text{ if false then } e \text{ fi }).x)]\!](M) & \text{otherwise} \end{cases}$
$=$ (Rule C2)
$\begin{cases} [\![\{\mathcal{U}\}(e.x)]\!](M) & \text{if } [\![\{\mathcal{U}\}b]\!](M) \\ [\![(\{\mathcal{U}\} \text{ if false then } e \text{ fi }).x]\!](M) & \text{otherwise} \end{cases}$
$=$ (See proof above about $\text{ if false then } e \text{ fi }$)
$\begin{cases} [\![\{\mathcal{U}\}(e.x)]\!](M) & \text{if } [\![\{\mathcal{U}\}b]\!](M) \\ [\![(\text{ if false then } e \text{ fi }).x]\!](M') & \text{otherwise} \end{cases}$
$=$ (Induction hypothesis)
$\begin{cases} [\![e.x]\!](M') & \text{if } [\![b]\!](M') \\ [\![(\text{ if false then } e \text{ fi }).x]\!](M') & \text{otherwise} \end{cases}$
$=$ (Semantics of if-then)
$[\![\text{ if } b \text{ then } e \text{ fi }.x]\!](M')$

4. $e \equiv e'.x$ and e' is neither u nor a conditional expression:

$[\![\{\mathcal{U}\}(e'.x)]\!](M)$
$=$ (rule C2)
$[\![(\{\mathcal{U}\}e').x]\!](M)$
$=$ (Semantics of field access)
$M(x)([\![\{\mathcal{U}\}e']\!](M))$
$=$ (Induction hypothesis)
$M'(x)([\![e']\!](M'))$
$=$ (Semantics of field access)
$[\![e'.x]\!](M')$

– If $\phi \equiv \forall l : \phi$ where l has the same (class) type as u then:

$M \models \{\mathcal{U}\}\forall l.\phi$
iff (rule C11, semantics of formulas)
$M \models \{\mathcal{U}\}\phi[u/l]$ and $M \models \forall l.\{\mathcal{U}\}\phi$
iff (induction hypothesis, definition of M')
$M' \models \phi[u/l]$ and $M \models \forall l.\{\mathcal{U}\}\phi$
iff (semantics of formulas)
$M' \models \phi[u/l]$ and $M[l := o'] \models \{\mathcal{U}\}\phi$ for all $o' \in M(\mathrm{C})$
iff (substitution lemma and $M'(u) = o$)
$M'[l := o] \models \phi$ and $M[l := o'] \models \{\mathcal{U}\}\phi$ for all $o' \in M(\mathrm{C})$
iff ($\langle \mathcal{U}, M[l := o'] \rangle \to^* M''$ implies $M'' = M'[l := o']$)
$M'[l := o] \models \phi$ and $M'[l := o'] \models \phi$ for all $o' \in M(\mathrm{C})$
iff (since $M'(\mathrm{C}) = M(\mathrm{C}) \cup \{o\}$)
$M'[l := o'] \models \phi$ for all $o' \in M'(\mathrm{C})$
iff (semantics of formulas)
$M' \models \forall l.\phi$

□

As an illustration of applying the rewrite relation, we derive $\{u := \text{ new }\}\forall l.\neg(u = l) \rightsquigarrow \neg(\text{ true }) \wedge \forall l.\neg \text{ false }$:

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

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

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

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.

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

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

6 Symbolic execution

6.1 Simultaneous updates for symbolic state representation

The proof system presented thus far allows for classical backward reasoning, in a weakest precondition manner. We now generalize 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, in Fig. 7.

The first application of the update rule mergeUpd introduces what is called the simultaneous update $w := u \mid u := v$. 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:

$$\mathsf{mergeUpd}\ \dfrac{\lfloor \{\mathcal{U}_1 \mid \cdots \mid \mathcal{U}_n \mid \mathcal{U}'\}\phi \rfloor}{\lfloor \{\mathcal{U}_1 \mid \cdots \mid \mathcal{U}_n\}\mathcal{U}\phi \rfloor}$$
$$\text{with } \{\mathcal{U}_1 \mid \cdots \mid \mathcal{U}_n\}\mathcal{U} \rightsquigarrow \mathcal{U}'$$

For this, we need to extend the rewrite relation $\rightsquigarrow$ toward 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 here is the definition of the application of simultaneous updates to *expressions*. For non-creating updates (like $\mathcal{U}_{nc}$ above), this is standard, so we do not include the full definition. (For a full account on simultaneous updates, see [49]. There, the standard cases are fully defined, in a small-step rewriting style, but without *creating* updates, instead using object activation.) Here, we only show one interesting special case, where two left-hand sides both write the field x which is accessed in $e.x$.

$$\dfrac{\mathsf{if}\ ((\mathcal{U}e) = e_2)\ \mathsf{then}\ e'_2\ \mathsf{else}\ \mathsf{if}\ ((\mathcal{U}e) = e_1)\ \mathsf{then}\ e'_1\ \mathsf{else}\ \mathcal{U}(e).x\ \mathsf{fi}\ \mathsf{fi} \rightsquigarrow e'}{\mathcal{U}(e.x) \rightsquigarrow e'}$$
$$\text{with } \mathcal{U} = \{e_1.x := e'_1 \mid e_2.x := e'_2\}$$

This already illustrates two principles: using updates allows us to delay the alias analysis to the actual application of an update to a field expression. Delaying the analysis allows us often to skip the analysis completely because of intermediate simplifications or because the analysis is—for the task at hand—not necessary at all (e.g., imagine one never has to evaluate an expression $e.x$). However, sometimes it is necessary to perform the alias analysis. Here, this means that we

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

Fig. 7 Symbolic execution style proof

have to evaluate e under the update $\mathcal{U}$ and to check whether it evaluates to same value as e_2 or e_1.

This brings us to the second principle, namely what happens in case of a clash, where e_1, e_2 and $\mathcal{U}(e)$ denote the same object. In our semantics, the rightmost update will "win." This exactly 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).

The idea to use simultaneous updates for symbolic execution was developed in the KeY project [11] 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 backward execution of (pure) weakest precondition calculi, thereby achieving better readability of proofs. The simultaneous update is only applied to the postcondition as a final, single step. The KeY tool uses these updates not only for verification, but also for test case generation with high code coverage [27] and for symbolic debugging.

6.2 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 $\{u := \mathsf{new}\}$ with other updates. Consider, for instance, the following formulas of which only the first is valid.

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

Symbolic execution generates the following formulas:

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

Merging the updates naively results in both cases in:

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

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:

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

The inner update $\{u := a\}$ 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 "neighbors," 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:

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

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

$$\begin{array}{rc}
 & A \\ \hline
\textsf{notRight, closeFalse} & \vdash \neg\,\mathsf{false} \\ \hline
\textsf{applyUpd} & \vdash \{a := \mathsf{new}\,\}\neg(v = a) \\ \hline
\textsf{applyUpd} & \vdash \{a := \mathsf{new}\,\}\{u := v \mid v := a \mid w := u\}\neg(w = v) \\ \hline
\textsf{mergeUpd} & \vdash \{a := \mathsf{new}\,\}\{u := v \mid v := a\}\{w := u\}\neg(w = v) \\ \hline
\textsf{mergeUpd, assignToUpd} & \vdash \{a := \mathsf{new}\,\}\{u := v\}\{v := a\}\langle w := u\rangle\neg(w = v) \\ \hline
\textsf{pullCreation} & \vdash \{u := v\}\{a := \mathsf{new}\,\}\{v := a\}\langle w := u\rangle\neg(w = v) \\ \hline
\textsf{split, createObj} & \vdash \{u := v\}\langle v := \mathsf{new}\,;\, w := u\rangle\neg(w = v) \\ \hline
\textsf{split, assignToUpd} & \vdash \langle u := v;\, v := \mathsf{new}\,;\, w := u\rangle\neg(w = v)
\end{array}$$

7 Case study and implementation

Consider a queue data structure, where items can be added to the beginning of the queue and removed from the end of the queue. The public interface of such a queue contains

- two global variables pointing to its $first$ and $last$ element (which are both null in an empty queue),
- an $enqueue(v)$ method which adds a Nat v to the beginning of the queue
- and a $dequeue$ method which removes the last item from the queue.

The queue is implemented as a linked list using a $next$ field which points to the next item in the queue. Figure 8 visualizes the result of the method call $first.enqueue(2)$, where $first$ initially (i.e., before executing the call) points to an item with value 3 and $last$ points to an item with value 25. We will verify the following reachability property of the $enqueue$ method, expressed in first-order logic using a ghost array:

$$
\begin{aligned}
&(first = last = \mathsf{null} \lor \exists a : \exists n : n \geq 0 \land a[0] \\
&\quad = first \land a[n] \\
&\quad = last \land \forall j (0 \leq j < n) : a[j].next = a[j+1]) \rightarrow \\
&[enqueue(v)]\langle s \rangle \\
&(\exists a : \exists n : n \geq 0 \land a[0] = first \land \\
&\quad a[n] = last \land \forall j (0 \leq j < n) : a[j].next = a[j+1])
\end{aligned}
$$

The statement s is the block on the right in Fig. 9 and contains updates to the ghost array variable. Intuitively, this property means that after executing $enqueue(v)$, $last$ is reachable from $first$ through repeated dereferencing of the $next$ field, provided this was the case initially. Note that by simply adding that $last.next = \mathsf{null}$ we can rule out cycles. For readability, we use the following abbreviations:

- $\text{reach}(f, l, a, n) \equiv n \geq 0 \land a[0] = f \land a[n] = l \land f \neq null \neq l \land l.next = null \land \forall j (0 \leq j < n) : a[j] \neq null \land a[j].next = a[j+1]$
- $p \equiv \text{reach}(first.next, last, a, n) \land first \neq null$
- $inv \equiv p \land 0 \leq i \land b \neq null \land b[0] = first \land \forall j (0 < j \leq i) : b[j] = a[j-1]$
- B: the method body of $enqueue$ (see the left code block in Fig. 9)
- s: updates to the auxiliary variables (right code block in Fig. 9)

Note that $\exists a : \exists n : \text{reach}(first, last, a, n)$ is the precondition and post-condition of the contract of $enqueue$, and p is an intermediate assertion, which holds directly after the body of $enqueue$ and before the updates to the ghost array variable. A symbolic execution style proof of this fact is shown in Fig. 10. The loop invariant of the updates to the ghost array variable in the right code block is abbreviated to inv.

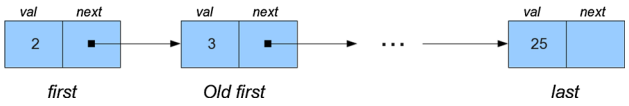

Fig. 8 Queue resulting from $first.enqueue(2)$

```
{reach(first, last, a, n)}
z := new;
z.next := first;
z.val := v;
first := z
{p}
```

```
{p}
b := new;
b[0] := first;
i := 0;
while b[i] ≠ last do
    {invariant: inv, bound: n + 1 − i}
    b[i+1] := b[i].next;
    i := i+1
od;
{reach(first, last, b, n + 1)}
```

Fig. 9 Proof outlines for the $enqueue(v)$ method body and ghost variable updates

$\langle z := new; z.next := first; z.val := v; first := z \rangle p$
(Rule createObj)
$\{b := new\}\{z := b\}\langle z.next := first; z.val := v; first := z \rangle p$
(Rule assignToUpd)
$\{b := new\}\{z := b\}\{z.next := first\}\langle z.val := v; first := z \rangle p$
(Rule mergeUpd)
$\{b := new\}\{z := b \mid b.next := first\}\langle z.val := v; first := z \rangle p$
(Rule assignToUpd)
$\{b := new\}\{z := b \mid b.next := first\}\{z.val := v\}\langle first := z \rangle p$
(Rule mergeUpd)
$\{b := new\}\{z := b \mid b.next := first \mid b.val := v\}\langle first := z \rangle p$
(Rule assignToUpd)
$\{b := new\}\{z := b \mid b.next := first \mid b.val := v\}\{first := z\}p$
(Rule mergeUpd)
$\{b := new\}\{z := b \mid b.next := first \mid b.val := v \mid first := b\}p$
$\rightsquigarrow$ (Definition of p and rules for update application in Figure 3, 4 and 6)
$\text{reach}(first, last, a, n) \land b \neq null$

Fig. 10 Symbolic execution proof of the left code block in Fig. 9

The proof outline on the right in Fig. 9 shows $p \rightarrow \langle s \rangle \text{reach}(first, last, b, n)$. This means after symbolic execution of the code (together with an application of the loop invariant rule), we end up with the following proof situation

$$
\begin{aligned}
&\text{reach}(first, last, b, n+1) \vdash \\
&\quad \exists a : \exists n : \text{reach}(first, last, a, n)
\end{aligned}
$$

which can be easily shown using our sequent calculus by instantiating both existential quantifiers to match the arguments of the reach predicate in the antecedent.

7.1 Implementation

The described dynamic logic for abstract object creation has been implemented based on the KeY verification system for Java [11]. In particular, we implemented all dynamic logic rules and update simplification rules as described in Sects. 4 and 5.

A notable feature of the implementation is that actual Java programs are supported, not just the core Java-like language introduced in this paper to focus on the relevant issues. In fact, due to building upon KeY, we inherit support for a considerably larger subset of Java than discussed in the paper. Besides

Table 1 Proof statistics

	Rule apps.	Interactions
AOC	4,366	1
Activation	5,959	28

inheritance, dynamic method binding and bounded arrays, which are discussed in detail in Sect. 3, we support classes with constructors, static fields and static methods, assignments with expressions containing side effects, etc. The supported subset is basically equal to sequential Java 1.4 without floating point arithmetic and garbage collection. Support for Java 5 features is preliminary. Enhanced for loops and variable arguments methods are fully supported; generics on the other side need to be transformed away using a provided but external tool.

None of these features required changes in the rules presented in this paper, which strengthens our belief that first, abstract object creation as introduced in Sect. 5 allows an orthogonal treatment of other features of Java, and second, that the base language in Sect. 2 is chosen appropriately. We could also reuse the proof search strategies from standard KeY with only minor modifications. This gives us instantaneous support for a highly automated proof search including advanced reasoning about linear and nonlinear arithmetic problems. To investigate the degree of automation, we mechanized the case study in both our own abstract object creation KeY variant (AOC) and the traditional activation style version. Table 1 summarizes the results of the comparison. The only required interaction in our AOC version was to provide a suitable loop invariant. Once the loop invariant was given, the correctness of the case study program could be proven fully automatically. The proof in the activation style KeY version required additional user interactions for suitable quantifier instantiations.

Standard KeY already supports the Java Modeling Language (JML) [17] as assertion language instead of dynamic logic. Assertions given in JML are first translated into dynamic logic proof obligations before being loaded into the prover. To achieve support for our KeY variant, we had to adopt the JML translation to make it aware of abstract object creation. By performing the required changes, we were able to support almost all of the JML features that are also supported by standard KeY and can use it as a convenient way to specify our programs.

8 Discussion

In this section, we discuss and compare the two main semantic approaches to object creation as well as the expressiveness of the core programming language and the assertion language.

8.1 Object creation versus object activation

Proof systems for object-oriented languages [1] 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 model, 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[13]. To avoid the extra quantifier, a 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" [12]. 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 object activation setting that a new object is indeed new, we need to prove the formula $\forall l.\,(l.\mathsf{created} \rightarrow \langle u := \mathsf{new}\rangle\neg(u = l))$. In fact, the corresponding formula $\forall l.\,\langle u := \mathsf{new}\rangle\neg(u = l)$ for abstract object creation is not valid in this setting. An object activation style proof of this is given in Fig. 11 (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.

8.2 Object destruction

We briefly illustrate the flexibility and generality of our approach by an indication of how to formalize in an abstract manner object destruction. To this end, consider the command destroy(u) which removes the object denoted by u from the domain of existing objects and sets u to null. In case u already denoted as null, the operation has no effect. In the calculation of the weakest precondition of destroy(u), setting u to null can be modeled in the standard way by the corresponding substitution. Formally, this leads to the base case:

$$\frac{}{\{\text{destroy}(u)\}u \rightsquigarrow \mathsf{null}}$$

The main problem concerns the dynamically changing scope of the quantifiers. This can be dealt with in an elegant manner

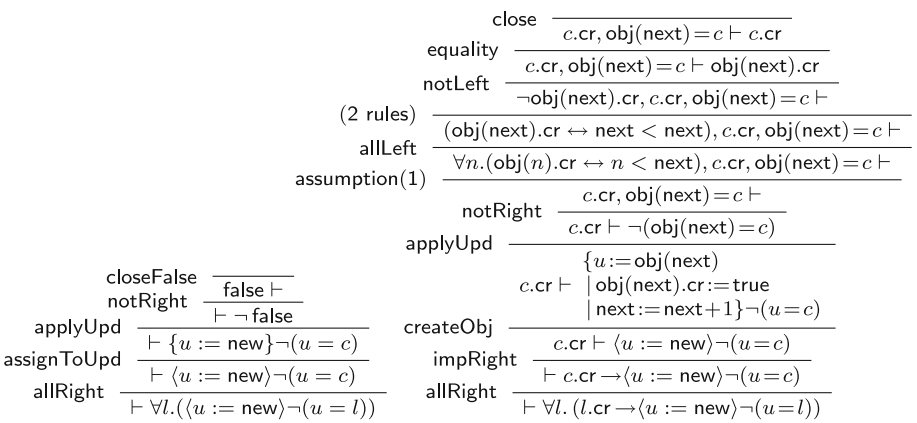

Fig. 11 Abstract object creation proof (*left*) versus activation (*right*)

by excluding the destroyed object from the range of quantification in the weakest precondition. More formally, this gives rise to the following rewrite rule:

$$\frac{\exists l \neq u.(\{\text{destroy}(u)\}\phi) \rightsquigarrow \psi}{\{\text{destroy}(u)\}\exists l.\phi \rightsquigarrow \psi}$$

8.3 Expressiveness

The programming language In Sect. 3, we show that our core language contains suitable primitive constructs from which more intricate features can be handled by a completely mechanical translation. As such, this justifies the core language as an intermediate target language for the verification of general object-oriented languages. The features to which this transformational approach applies include failures, *bounded* arrays, inheritance and dynamic binding. The transformational approach (see also [9]) in general clarifies the relation between programs of our core language and programs written in (for example) Java using such "derived" features. Moreover, the transformations allow us to treat object creation orthogonal to such features, thereby indicating that our approach scales up to modern languages.

The assertion language Standard first-order logic cannot express reachability properties. We have proposed first-order logic *together with auxiliary variables* to specify properties of the heap. In the case study, an example of a reachability property was expressed and proved subsequently. The question now arises how expressive our approach is in general. In the presence of general abstract data types, Tucker and Zucker [51] observe that for expressing, for example, strongest post-conditions standard arithmetic coding techniques do not apply. Therefore, Tucker and Zucker prove expressibility of strongest post-conditions in a weak second-order language that contains quantification over finite sequences. It is not difficult though tedious to show that using auxiliary array variables we can express in our first-order language strongest post-conditions for our programming language. In fact, in [23], we prove that the strongest post-condition of a formula in the language of Presburger arithmetic and a program instrumented with auxiliary variables in a suitable manner is definable in Presburger arithmetic itself. This is surprising, since the standard approach to show that the strongest post-condition is definable is based on the usual Gödel encoding of partial recursive functions, which relies on the presence of multiplication in the assertion language, and multiplication is not available in Presburger arithmetic. The basic idea is that one can instrument any program to store the computation in auxiliary array variables. The computation can then be recovered in an assertion by accessing these auxiliary variables. The above encoding of the strongest post-condition provides the basis for a standard relative completeness proof for instrumented programs as described in [31].

9 Conclusion

In this paper, we presented the state of the art in the KeY theorem prover. We showed how the assertion language used in KeY can be used conveniently together with auxiliary variables to provide a powerful way to express properties of the heap. Moreover, the assertion language supports *abstract* object creation (including dynamically created arrays), abstracting from irrelevant implementation details of object creation, which in general complicate proofs. We formalized a computation of the weakest precondition of abstract object creation in terms of a rewrite relation on formulas in first-order dynamic logic. We further showed how more complicated features can be handled by the transformational approach. Tool support is provided by a special version of KeY available on http://www.key-project.org/aoc/.

Future work A main line of future research concerns the integration of different techniques to further support modularity, i.e., local reasoning as supported by the separating conjunction of separation logic and dynamic frames.

References

1. Abadi, M., Leino, K.R.M.: A logic of object-oriented programs. In: Proceedings of 7th International Conference Theory and Practice of Software, volume 1214 of Lecture Notes in Computer Science, pp. 682–696. Springer, Berlin (1997)
2. Ábrahám, E., de Boer, F.S., de Roever, W.P., Steffen, M.: A deductive proof system for multithreaded Java with exceptions. Fundam. Inform. **82**(4), 391–463 (2008)
3. Ahrendt, W., de Boer, F.S., Grabe, I.: Abstract object creation in dynamic logic. In: FM, pp. 612–627 (2009)
4. Ahrendt, W., Mostowski, W., Paganelli, G.: Real-time Java API specifications for high coverage test generation. In: Proceedings of the 10th International Workshop on Java Technologies for Real-time and Embedded Systems. JTRES '12, pp. 145–154. ACM, New York, USA (2012)
5. America, P.: Designing an object-oriented programming language with behavioural subtyping. In: REX Workshop, pp. 60–90 (1990)
6. America, P.: Formal techniques for parallel object-oriented languages. In: CONCUR, pp. 1–17 (1991)
7. Apt, K.R.: Ten years of Hoare's logic: a survey—part 1. ACM Trans. Program. Lang. Syst. **3**(4), 431–483 (1981)
8. Apt, K.R., de Boer, F.S., Olderog, E.-R.: Verification of sequential and concurrent programs, 3rd edn. Texts in computer science. Springer, Berlin (2009). ISBN: 978-1-84882-744-8
9. Apt, K.R., de Boer, F.S., Olderog, E.-R., de Gouw, S.: Verification of object-oriented programs: a transformational approach. J. Comput. Syst. Sci. **78**(3), 823–852 (2012)
10. Barnett, M., DeLine, R., Fähndrich, M., Leino, K.R.M., Schulte, W.: Verification of object-oriented programs with invariants. J. Object Technol. **3**(6), 27–56 (2004)
11. Beckert, B., Hähnle, R., Schmitt, P.H. (eds.): Verification of Object-Oriented Software: The KeY Approach, volume 4334 of Lecture Notes in Computer Science. Springer, Berlin (2007)
12. Beckert, B., Klebanov, V., Schlager, S.: Dynamic logic. In: Beckert et al. [11], pp. 69–177
13. Beckert, B., Platzer, A.: Dynamic logic with non-rigid functions. In: Furbach, U., Shankar, N. (eds.) IJCAR, volume 4130 of Lecture Notes in Computer Science, pp. 266–280. Springer, Berlin (2006)
14. Berdine, J., Calcagno, C., O'Hearn, P.W.: Smallfoot: modular automatic assertion checking with separation logic. In: FMCO, pp. 115–137 (2005)
15. Berdine, J., Cook, B., Ishtiaq, S.: SLAyer: Memory safety for systems-level code. In: CAV, pp. 178–183 (2011)
16. Bormer, T., Brockschmidt, M., Distefano, D., Ernst, G., Filliâtre, J.-C., Grigore, R., Huisman, M., Klebanov, V., Marché, C., Monahan, R., Mostowski, W., Polikarpova, N., Scheben, C., Schellhorn, G., Tofan, B., Tschannen, J., Ulbrich, M.: The COST IC0701 verification competition 2011. In: Beckert, B., Damiani, F., Gurov, D. (eds.) FoVeOOS, volume 7421 of Lecture Notes in Computer Science, pp. 3–21. Springer, Berlin (2011)
17. Burdy, L., Cheon, Y., Cok, D.R., Ernst, M.D., Kiniry, J.R., Leavens, G.T., Leino, K.R.M., Poll, E.: An overview of JML tools and applications. STTT **7**(3), 212–232 (2005)
18. Burstall, R.M.: Program proving as hand simulation with a little induction. In: Information Processing '74, pp. 308–312. Elsevier/North-Holland, Amsterdam (1974)
19. Cadar, C., Dunbar, D., Engler, D.R.: KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs. In: Draves, R., van Renesse, R. (eds.) 8th USENIX Symposium on Operating Systems Design and Implementation, OSDI, San Diego, USA. USENIX Association (2008)
20. Calcagno, C., Yang, H., O'Hearn, P.W.: Computability and complexity results for a spatial assertion language for data structures. In: Hariharan, R., Vinay, V., Mukund, M. (eds.) FST TCS 2001: Foundations of Software Technology and Theoretical Computer Science. vol. 2245, pp. 108–119, Springer, Berlin, Heidelberg (2001)
21. Darvas, Á., Mehta, F., Rudich, A.: Efficient well-definedness checking. In: IJCAR, pp. 100–115 (2008)
22. de Boer, F.S.: A WP-Calculus for OO. In: Thomas, W. (eds.) FoSSaCS, volume 1578 of Lecture Notes in Computer Science, pp. 135–149. Springer, Berlin (1999)
23. de Gouw, S., de Boer, F.S., Ahrendt, W., Bubel, R.: Weak arithmetic completeness of object-oriented first-order assertion networks. In: SOFSEM, pp. 207–219 (2013)
24. De Halleux, J., Tillmann, N.: Parameterized unit testing with Pex. In: Proceedings of 2nd International Conference on Tests and Proofs, LNCS, pp. 171–181. Springer, Berlin (2008)
25. Dijkstra, E.W.: A Discipline of Programming. Prentice-Hall, Englewood Cliffs (1976)
26. Distefano, D., Parkinson, M.J.: jStar: Towards practical verification for Java. In: OOPSLA, pp. 213–226 (2008)
27. Engel, C., Hähnle, R.: Generating unit tests from formal proofs. In: Gurevich, Y., Meyer, B. (eds.) TAP, volume 4454 of Lecture Notes in Computer Science, pp. 169–188. Springer, Berlin (2007)
28. Filliâtre, J.-C., Marché, C.: The Why/Krakatoa/Caduceus platform for deductive program verification. In: CAV, pp. 173–177 (2007)
29. Galeotti, J., Rosner, N., Lopez Pombo, C., Frias, M.: TACO: efficient SAT-based bounded verification using symmetry breaking and tight bounds. IEEE Trans. Softw. Eng. **39**(9), 1283–1307 (2013)
30. Giese, M.: First-order logic. In: Beckert et al. [11], pp. 21–68
31. Harel, D.: Arithmetical completeness in logics of programs. In: ICALP, pp. 268–288 (1978)
32. Hoare, C.A.R.: An axiomatic basis for computer programming. Commun. ACM **12**(10), 576–580 (1969). 583
33. Huizing, K., Kuiper, R.: Verification of object-oriented programs using class invariants. Fundam. Approach. Softw. Eng. **1783**, 208–221 (2000)
34. Jacobs, B., Smans, J., Philippaerts, P., Vogels, F., Penninckx, W., Piessens, F.: VeriFast: a powerful, sound, predictable, fast verifier for C and Java. In: Proceedings of the Third international conference on NASA Formal Methods, NFM'11, pp. 41–55, Springer, Berlin, Heidelberg (2011)
35. Kassios, I.T., Müller, P., Schwerhoff, M.: Comparing verification condition generation with symbolic execution: an experience report. In: Proceedings of the 4th International Conference on Verified Software: Theories, Tools, Experiments, VSTTE'12, pp. 196–208, Springer, Berlin, Heidelberg (2012)
36. King, J.C.: Symbolic execution and program testing. Commun. ACM **19**(7), 385–394 (1976)
37. Klebanov, V., Müller, P., Shankar, N., Leavens, G.T., Wüstholz, V., Alkassar, E., Arthan, R., Bronish, D., Chapman, R., Cohen, E., Hillebrand, M., Jacobs, B., Leino, K.R.M., Monahan, R., Piessens, F., Polikarpova, N., Ridge, T., Smans, J., Tobies, S., Tuerk, T., Ulbrich, M., Weiß, B.: The 1st verified software competition: experience report. In: Proceedings of the 17th International Conference on Formal Methods. FM'11, pp. 154–168. Springer, Berlin, Heidelberg (2011)
38. Klein, G., Nipkow, T.: A machine-checked model for a Java-like language, virtual machine, and compiler. Trans. Program. Lang. Syst. **28**(4), 619–695 (2006)
39. Leavens, G.T., Kiniry, J.R., Poll, E.: A JML tutorial: Modular specification and verification of functional behavior for Java. In: CAV, p. 37 (2007)
40. Leavens, G.T., Leino, K.R.M., Poll, E., Ruby, C., Jacobs, B., Leavens, G.T., Ruby, C.: JML: Notations and tools supporting detailed design in Java. In: In OOPSLA 2000 Companion, pp. 105–106. ACM (2000)

41. Leino, K.R.M.: Dafny: an automatic program verifier for functional correctness. In: Clarke, E.M., Voronkov, A. (eds.) LPAR (Dakar), volume 6355 of Lecture Notes in Computer Science, pp. 348–370. Springer, Berlin (2010)
42. Leino, K.R.M., Müller, P., Smans, J.: Foundations of security analysis and design V. In: Chapter Verification of Concurrent Programs with Chalice, pp. 195–222. Springer, Berlin, Heidelberg (2009)
43. Leino, K.R.M., Müller, P., Smans, J.: Deadlock-free channels and locks. In: Proceedings of the 19th European Conference on Programming Languages and Systems. ESOP'10, pp. 407–426. Springer, Berlin, Heidelberg (2010)
44. Liskov, B., Wing, J.M.: A behavioral notion of subtyping. ACM Trans. Program. Lang. Syst. **16**(6), 1811–1841 (1994)
45. Mostowski, W.: Formalisation and verification of Java card security properties in dynamic logic. In: Cerioli, M. (ed.) Proceedings of Fundamental Approaches to Software Engineering (FASE), Edinburgh, volume 3442 of Lecture Notes in Computer Science, pp. 357–371. Springer, Berlin (2005)
46. Mostowski, W.: Fully verified Java Card API reference implementation. In: Beckert, B. (ed), VERIFY (2007)
47. Object Modeling Group: Object Constraint Language Specification, Version 2.0, June (2005)
48. Owicki, S.S., Gries, D.: An axiomatic proof technique for parallel programs I. Acta Inf. **6**, 319–340 (1976)
49. Rümmer, P.: Sequential, parallel, and quantified updates of first-order structures. In: Hermann, M., Voronkov, A, (eds.) LPAR, volume 4246 of Lecture Notes in Computer Science, pp. 422–436. Springer, Berlin (2006)
50. Smans, J., Jacobs, B., Piessens, F.: Implicit dynamic frames. ACM Trans. Program. Lang. Syst. **34**(1), 2:1–2:58 (2012). doi:10.1145/2160910.2160911
51. Tucker, J., Zucker, J.: Program Correctness over Abstract Data Types, With Error-State Semantics. Elsevier Science, Amsterdam (1988)
52. van den Berg, J., Jacobs, B.: The LOOP compiler for Java and JML. In: Margaria, T., Yi, W. (eds.) TACAS, volume 2031 of Lecture Notes in Computer Science, pp. 299–312. Springer, Berlin (2001)
53. Wittgenstein, L.: Tractatus logico-Philosophicus. London: Routledge, 1981 (1922)

Stijn de Gouw is a postdoc in a public (Centre for Mathematics and Computer Science)-private (SDL) partnership project focussing on checking and monitoring cloud applications. He previously obtained his Ph.D. in 2013 at Leiden University and the Centre for Mathematics and Computer Science in the context of the 7th Framework European HATS project, introducing a novel method that allows combining run-time assertion checking with monitoring. He graduated with a master degree in Computer Science in 2009 at Leiden University.

Frank de Boer graduated in 1985 at the University of Groningen (The Netherlands) in Philosophy. He obtained his Ph.D. degree in Computer Science (Free University, Amsterdam) in 1991. He is currently employed by the Centre for Mathematics and Computer Science (Amsterdam) as leader of the Formal Methods Research Group and as professor "program Correctness" by the Leiden Advanced Institute of Computer Science. His main interests are formal methods for concurrency.

Wolfgang Ahrendt is associate professor at Chalmers University of Technology in Gothenburg, Sweden. He graduated in 1995 at the University of Karlsruhe in Computer Science, and obtained his Ph.D. there in 2001. His main interests and contributions are in automated reasoning, logics and systems for the verification of (in particular object-oriented) software, and the verification of distributed systems. Wolfgang Ahrendt has served as secretary of the Association for Automated Reasoning and was member of the steering committee of CADE (Conference on Automated Deduction).

Richard Bubel He graduated in 2003 at the University of Karlsruhe (Germany) in Computer Science and obtained his Ph.D. there in 2007. He worked as a postdoc at the Chalmers University in Gothenburg. Currently he is employed as postdoc at the Computer Science Department of the Technical University of Darmstadt. He participated in the EU projects MOBIUS, HATS and Envisage. His main interests are deductive program verification, theorem proving and formal methods in software development.