Cite as follows: Bertrand Meyer and Alexander Kogtenkov, *Negative Variables and the Essence of Object-Oriented Programming*, in SAS 2014, Kanazawa (Japan), April 2014, eds S. Iida, J. Meseguer, and K. Ogata, Lecture Notes in Computer Science 8373, Springer, 2014, pages 171–187.

Negative Variables and the Essence of Object-Oriented Programming

Bertrand Meyer

ETH Zurich
& Eiffel Software (Santa Barbara)
& ITMO National Research University (Saint Petersburg)
Bertrand.Meyer@inf.ethz.ch

Alexander Kogtenkov

ITMO National Research University (Saint Petersburg)
& Eiffel Software (Moscow)

alexk@eiffel.com

Abstract. Reasoning about object-oriented programs requires an appropriate technique to reflect a fundamental "general relativity" property of the approach: every operation is relative to a current object, which changes with every qualified call; such a call needs access to the context of the client object. The notion of negative variable, discussed in this article, provides a framework for reasoning about OO programs in any semantic framework. We introduce a fundamental rule describing the semantics of object-oriented calls, its specific versions for such frameworks as axiomatic (Hoare-style) logic and denotational semantics, and its application to such problems as alias analysis and the consistency of concurrent programs. The approach has been implemented as part of a verification environment for a major object-oriented language and used to perform a number of proofs and analyses.

Keywords: Program logic, operational semantics, object-oriented language.

1 Preamble: the need for coordinate transform

The concept of negative variable, discussed in this article, addresses a specific but important aspect of reasoning about object-oriented programs: the need to obtain reverse access to the context of your caller. Current verification approaches miss it, and hence cannot express certain important properties, let alone verify them. Even for properties that can be expressed otherwise, the negative variable technique provides a simpler and more elegant framework, making automatic verification easier.

A little non-technical example illustrates the issue (all person names are fictitious). Eri likes to party, and has many followers who send her lots of invitations on Twitter, but she is selective. A typical tweet says "*Restaurant Komatsu Yasuke, today at 19:30, Shin also coming*". But she would like to know more: how many people are invited? Is Junko coming? (If so Eri will stop at home on the way, to pick up a nice bracelet that she has bought for her.) Now whoever is inviting Eri — today Kokichi, say, and tomorrow Taku — could answer these questions; but Eri's procedure for

accepting or skipping an invitation can only be based on the message she has received; she would need access to information available only to the tweet's author.

All she does know is the content of the tweet: place, time, and possibly the name of another person who is also invited. Maybe that person has the other information; but maybe not. The only way to answer the pending questions would be to reach the original tweeter.

This setup, including lack of access to the tweeter's own context, exactly mirrors what happens in the execution of a routine (method) on a target object in an object-oriented programming language. We are considering a "qualified call"

call *Eri*.*invite* (*Komatsu_Yasuke*, [*Today*, "19:30"], *Shin*) (1)

(using an explicit **call** keyword for clarity, although it usually does not appear in programming languages). This call executes the procedure *invite* on the "target object" denoted by *Eri*, with the arguments given. The procedure is declared with the corresponding arguments:

invite (*p*: *PLACE*; *d*: *DATE*; *other_invitee*: *PERSON*)
require … **do** … **ensure** … **end**

To do its work, the procedure can only use the arguments it has; but then it lacks context. For example it cannot answer Eri's question, which we can rephrase in software terms. The question applies to a given object such as the restaurant, accessible to the procedure as the formal argument *p*:

- Is *x* (some person) also invited to *p* today? (2)
- How many people are invited to *p* today? (3)

In a particular call, such as (1), this information is accessible to the calling object, but not to the object on which the call executes.

In the *writing* of object-oriented programs, this restriction is not a major obstacle (otherwise people would have been complaining about it loudly). In fact one can argue that not knowing the caller helps write self-contained, reusable code.

For *reasoning* about OO programs, however, the restriction also exists, and it hurts. For example Müller [13] states, in presenting a proof rule for OO routines:

> *Req-clauses* [shared precondition components] *and* [the rest of the] *preconditions may refer to formal* [arguments], *the object store, and the current universe, whereas the postcondition may only refer to the object store and* ***result*** [denoting the result of a function].

This information does not identify the caller, and hence does not make it possible to *express* properties such as the above.

The usual technique for modelling qualified calls is to treat the target as if it were a supplementary argument, understanding (1), for example, as **call** $invite_C$ (*Eri*, *Komatsu_Yasuke*, [*Today*, "19:30"], *Shin*) where $invite_C$ is the non-OO equivalent to *invite*, extended with one argument, as it would be written for example in the C language (or in the C output of an Eiffel compiler generating C code). Verification techniques will then handle the target just as it handles other arguments, through proof rules that transpose any property of the routine to a property of a call by substituting actual arguments for the formal arguments. This standard approach, however, will fail for properties such as (2) and (3) above, because it ignores the distinctive object-

oriented style of programming, detailed in the next section: the target of an OO call is more than just another argument.

The gist of the present paper is a simple notation that addresses the issue: for any call *x*.*r* (*args*), one may use *x'*, called the "negation" of *x*, to represent a back reference to the calling object, making it accessible to the target object (the object on which *r* is executed). Negative variables enjoy simple mathematical properties, such as *x*.*x'*= **Current** where **Current** denotes the "current object" of execution.

Through the negative variable, any analysis of the call has access to the caller context, enabling it to answer questions such as those in our example: if the caller has (as it must) a list *invited* of persons invited, the call can use the integer *Eri'*.*invited*.*count* and the test *Junko* ∈ *Eri'*.*invited*. More generally, the basic rule for reasoning about calls makes it possible to establish any property for the call *x*.*r* (*args*) by:

- Establishing the property for *r* (*x'*.*args*), that is to say, a call executed locally in the context of the target object, but with access to the caller's context through *x'*.
- Transposing the result back to the caller's context by prefixing it with "*x*."; occurrences of *x'* will normally disappear through the rule just mentioned.

The negative variable technique is an application to formal program analysis of a well-established mathematical technique: coordinate transform. Reasoning about the effect of a call is easier if we transpose the coordinates to the context of the target; then we interpret the results back in the caller context by performing the reverse coordinate transformation.

2 Overview: general relativity

2.1 In the space capsule

The negative variable technique is a response to the special nature of object-oriented programming, based on what has been called a principle of "General Relativity" [10]. This style sets OO programming apart from all other approaches even before one considers inheritance and other advanced techniques (which require it).

What is relative is the meaning of every operation in the program text: it applies to a "current object" ("this", "**Current**", "self") known only at the time of each execution. In a non-OO language, *x* = 3 states a property of a variable of the program; in an OO language, it states a property of "the *x* of the current object". The name *x* by itself is meaningless except with respect to that context.

We can think of the execution of an OO program (see fig. 1 on the next page) as occurring, at any given time, in a space vehicle that operates in its own set of coordinates (the current object). The cosmonauts responsible for executing these operations, and the operations themselves, do not see the larger context in which the vehicle exists. In fact the vehicle was launched from another, itself launched from yet another and so on up to the initial event that started the entire execution ("root procedure").

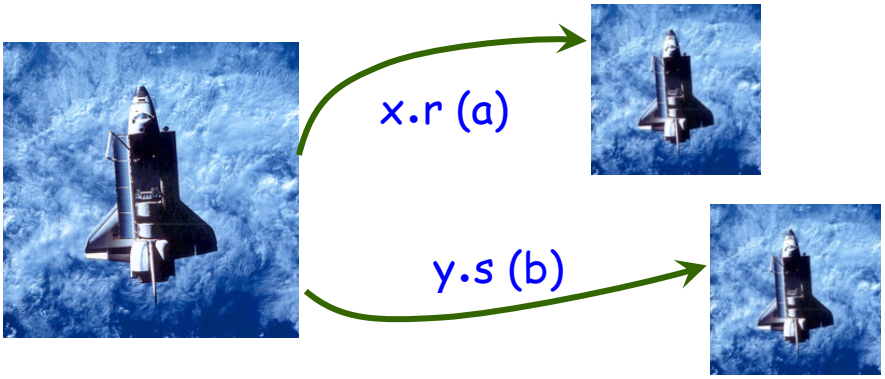

Fig. 1. Objects as space vehicles

2.2 The execution of an object-oriented program

In an OO language the operations are of two kinds: basic operations (assignments and such, sequenced by control structures such as conditionals and loops) and routine calls.

Every basic operation is relative to a designated object, the "current object" at the time of the operation's execution.

Routine calls have two variants:

- An **unqualified** call, written **call** *r* (*args*), executes the body of *r* on the current object, with the given arguments.
- A **qualified** call, written **call** *t*.*r* (*args*) causes another object, the **target** of the call, to execute the body of the routine on itself. The target is the object denoted by *t* at the time of execution. (The term "target" denotes both a static notion, the variable or expression *t* in the program text, and a dynamic one, the object attached to *t* in a particular call.)

For all operations of all kinds except one, the current object remains current: such operations execute within the current spacecraft. This is true in particular for unqualified calls. The one exception is qualified call. More precisely:

- At the start of a qualified call, the target object (the object attached to *t* in **call** *t*.*r* (*args*)) becomes the new current object. All the operations of the body of *r* will treat it as their current object.
- At the end of the execution of the qualified call, the formerly current object becomes current again.

This process is recursive since the execution of the routine can execute qualified calls on new targets.

The names of all variables occurring in an operation are understood in relation to the current object; the name *t* means "the *t* of the current object". This property applies to basic operations, such as the assignment *t* := *u*, but also to qualified calls: to determine the target object (target in the dynamic sense) in the call *t*.*r* (*args*) requires finding out the value of *t* (the target in the static sense) relative to the current object.

For generality we assume the Eiffel convention for executing entire programs: the execution consists of creating an instance of a designated "root class" and executing a designated "root procedure" on that target. (In languages with a more traditional "main program" we can posit a fictitious root object and consider the main program as the root procedure. Global variables do not fit well in the OO paradigm and do not exist in Eiffel, but their presence in other languages does not fundamentally affect the discussion.) Any operation is executed as part of a **current call**: the qualified call last started and not yet terminated (or, if there is no such call, meaning that we are at the top level of the execution flow, the root object). The target of that call serves as current object during the execution of the call; we may call it the **current target**, or just "the target", of the current call. The object that was current at the time of the call is the **caller object**, or just "the caller". In the root call the target is the root object and there is no caller (in all other cases there is a caller).

Object-oriented programming languages do not provide access to the caller object. The cosmonauts are in their own vehicle, and may launch new vehicles, but have no information or access to the vehicle that launched them.

For reasoning and verification purposes, we may need such access. If the current call is of the form $x.r\ (args)$, the negative variable, written x' (x negated), denotes a backward reference to the caller.

2.3 Negative variable basics

From an implementation perspective, negative variables are only a fiction, as no backward reference exists in the execution-time structure. Their role is to support reasoning and verification.

The notion was introduced in [11] and [12], in the context of developing the "alias calculus" for automatic may-alias analysis of OO programs; the calculus needs negative variables in the rules for qualified calls. The present work generalizes the original concept, showing that beyond alias analysis it can provide a framework for reasoning about a wide variety of properties of object-oriented programs.

The traditional approach, as noted, treats the target as if it were just one more argument, then applying the usual technique for dealing with arguments to calls: substitution of actuals for formals. This approach ignores the specific role of the target in object-oriented programming. As we have seen, it precludes the very expression of some important properties of the object store; aliasing properties are an example.

Negative variables define a basic semantic rule for handling qualified call, the fundamental operation of object-oriented programming. A simplified version of the rule (the full version appears in Section 5) is, for any property Π of program elements:

$$\Pi\ (x.r\ (args)) = x.\Pi\ (r\ (x'.args)) \qquad (4')$$

meaning, informally, that to derive a property of the qualified call $x.r\ (\ldots)$ we start from a property of the unqualified call $r\ (\ldots)$, where we interpret the arguments in relation to the calling context, hence the prefixing by x', then plunge the result back into that calling context by prefixing it with x.

We may, as noted, view the technique as coordinate transform. The rule tells us that to reason about a call, we first transport ourselves to the new spacecraft, evaluating Π for an unqualified call to *r*; in this evaluation, we may need back-access to the caller spacecraft's context, which we obtain by prefixing arguments with *x'*. Then we perform the reverse coordinate transform, getting everything back to the original context, by prefixing the results with *x*. As a result the property Π of the unqualified call, be it a value, a set, a list, a relation or a function is reinterpreted in the caller's context. In normal usage the result will no longer contain any occurrence of negative variables, thanks to rules stating that *x* and *x'* cancel each other out.

Section 3 further illustrates, through examples, the need for negative variables. Section 4 introduces the notations and conventions. Section 5 introduces the basic rules. Section 6 presents a number of applications; Section 7 provides comparison to previous work and Section 8 describes opportunities for further development.

3 Examples

The usual modes of reasoning about programs cannot be transposed to OO programs without adaptation. Even simple examples bring out the need for different techniques.

Consider classes *C* (client) and *S* (supplier). *S* has a simple argument-less procedure *r* with the postcondition $m = n$, where *m* and *n* are attributes (fields) of S. The procedure may be written as

```
r
        -- Among other possible effects, make sure that the fields m and n
        -- of the current object have equal values.
    do
        … Appropriate implementation, including the assignment m := n …
    ensure
            m = n
    end
```

In *C*, with *x* declared of type *S*, we may call $x.r$. We may deduce properties of such a call from the properties of the routine simply by prefixing the latter with "$x.$"; in this case the postcondition $m = n$ tells us, after actual-formal argument substitution, that the following will hold after the call:

$$x.m = x.n$$

To cover such cases it would suffice to use a naïve adaptation to object-oriented programming of the standard Hoare rule for procedures [6]:

$$\frac{\{P\ (f)\}\ \textbf{call}\ r\ (f)\ \{Q\ (f)\}}{\{x.P\ (a)\}\ \textbf{call}\ x.r\ (a)\ \{x.Q\ (a)\}}$$

-- *Warning*: naïve rule,
-- corrected in (6) below.

(Conventions: *f* stands for the list of formal arguments, *a* for the list of actual arguments; *P* and *Q* are explicitly parameterized by arguments, as an alternative to using substitution; we ignore recursion, which can be handled as described in [6]; we also ignore the role of class invariants, essential in practice for OO programs but not directly related to this discussion.)

The "." operator is a "distributed dot" which distributes the period of OO programming, used for calls and "path expressions" such as $x.y.z$ (which in fact are a special case of calls, resulting in this example from applying z to the result of applying y to x) over:

- An equality: $x.(u = v)$ denotes the equality $x.u = x.v$
- A set: $x.\{a, b, c\}$ denotes $\{x.a, x.b, x.c\}$.
- A pair: $x.[y, z]$ denotes $[x.y, x.z]$.
- More generally, a list: $x.[u, v, w]$ denotes the list $[x.u, x.v, x.w]$.
- A relation (a set of pairs): $x.\{[a, b], [c, d]\}$ denotes $\{[x.a, x.b], [x.c, x.d]\}$.
- A function (a special case of relations): if $f(u) = v$ then $x.(f(u)) = x.v$. Another way of denoting this property is to state that $x.(f(\underline{u})) = x.f(x.\underline{u})$. Note the double application of the dot; the reason is that stating that $f(u) = v$ means, if we look at f as a relation, that $[u, v] \in f$. This rule (like the preceding ones) is recursive: u could be, for example, a list.

As soon as we move on to less trivial properties, however, the simple device of prefixing properties by "$x.$" no longer works. Assume that r now has an argument and new postconditions:

```
r (u: T)
    do
        …
    ensure
        m.count > 0
        u = m
    end
```

and we call $x.r(a)$, for a of type T. Application of the naïve rule would give us meaningless properties for the call: $x.m.count > x.0$, where it makes no sense to prefix the constant 0 with "$x.$"; and $x.a = x.m$, where $x.a$ also makes no sense since a is an expression defined in the calling context, C, and prefixing it with x is pointless. We can get away in the first case through a general rule that identifies $x.const$, for any constant *const*, with *const*; but such tricks would not work for more significant properties such as the second postcondition. The problem is not syntactical but conceptual: every expression needs to be interpreted in the right object context (the right space vehicle). The actual argument a belongs to the **client** context (C) whereas m, an attribute of S, makes sense in the context of the **supplier** object.

With negative variables, the correct consequent for the procedure rule, replacing $\{x.P(a)\}$ **call** $x.r(a)$ $\{x.Q(a)\}$ above, is

$$\{x.P(x'.a)\} \textbf{ call } x.r(a) \{x.Q(x'.a)\}$$

stating that the arguments must be interpreted relative to the caller's context, accessible through the (fictitious) back-pointer *x'*. Applying this rule gives, as the second postcondition of the call:

$$x.x'.a = x.m$$

Then we apply two of the fundamental rules listed below: $x.x'$ = **Current**, and **Current.***e* = *e* for any expression *e*, giving

$$a = x.m$$

which correctly describes the effect of the call.

The example remains sufficiently simple to suggest that other rules could do the job, for example a set of ad hoc rules stating that $x.v = v$ for various kinds of elements *v* in the caller context. But such an approach fails to capture the "general relativity" property of object-oriented programming discussed in section 2, which implies that every program element or program property makes sense only with respect to a well-defined context. For a call, in particular, a property belongs to the context of either the caller (client) or the supplier. Consider the following new variant of our example routine, now with a precondition:

```
r (u: T)
    require
        u.p + q > 0
    do
        u.set_m (n + 1)
            -- The procedure set_m, in T, sets the value of the attribute m.
    ensure
        u.m = n + 1
    end
```

Consider the call $x.r\ (a)$. By applying the rule we get as a postcondition of the call

$$x.(x'.a).m = x.n + x.1$$

(distributing "." over addition, as justified in Section 4). Simplifying, this yields

$$a.m = x.n + 1$$

Similarly, the precondition making this call legal (assuming, as implied by the example, that *p* and *q* are integer attributes of classes *T* and *S* respectively) is

$$x.(x'.a).p + x.q > x.0$$

or, after simplification:

$$a.p + x.q > 0$$

Note how *u*.*p* refers to a property of the client context and *q* to a property of the supplier context. The general rule makes it possible to switch back and forth effortlessly between these contexts:

- As stated in the routine, the properties (here a precondition and a postcondition, but the same rules will apply to any kind of a property) are expressed relative to the supplier context. *T* has access to the client context through the formal arguments which, however, describe an unknown caller.
- When the caller is known, here *x*, the formal arguments can be transposed back to the client context through prefixing by *x'*, representing a fictitious back pointer.
- The resulting properties are also transposed back to the client context, but in this case through prefixing by *x*.

This example illustrates only one of the applications of the general approach: the Hoare-style rule. We will now explore the general framework and the general rules.

4 Notations and conventions

The discussion is applicable to any object-oriented language. We assume an imperative language, with an assignment instruction written *target* := *source*, and routines (methods) that can be functions (returning a result) or procedures (changing the state). Examples of such languages include Java, Eiffel and C#. The imperative character of the language has no influence on the discussion, so the results are also applicable to a functional (applicative) object-oriented language.

We make the assumption that (as in Eiffel) no direct assignment is permitted to fields of an object: rather than *x*.*a* := *v*, the programmer must write a procedure call *x*.*set_a* (*v*), with the appropriate setter procedure *set_a* declared in the corresponding class. (Some languages, such as Eiffel, allow the syntax *x*.*a* := *v* provided the class author has marked the setter procedure as "assigner"; but this instruction is not an assignment, only a different syntactical form of the explicit call *x*.*set_a* (*v*). C#'s "properties" have a similar role.) This restriction, justified by information hiding principles, does not limit the application of the approach to languages that permit direct field assignments: one should simply replace such assignments, for the purpose of program analysis or verification, by the application of a suitable setter.

Among routines we will only consider procedures, with the understanding that a function call can be handled as a procedure call followed by assignment of the result.

Calls, qualified and unqualified, are as discussed in Section 2.2, which also introduced the notions of target and caller objects.

Since the matter of defining the semantics of unqualified calls is independent from the problem tackled in this article, we assume that such a semantic definition exists. The simplest way to define it (depending on the rules of argument passing) is that the semantics of call *r* (*a*) is the semantics of the *body* of the routine *r*, after substitution of actual arguments *a* for formals.

The notation **old** *e*, for an expression *e*, denotes the value that *e* had at the start of the current call. **Current** denotes the current object.

The dot operator is generalized as explained in Section 3, complemented by the convention that if c is a constant then $x.c$ is c. The combination of all the variants allows us to generalize the distributive dot to a wide class of operators:

$x.(u \boxtimes v)$ is $(x.u) \boxtimes (x.v)$

where $\boxtimes$ is any operator that can be defined from functions, relations, sets, pairs, lists and equality; for example, in a pure OO language, $u + v$ on numerical arguments is simply an abbreviation for the function call $u.plus\ (v)$, so that by application of the second case $x.(u + v)$ is $(x.u) + (x.v)$.

Thus generalized, the dot operator covers, in our experience so far, all the kinds of properties that one may want to express about a program.

5 Negative variables: definitions and rules

For any variable x that may be used as target of a qualified call, the "negation" of x, written x', denotes a reference, defined during the execution of a qualified call of target x, to the object that started this call. (The existence of such an object is traditionally checked at run time, through "null pointer" exceptions, but in some recent languages it has become a static property enforced by the compiler, as in Eiffel's "void safety" mechanism [9]. The present discussion assumes that all calls are void-safe, i.e. pointers are not null.)

The following rules are applicable to any variable x and its negation x', and to any expression e of the target programming language[1]:

N1 **Current**' = **Current**
N2 e.**Current** = e
N3 **Current**.e = e
N4 $x.x'$ = **Current**[2]
N5 x'.(**old** x) = **Current**
N6 **old** x' = x'

N1 enables us, by application of the call rules that follow, to treat a qualified call of the form **Current**.r (a) as equivalent to the unqualified call r (a). In N5, note the use of **old**, without which the rule would be unsound since it is in principle possible for a routine r, during the execution of $x.r$ (a), to modify (through callbacks) the value of the very variable x that the client object used as target of the current call. Such a setup is of course error-prone; we say that a routine is *nonprodigal* if it cannot modify the target of its own call. For a nonprodigal routine, N5 yields a more practical variant (symmetric with N4):

$$x'.x = \textbf{Current}$$

[1] Depending on the rules of the programming language, occurrences of e may have to be enclosed in parentheses to avoid syntactic ambiguity.

[2] Depending on the context $x.x'$ can also be replaced with an implicit current object that is usually omitted, for example, $x.x'.y$ simplifies to y.

N6 expresses that the back link to a routine's caller cannot be changed: your spacecraft was launched by a given spacecraft, and there is nothing you can do about it.

In the application to aliasing, rules N4 and N5 may produce an over-approximation for some cyclic structures. Adding integer indexes can improve the precision. This issue has no influence on the rest of the discussion and is hence not considered further in this article.

The fundamental rule was previewed in Section 3 and will now be given in full. It considers an arbitrary property Π applicable to a program element such as an instruction, an expression, a class or an entire program.

In the initial version, Π had just one argument, the program element. In practice, any realistic framework for reasoning about programs involves properties of *two* arguments: a program element, and an **environment** representing what is already known, or assumed, about the context of the program element's current execution. In static analysis, for example, we may compute the "defined" and "used" variables of a block in relation to the values of these properties for the context in which it is executed. As another example, the alias calculus [11] is a set of rules giving the value of a » p for the various constructs p of an OO programming language; a is an alias relation, consisting of a set of pairs of expressions that may be aliased to each other (denote the same object) at a given program point, and a » p is the new alias relation that results from executing p when the original alias relation is a. In this case the alias relation is the environment.

With this convention, the fundamental rule for reasoning about properties Π of object-oriented programming languages is

$$\Pi\ (\textbf{call}\ x.r\ (args),\ env) = x.\Pi\ (\textbf{call}\ r\ (x'.args),\ x'.env) \qquad (4)$$

The rule enables us to deduce, from a property of the unqualified call (that is to say, a property that makes sense in the context of the supplier object), the corresponding property of a qualified call (in the client context).

The prefixing by "*x'*." must be applied to the environment as well as to the actual arguments, since both are relative to the client context.

The rule is applicable to properties for which the prefixing by "*x'*." is defined, as discussed in section 4. It appears to cover all properties used in existing frameworks for semantics and verification of programs, from static analysis to denotational and axiomatic semantics.

In denotational (and operational) semantics, a common scheme is to define a program construct such as an instruction as a function (usually partial) in *Environment* → *State* → *State*, preceded by *Arguments* → for a routine. The Fundamental Rule applied to this framework gives[3]:

$$\textbf{call}\ x.r = \lambda\ args \mid \lambda\ env \mid x.(\textbf{call}\ r\ (x'.args)\ (x'.env)) \qquad (5)$$

[3] It is common practice to define the semantics through a "meaning function" M, which for any program element p yields a mathematical function M [p], the "denotation" of p. The alternative, used here for simplicity, is to define every construct directly as a mathematical function, skipping the meaning function. The "M" variant is easy to deduce from this form.

In axiomatic semantics, the environment does not need to be explicitly stated since it is embedded in the precondition, postcondition and invariant[4]:

$$\frac{\{P\ (f)\ \textbf{and}\ INV\}\ \textbf{call}\ r\ (f)\ \{Q\ (f)\ \textbf{and}\ INV\}}{\{x.P\ (x'.a)\ \textbf{and}\ x.INV\}\ \textbf{call}\ x.r\ (a)\ \{x.Q\ (x'.a)\ \textbf{and}\ x.INV\}} \quad (6)$$

6 Applications

We now show some potential uses of the rules given.

The alias calculus rule given in [11] is a direct application of the fundamental rule (4). The purpose of the alias calculus is to answer, for any two reference (pointer) expressions *e* and *f* and any program point *pp* at which they are both defined, the question: "can *e* and *f*, at any time execution reaches *pp*, have as their values references to the same object?". To this end, the calculus is a set of rules to compute $a \raisebox{0.5ex}{\scriptsize »} p$ for every programming language construct *p*, where *a* is an alias relation, containing all pairs of expressions that may be aliased to each other. If *a* is the alias relation before execution of *p*, $a \raisebox{0.5ex}{\scriptsize »} p$ will be the alias relation after that execution. The rule for qualified calls, where *l* denotes a list of actual arguments, is:

$$a \raisebox{0.5ex}{\scriptsize »} \textbf{call}\ x.r\ (l) = x.((x'.a) \raisebox{0.5ex}{\scriptsize »} \textbf{call}\ r\ (x'.l\ )) \quad (7)$$

This rule shows a typical use of the negative variable technique in its full extent. Both the initial alias relation *a* and the list of arguments *l* are defined on the client's side (the caller's context). To apply the unqualified call rule on the right side of (7), we must be able to interpret *a* and *l* on the supplier side; this is achieved by prefixing both of them with "*x'.*" to interpret them in the context of the callee. The expression $(x'.a) \raisebox{0.5ex}{\scriptsize »} \textbf{call}\ r\ (x'.l\ )$ then gives us the resulting alias relation, but still in the supplier context. To transpose it back to the client context, which is where we need the final result, we prefix that supplier-side relation with "*x.*", yielding a client-side property.

Here now are examples of application of the axiomatic rule (6). Consider a routine *sign* used to sign a message with a signature computed from a key, according to the specification:

$$\{is\_valid\_key\ (k)\}\ \textbf{call}\ sign\ (k, s)\ \{signed\ (k, s)\}$$

where *k* is a key and *s* a message to be signed. Applying the rule (6) to a qualified call

$$\textbf{call}\ x.sign\ (y, z)$$

where *y* and *z* are local variables or attributes, we get

$$\{x.(is\_valid\_key\ (x'.y))\}\ \textbf{call}\ x.sign\ (y, z)\ \{x.(signed\ (x'.y, x'.z))\}$$

which rules N4 and N3 from Section 5 allow us to simplify into

[4] This rule implies some conditions on callbacks (to ensure that they satisfy the invariant), an issue separate from the theme of this article.

$$\{x.is\_valid\_key\ (y)\}\ \textbf{call}\ x.sign\ (y, z)\ \{x.signed\ (y, z)\}$$

reflecting the intuitive result.

Another application area is purity. A routine is pure if it does not modify the state. In the case of *weak* purity [3] it may, however, create and modify new objects. Consider a pure routine *r* and purity (strong or weak) for *r* relative to an expression *e*:

$$\{\ldots\}\ \textbf{call}\ f\ (t)\ \{e == \textbf{old}\ e\}$$

where == expresses deep equality (equality not only of the values themselves but of all reachable objects). Rule (6) yields

$$\{\ldots\}\ \textbf{call}\ x.f\ (a)\ \{x.(x'.e == \textbf{old}\ x'.e)\}$$

The postcondition can be simplified through distributivity to

$$x.x'.e == x.(\textbf{old}\ x').e$$

which through N4, N3 and N6 gives

$$e == \textbf{old}\ e$$

In other words, a qualified call to a pure routine (weak or strong) is itself pure.

The same approach generalizes to a full-fledged frame rule. A frame rule is a specification of which properties an operation may modify; it is typically stated by listing the possibly affected expressions in a **modifies** or **only** clause. (Purity is a special case, expressed as a frame clause with an empty list of attributes.) Consider a routine with such a specification:

```
f (p: X ; q: Y)
    …
    ensure
        a = p
        p.b = q
        g.v = old g.v + 1
    only
        a, p.b, g.v
    end
```

Ignoring the rest of the postcondition, we may write the frame property in Hoare style as

$$\{\ldots\}\ \textbf{call}\ f\ (p, q)\ \{\textbf{only}\ a, p.b, g.v\}$$

The transposition to a qualified call through (6) is

$$\{\ldots\}\ x.\textbf{call}\ f\ (p, q)\ \{x.(\textbf{only}\ a, x'.p.b, g.v)\}$$

which after simplification yields

$$\{\ldots\}\ x.\textbf{call}\ f\ (p, q)\ \{\textbf{only}\ x.a, p.b, x.g.v\}$$

SCOOP, a concurrency model developed for simple and reliable concurrent programming through the safe use of shared resources ([14]), provides another example of application of negative variables. SCOOP binds the concurrency structure to the object-oriented structure by partitioning the object space into a number of "regions", each associated with a given thread of control or "processor", the "handler" of these objects, so that a qualified call $x.r\ (args)$ is always processed by the handler of the target object (the object denoted by x). If a variable x may denote an object in another region (so that calls $x.r\ (args)$ will be handled by a different processor), it must be declared **separate**. The SCOOP type system includes a set of rules to ensure consistent semantics. The rules imply in particular that if x is separate the formal arguments corresponding to *args* must also be declared separate. The reason for this rule is that if the call is executed on behalf of processor A and the processor of x is B, *args* denotes objects in A, which for B are separate and hence must be declared accordingly. In other words, the notion of separateness is always relative.

Applying this observation to negative variables yields the rule that if the variable x is separate, its negation x' is also separate (if the supplier S is separate from the client C, then C is separate from S).

Then in the application of any semantic rule, for example the axiomatic rule (6), to a call

$$\textbf{call}\ x.r\ (args)$$

the formal arguments will be prefixed with "$x'.$", since the rules deduce properties of the qualified call from the properties of its unqualified version **call** $r\ (x'.args)$. This observation indicates that, in the program text, the formal arguments should themselves be declared as **separate** for consistency. This is indeed one the rules of the SCOOP type system. Here we see it arising as a consequence of the general properties of negative variables, without any domain-specific reasoning.

7 Related work

Even before OO came to the scene, back pointers were used to simplify and optimize the implementation of algorithms working on complex data structures. Such back-pointers, however, are physically present in the corresponding data structures and usually take up memory (although some algorithms, such as the Deutsch-Schorr-Waite stack-free technique for tree or graph traversal, reuse other fields for the temporary representation of back pointers). Any reasoning about and manipulation of such back pointers follows the same rules as for other references and makes no use of their specific nature. Negative variables as discussed in this article are a conceptual mechanism to reason about OO programs; it is not necessary (but of course not prohibited) to turn them into physical components of the data structure representations.

Operating systems have used back pointers for a long time. They serve in particular to keep references to the parent directory in a file system, making it possible to use

".."to refer to the parent directory without knowing the current directory's actual location. OO languages usually do not support such a mechanism for their run-time data structures, since this would require keeping track of the invocation structure. Negative variables give us the concept without requiring its implementation.

Usually the axiomatic semantics of a method call is described using substitution rules of actual arguments to formal arguments, target of a call as the current object, and return value as a result after the call; see in particular the work of Müller, Leino and their colleagues [13] [8] [4]. Negative variables are not explicitly used in these approaches and are not available for formal reasoning on program properties. Meyer's "Calculus of Object Programs" [12] is an exception, integrating the alias calculus [11] and negative variables. Schoeller's path-based alias analysis [16] comes close to the need to use negative variables, but still uses the standard substitution technique to describe the semantics of a qualified method call. Other semantic descriptions of object-oriented languages, such as algebraic specifications [5], also use substitution.

The specifications and subtleties of pure functions are described by Darvas, Müller and Leino in [3] and [4]. We used a simplified version of the specification.

Nienaltowski provides in [14] an analysis of the type requirements for safe concurrent programming and the resulting design of a type system for SCOOP. The approach covers both the attachment (non-nullness) status and the separateness status of the target and arguments of a call. The target's attachment status ensures that a call cannot lead to an exception at run-time. Meyer, Kogtenkov and Stapf address this issue in [9]; in the examples we have taken the assumption of attachment for granted. The other key property presented in [14] can be deduced, as we have seen, from the general rules for negative variables.

Shield [17] makes the current object explicit through a variable *self*. He treats every qualified call as an operation that saves the value of the current object to a stack, and assigns the call's target to *self*. After the call, the original value is restored. The author notes that this technique works for recursive calls only when the stack stores a reference to the current object, not the object itself, on the stack. The present work makes a similar assumption for negative variables.

Research in automatic program verification, particularly around the ESC/Java and JML languages and verification systems, uses the notion of *model fields* [1] or *ghost variables* [2]: variables used only for verification, without influence on the generated code, as in the classic Owicki-Gries approach [15] to the verification of concurrent programs. The variables should be specified by the developer and should be kept in sync with the rest of the program in the annotation sections. The verifier can use the properties of these variables to perform the verification of the actual code. Negative variables have a similar status: useful for reasoning and verification, but not used directly in the program.

Kassios [7] proposes an extension to ghost fields by introducing implicit back-pointers that are automatically added to the explicit ghost fields as soon as the corresponding forward field is marked as **tracked**. The back-pointers are really object sets and are used to turn unstable class invariants into stable ones by making sure that data reflecting the references to the given object are always synchronized with the references themselves (the example in [7] uses reference count for this purpose). This

approach makes it possible to apply separation logic rules to cases when actual object disjointness is replaced by *observable* disjointness.

Wei Ke at al [18] use a special $-edge in object state graphs to denote a call stack. Whenever a qualified call is made, a new $-edge that points to the current root is created and points from the new root object node. On return the $-edge is removed and the current object is popped from it. Our approach is quite similar but goes beyond graph-based framework and state representation. Moreover, it allows using both – normal and reverse edges indistinguishably in cases when caller's and callee's contexts are to be taken into account, as in alias calculus.

8 Implementation, discussion and future work

We have proposed a simple concept, negative variables, reflecting an essential property of object-oriented computation: the relativity of all program constructs to a "current object" known only at the very last moment during execution. The corresponding fundamental rule, (4), provides a general framework for reasoning about object-oriented programs regardless of the programming language and semantic framework; directly applicable versions of the general rule have been shown for specific frameworks such as denotational (5) and Hoare-style axiomatic (6) semantics, as well as alias analysis. Other examples, such as the application to concurrency, show the generality of the approach.

The mechanisms for dealing with negative variables, particularly in the axiomatic and alias calculus applications, have been implemented in EVE, the research version of the EiffelStudio IDE (integrated development environment) and have been used to prove a number of properties of example programs.

The discussion has not considered some important OO mechanisms such as inheritance, polymorphism, genericity, expanded (value) types, closures (C# delegates, Eiffel agents) and the full extent of concurrency; specific rules may (or not) be needed to handle them. More generally, the use of negative variables in the verification of ever larger object-oriented programs may lead to generalizations of the techniques described here.

Acknowledgments

This work was performed in the ITMO Software Engineering Laboratory, made possible by a grant from the mail.ru group. Support from the CME, "*Concurrency Made Easy*" Advanced Investigator Grant of the European Research Council (grant number 291389 under FP7/2007-2013) is also gratefully acknowledged.

We are indebted to the anonymous referees for their very useful comments.

It is a pleasure to dedicate this article to Professor Kokichi Futatsugi in celebration of thirty-five years of friendship with the first author (going back to lectures in the same session of the IFIP 78 World Computer Congress in Tokyo under the aegis of Harlan Mills) and of his seminal contribution to the science and practice of software specification as reflected in particular in the CaféOBJ language and system.

References

1. Chalin, Patrice, Joseph R. Kiniry, Gary T. Leavens, and Erik Poll. *Beyond Assertions: Advanced Specification and Verification with JML and ESC/Java2*. – Formal Methods for Components and Objects (FMCO) 2005, Revised Lectures, pages 342-363. Volume 4111 of Lecture Notes in Computer Science, Springer Verlag, 2006.
2. Cohen, Ernie, Michał Moskal, Wolfram Schulte, Stephan Tobies. *Local Verification of Global Invariants in Concurrent Programs.* - Computer Aided Verification, LNCS, Springer, 2010, pp.480-494. doi:10.1007/978-3-642-14295-6_42
3. Darvas, Ádám, K. Rustan M. Leino. *Practical reasoning about invocations and implementations of pure methods.* – Proceedings of the 10th international conference on Fundamental approaches to software engineering, pp. 336--351, Volume 442 of LNCS, Springer-Verlag, 2007.
4. Darvas, Ádám, Peter Müller. *Reasoning about Method Calls in Interface Specifications.* – Journal of Object Technology, vol. 05, no. 5, Special Issue: ECOOP 2005 Workshop FTfJP, June 2006, pages 59–85, http://www.jot.fm/issues/issues 2006 06/article3
5. Fronk, Alexander. *An Approach to Algebraic Semantics of Object-Oriented Languages.* – Software-Technology, University of Dortmund, Germany, 2003. DOI:2003/2682.
6. Hoare, C.A.R.: *Procedures and Parameters, an Axiomatic Approach.* Symposium on Semantics of Algorithmic Languages (1971), pp. 102-116, doi:10.1007/BFb0059696
7. Kassios, Ioannis T. and Kritikos, Eleftherios. *A Discipline for Program Verification based on Backpointers and its Use in Observational Disjointness.* ETH Zurich, Department of Computer Science (2012). http://dx.doi.org/10.3929/ethz-a-007560318
8. Leino, K. Rustan M.. Ecstatic. *An object-oriented programming language with an axiomatic semantics.* – Digital Equipment Corporation Systems Research Center, 1996.
9. Meyer, Bertrand, Alexander Kogtenkov and Emmanuel Stapf. *Avoid a Void: The Eradication of Null Dereferencing*, in *Reflections on the Work of C.A.R. Hoare*, eds. C. B. Jones, A.W. Roscoe and K.R. Wood, Springer-Verlag, 2010, pages 189-211.
10. Meyer, Bertrand. *Object-Oriented Software Construction*, 2nd edition, Prentice Hall, 1997.
11. Meyer, Bertrand. *Steps Towards a Theory and Calculus of Aliasing*, in *International Journal of Software and Informatics*, 2011.
12. Meyer, Bertrand. *Towards a Calculus of Object Programs*, in Judith Bishop Festschrift, eds. Karin Breitman and Nigel Horspool, Lecture Notes in Computer Science, Springer-Verlag, 2012.
13. Müller, Peter. *Modular Specification and Verification of Object-Oriented Programs.* PhD thesis, Lecture Notes in Computer Science, Springer-Verlag, 2001.
14. Nienaltowski, P. *Practical framework for contract-based concurrent object-oriented programming*. – PhD dissertation 17061, Department of Computer Science, ETH Zurich, February 2007. Other SCOOP references at http://se.inf.ethz.ch/research/cme/.
15. Owicki, Susan and David Gries. An axiomatic proof technique for parallel programs, *Acta Informatica*, vol. 6, no. 4, 1976, pp. 319-340.
16. Schoeller, Bernd. *Aliased-based Reasoning for Object-Oriented Programs.* Tech. Report, ETH Zurich, se.ethz.ch/people/schoeller/pdfs/10-Annual_Report_CSE_ETHZ_2005.pdf, 2005.
17. Shield, Jamie. *Towards an Object-Oriented Refinement Calculus.* - PhD Thesis, The University of Queensland, 2004.
18. Wei Ke, Zhiming Liu, Shuling Wang, Liang Zhao. *A graph-based generic type system for object-oriented programs*. – Frontiers of Computer Science, vol.7, no 1, pp.109-134, doi: 10.1007/s11704-012-1307-8. SP Higher Education Press, 2013.