Journal of Computer and System Sciences 78 (2012) 823–852

Contents lists available at SciVerse ScienceDirect

Journal of Computer and System Sciences

www.elsevier.com/locate/jcss

# Verification of object-oriented programs: A transformational approach

Krzysztof R. Apt [a,b,*], Frank S. de Boer [a,c], Ernst-Rüdiger Olderog [d], Stijn de Gouw [a,c]

[a] *Centre for Mathematics and Computer Science (CWI), Amsterdam, The Netherlands*
[b] *University of Amsterdam, Institute of Language, Logic and Computation, Amsterdam, The Netherlands*
[c] *Leiden Institute of Advanced Computer Science, University of Leiden, The Netherlands*
[d] *Department of Computing Science, University of Oldenburg, Germany*

A R T I C L E   I N F O

*Article history:*
Received 21 April 2010
Received in revised form 16 February 2011
Accepted 5 August 2011
Available online 25 August 2011

*Keywords:*
Object-oriented programs
Null references
Aliasing
Inheritance
Subtyping
Syntax-directed transformation
Recursive programs
Program verification
Strong partial correctness
Relative completeness

A B S T R A C T

We show that verification of object-oriented programs by means of the assertional method can be achieved in a simple way by exploiting a syntax-directed transformation from object-oriented programs to recursive programs. This transformation suggests natural proofs rules and its correctness helps us to establish soundness and relative completeness of the proposed proof system. One of the difficulties is how to properly deal in the assertion language with the instance variables and aliasing. The discussed programming language supports arrays, instance variables, failures and recursive methods with parameters. We also explain how the transformational approach can be extended to deal with other features of object-oriented programming, like classes, inheritance, subtyping and dynamic binding.

## 1. Introduction

### *1.1. Background and motivation*

Ever since its introduction in [14] the assertional method has been one of the main approaches to program verification. Initially proposed for the modest class of **while** programs, it has been extended to several more realistic classes of programs, including recursive programs (starting with [15]), programs with nested procedure declarations (see [19]), parallel programs (starting with [23] and [24]), and distributed programs based on synchronous communication (see [4]). At the same time research on the theoretical underpinnings of the proposed proof systems resulted in the introduction in [10] of the notion of relative completeness and in the identification of the inherent incompleteness for a comprehensive ALGOL-like programming language (see [9]).

However, (relative) completeness of proof systems proposed for current object-oriented programming languages (see the related work section below) remained largely beyond reach because of the many intricate and complex features of languages like Java. In this paper we present a transformational approach to the formal justification of proof systems for object-oriented programming languages. We focus on the following main characteristics of objects:

- objects possess (and *encapsulate*) their own (so-called instance) variables, and
- objects interact via *method* calls.

[*] Corresponding author at: Centre for Mathematics and Computer Science (CWI), Amsterdam, The Netherlands.
*E-mail addresses:* apt@cwi.nl (K.R. Apt), F.S.de.Boer@cwi.nl (F.S. de Boer), olderog@informatik.uni-oldenburg.de (E.-R. Olderog).

0022-0000/$ – see front matter © 2011 Elsevier Inc. All rights reserved.
doi:10.1016/j.jcss.2011.08.002

The execution of a method call involves a temporary *transfer* of control from the local state of the caller object to that of the called object (also referred to by *callee*). Upon termination of the method call the control returns to the local state of the caller. The method calls are the *only way* to transfer control from one object to another. We illustrate our approach by a syntax-directed transformation of the considered object-oriented programs to *recursive programs*. This transformation naturally suggests the corresponding proof rules. The main result of this paper is that the transformation preserves (relative) completeness.

To make this approach work a number of subtleties need to be taken care of. To start with, the 'base' language needs to be appropriately chosen. More precisely, to properly deal with the problem of avoiding methods calls on the **null** object we need a failure statement. In turn, to deal in a simple way with the call-by-value parameter mechanism we use parallel assignment and block statement. Further, to take care of the local variables of objects at the level of assertions we need to appropriately define the assertion language and deal with the substitution and aliasing.

We introduced this approach to the verification of object-oriented programs in our recent book [3] where we proved soundness. The aim of this paper is to provide a systematic and self-contained presentation which focuses on (relative) completeness and to explain how to extend this approach to other features of object-oriented programming. Readers interested in example correctness proofs may consult [3, pp. 226–237].

## 1.2. Related work

The origins of the proof theory for recursive method calls presented here can be traced back to [12]. However, in [12] the transformational approach to soundness and relational completeness was absent and failures were not dealt with. In [25] an extension to the typical object-oriented features of *inheritance* and *subtyping* is described. There is a large literature on assertional proof methods for object-oriented languages, notably for Java. For example, [17] discusses a weakest precondition calculus for Java programs with annotations in the Java Modeling Language (JML). JML can be used to specify Java classes and interfaces by adding annotations to Java source files. An overview of its tools and applications is provided in [8]. In [16] a Hoare logic for Java with abnormal termination caused by failures is described. However, this logic involves a major extension of the traditional Hoare logic to deal with failures for which the transformational approach breaks down.

Object-oriented programs in general give rise to dynamically evolving *pointer* structures as they occur in programming languages like Pascal. This leads to the problem of *aliasing*. There is a large literature on logics dealing with aliasing. One of the early approaches, focusing on the linked data structures, is described in [21]. A more recent approach is that of *separation logic* described in [28]. In [1] a Hoare logic for object-oriented programs is introduced based on an explicit representation of the global store in the assertion language. In [5] restrictions on aliasing are introduced to ensure encapsulation of classes in an object-oriented programming language with pointers and subtyping.

Recent work on assertional methods for object-oriented programming languages (see for example [6]) focuses on *object invariants* and a corresponding methodology for *modular* verification. In [22] also a class of invariants is introduced which support modular reasoning about complex object structures.

Formal justification of proof systems for object-oriented programming languages have been restricted to soundness (see for example [30] and [18]). Because of the many intricate and complex features of current object-oriented programming languages (relative) completeness remained largely beyond reach. Interestingly, in the above-mentioned [1] the use of the global store model is identified as a potential source of *incompleteness*.

## 1.3. Technical contributions

The proof system for object-oriented programs presented in our paper is based on an assertion language comparable to JML. This allows for the specification of dynamically evolving object structures at an abstraction level which coincides with that of the programming language: in this paper the only operations on objects we allow are testing for equality and dereferencing. Our transformation of the considered object-oriented programs to recursive programs preserves this abstraction level. As a consequence we have to adapt existing completeness proofs to recursive programs that use variables ranging over *abstract data types*, e.g., the type of objects.

In this paper we focus on *strong* partial correctness which requires absence of failures. Note that absence of failures is naturally expressed by a corresponding condition on the *initial* state, that is, by a corresponding notion of *weakest precondition*. Similarly, total correctness of recursive programs is also naturally expressed by weakest preconditions, see [2].

To express weakest preconditions over abstract data types in an assertion language [29] use a coding technique that requires a weak second-order language. In contrast, we introduce here a new state-based coding technique that allows us to express weakest preconditions over abstract data types in the presence of infinite arrays in a *first-order* assertion language.

Further, we generalize the original completeness proof of [13] for the partial correctness of recursive programs to weakest preconditions in order to deal with strong partial correctness. The completeness proof of [13] is based on the expression of the *graph* of a procedure call in terms of its strongest postcondition of a precondition which "freezes" the initial state by some fresh variables. If we use instead weakest preconditions to express the graph of a procedure call these freeze variables are used to denote the *final* state. Because of possible *divergence* or *failures* however we cannot eliminate in the precondition these freeze variables by existential quantification. As such the completeness proof of [13] breaks down. We show in this