Automatic verification of Java programs with dynamic frames

Jan Smans¹, Bart Jacobs¹, Frank Piessens¹ and Wolfram Schulte²

¹ Katholieke Universiteit, Leuven, Belgium. E-mail: jan.smans@cs.kuleuven.be
² Microsoft Research, Redmond, USA

Abstract. Framing in the presence of data abstraction is a challenging and important problem in the verification of object-oriented programs Leavens et al. (Formal Aspects Comput (FACS) 19:159–189, 2007). The dynamic frames approach is a promising solution to this problem. However, the approach is formalized in the context of an idealized logical framework. In particular, it is not clear the solution is suitable for use within a program verifier for a Java-like language based on verification condition generation and automated, first-order theorem proving. In this paper, we demonstrate that the dynamic frames approach can be integrated into an automatic verifier based on verification condition generation and automated theorem proving. The approach has been proven sound and has been implemented in a verifier prototype. The prototype has been used to prove correctness of several programming patterns considered challenging in related work.

Keywords: Program verification, Dynamic frames, Frame problem, Data abstraction

1. Introduction

How does one specify which memory locations a method can touch? This problem is called the frame problem [BMR95]. As an example, consider the class Cell from Fig. 1a. This class provides a constructor for creating new instances and a mutator for modifying their state. The client code in Fig. 1b uses these methods to create and modify two Cell objects. At the end of the code snippet, an assert statement checks that c₁.x holds 5. Can we prove based on Cell’s method contracts that this assertion never fails? The answer is no. setX’s contract allows the method to change the program state arbitrarily, as long as it ensures that c₁.x equals v on exit. In particular, the contract does not prevent c₂.setX(10); from modifying c₁.x.
class Cell {
    int x;

    Cell() {
        ensures x = 0;
        { x := 0; }
    }

    void setX(int v) {
        modifies this x;
        ensures x = v;
        { x := v; }
    }
}

(a)

(b)

Fig. 1. A class Cell and some client code

class Cell {
    int x;

    Cell() {
        modifies nothing;
        ensures x = 0;
        { x := 0; }
    }

    void setX(int v) {
        modifies this x;
        ensures x = v;
        { x := v; }
    }
}

Fig. 2. A new version of the Cell with modifies annotations

Modifies clauses are a standard technique in verification to solve the frame problem. A modifies clause traditionally consists of a comma-separated list of memory locations. A method satisfies its modifies clause if all allocated locations except those listed in the modifies clause retain their value. For example, consider the new version of Cell from Fig. 2. setX’s modifies clause only lists the location this.x which indicates that the method can only change the value of that particular location. A modifies clause does not prevent a method from modifying fields of objects allocated during execution of the method itself. For example, although the modifies clause of the constructor contains no locations, it can modify the fields of the object created by the constructor.¹

The new contracts for Cell shown in Fig. 2 suffice to prove that the assert statement of Fig. 1b never fails. Informally, the reasoning is as follows. At program location A, the postcondition of c1.setX(5); holds: c1.x = v. Since c1 is allocated and c2’s constructor does not modify fields of allocated objects, c1.x still equals 5 after the constructor. Moreover, c2 is different from c1, since c2 is a newly allocated object. It follows from setX’s modifies annotation that c2.setX(10); only modifies c2.x and does not affect c1.x. We may conclude that the condition of the assert statement, c1.x = 5, always evaluates to true.

Data abstraction is crucial in the construction of modular programs, since it ensures that internal changes in one module do not propagate to other modules. However, the class Cell and its contracts were not written with data abstraction in mind. In particular, client code must directly access the internal field x to query a Cell’s internal state. Moreover, Cell’s method contracts are not implementation-independent as they mention x. To solve the former issue, programmers typically restrict the visibility of internal fields and add getters to their classes. For example, Fig. 3a extends Cell with a getter getX. The client code of Fig. 3b can now use the getter instead of the internal field x. Note that getX has a pure modifier which indicates the method does not have side-effects and that it can be used within method contracts. In the remainder of this paper, we use the terms getter and pure method interchangeably.

¹ This description of object creation is slightly simplified. In our verifier prototype, object creation is modeled by first allocating new memory locations and afterwards calling the constructor. Each constructor in a class C has a default modifies clause that allows it to modify fields of this in C and superclasses of C.