Abstract

Program composition and modularity have proven themselves as an important approach for simplifying the design and verification of large programs. The contributions of this paper include:

1. A proposal of a modular and complete proof system for fair termination of a parallel-composed program.

2. A proposal of a proof system for union and superposition.

Modular termination proof systems that have been suggested before are defined for models with an unfair scheduler. The proof approach presented in them fails to be complete in a model with a fair scheduler. The main idea suggested here which allows for the development of a modular and complete proof system for fair termination is a new program property, called gapped-termination.

1 Introduction

Program composition and modularity have proven themselves as an important approach for simplifying the design and verification of large programs. This approach enables separate development and verification of parts of a software system. Moreover, programs and their proofs can be reused in different applications [G86].

The contributions of this paper are:

1. A proposal of a modular and complete proof system for fair termination of parallel-composed programs.

2. A proposal of a proof system for partial correctness and termination of a jigsaw-composed program [FFG90].

3. A proposal of simplified versions of the proof system for two special cases: union [CM88] and superposition [K87, BF88, CM88, FF90].

1. A modular and complete proof system for proving fair termination of parallel-composed programs is presented. Modular termination proof systems that have been suggested before ([OG76], [ABO90] and [Apt83]) are defined for models with an unfair scheduler. Those proof systems are based on proving the separate termination of each of the components and showing that those proofs are interference-free [OG76, ABO90] or cooperating [Apt83]. This approach fails to be complete in a model with a fair scheduler since two non-terminating programs in this model can be composed into a terminating one. In fact, all
combinations are possible, i.e., there exist examples in which some components terminate and some do not and the composed programs either terminate or do not.

The main idea suggested here which allows for the development of a modular and complete fair termination proof system is a new program property, called *gapped-termination*. A program \( P \) is gapped-terminating w.r.t. a set of programs with their preconditions, \( \mathcal{P} \), if the parallel composition of \( P \) with any program in \( \mathcal{P} \) is a terminating program w.r.t. the corresponding precondition. The gapped-termination specification of a program, \( P \), formally captures the contribution of \( P \) to the progress towards termination.

The proposed proof system is *modular* but not *compositional*. The interpretation used here for these concepts is as follows. A proof system is *modular* if it is based on a separate verification of the components of the program. It is *compositional* as well if only specifications of the components are manipulated to deduce a specification of the composed program. When additional reference to the components is needed the system is not compositional but it is still considered *modular*.

In the compositional proof systems suggested in [MC81, ZBR85, St88] compositionality is achieved by using an *assume-guarantee* scheme in which assumptions on the environment are explicitly presented in the specification of a component.

The approach presented here is an *implicit* assume-guarantee scheme. A specification of a component formalizes only the behavior of the given component. However, this specification induces implicit assumptions on the environment. Leaving the environment explicitly unspecified results in assumptions on the environment which are the weakest possible, given the component's specification. Weak assumptions are essential when a program with its specification are to be used in different applications. Here, a specification of a composed program is derived by applying additional consistency tests on the components and their specifications.

A different approach for overcoming the need for weak assumption on the environment when developing a program in a bottom up manner is suggested in [Zw88, Ra90]. They suggest the use of *adaptation* rules [Ho71] to maintain a proof system which is, according to their terminology, *modular complete*. A proof system is modular complete if, whenever given specifications of the components semantically imply the correctness of a specification, \( S \), of the composed program, the specification \( S \) can be deduced in the proof system from the specifications of the components, treating the components as black boxes.

2. A program composition operator, called *jigsaw*, is considered. The jigsaw composition was first presented in [FFG90]. The interaction among the components defined by this operator is more complex than the one defined by parallel composition. The jigsaw composition can be added to an arbitrary programming language provided that the language is extended to allow *gapped programs*. A gapped program is a program in the language under consideration which includes scattered gaps. The jigsaw operator composes gapped programs by "filling" gaps in a component with statements taken from other components. The definition of the jigsaw operator in [FFG90] is more restrictive than the one suggested here. In [FFG90] only the semantics of a gap-free program is defined; therefore, every gap of a gapped program component must be filled. This requires a high degree of syntactic matching among the components and thus the reusability of a gapped program in several different contexts is not easily achieved. Here the semantics of a gapped program is also defined. This allows for jigsaw composition of gapped programs in which some of the gaps are left unfilled. As a result, a given program \( P \) is more easily reused since it can be jigsaw-composed with a variety of components which fill different sets of gaps in \( P \). For