

Relational Verification Using Product Programs^{*}

Gilles Barthe¹, Juan Manuel Crespo¹, and César Kunz^{1,2}

¹ IMDEA Software Institute

² Universidad Politécnica de Madrid

Abstract. Relational program logics are formalisms for specifying and verifying properties about two programs or two runs of the same program. These properties range from correctness of compiler optimizations or equivalence between two implementations of an abstract data type, to properties like non-interference or determinism. Yet the current technology for relational verification remains underdeveloped. We provide a general notion of product program that supports a direct reduction of relational verification to standard verification. We illustrate the benefits of our method with selected examples, including non-interference, standard loop optimizations, and a state-of-the-art optimization for incremental computation. All examples have been verified using the Why tool.

1 Introduction

Relational reasoning provides an effective mean to understand program behavior: in particular, it allows to establish that the same program behaves similarly on two different runs, or that two programs execute in a related fashion. Prime examples of relational properties include notions of simulation and observational equivalence, and 2-properties, such as non-interference and continuity. In the former, the property considers two programs, possibly written in different languages and having different notions of states, and establishes a relationship between their execution traces, whereas in the latter only one program is considered, and the relationship considers two executions of that program.

In spite of its important role, and of the wide range of properties it covers, there is a lack of applicable program logics and tools for relational reasoning. Indeed, existing logics [4,20] are confined to reasoning about structurally equal programs, and are not implemented. This is in sharp contrast with the more traditional program logics for which robust tool support is available. Thus, one natural approach to bring relational verification to a status similar to standard verification is to devise methods that soundly transform relational verification tasks into standard ones. More specifically for specifications expressed using pre and post-conditions, one would aim at developing methods to transform Hoare quadruples of the form $\{\varphi\} c_1 \sim c_2 \{\psi\}$, where φ and ψ are relations on the states

^{*} Partially funded by European Projects FP7-231620 HATS and FP7-256980 NESSoS, Spanish project TIN2009-14599 DESAFIOS 10, Madrid Regional project S2009TIC-1465 PROMETIDOS. C. Kunz is funded by a Juan de la Cierva Fellowship, MICINN, Spain.

of the command c_1 and the states of the command c_2 , into Hoare triples of the form $\{\bar{\varphi}\} c \{\bar{\psi}\}$, where $\bar{\varphi}$ and $\bar{\psi}$ are predicates on the states of the command c , and such that the validity of the Hoare triple entails the validity of the original Hoare quadruple; using \models to denote validity, the goal is to find c , $\bar{\varphi}$ and $\bar{\psi}$ s.t.

$$\models \{\bar{\varphi}\} c \{\bar{\psi}\} \quad \Rightarrow \quad \models \{\varphi\} c_1 \sim c_2 \{\psi\}$$

Consider two simple imperative programs c_1 and c_2 and assume that they are separable, i.e. operate on disjoint variables. Then we can let assertions be first-order formulae over the variables of the two programs, and achieve the desired effect by setting $c \equiv c_1; c_2$, $\bar{\varphi} \equiv \varphi$ and $\bar{\psi} \equiv \psi$. This method, coined self-composition by Barthe, D’Argenio and Rezk [2], is sound and relatively complete, but it is also impractical [19]. In a recent article, Zaks and Pnueli [21] develop another construction, called cross-product, that performs execution of c_1 and c_2 in lock-step and use it for translation validation [22], a general method for proving the correctness of compiler optimizations. Cross-products, when they exist, meet the required property; however their existence is confined to structurally equivalent programs and hence they cannot be used to validate loop optimizations that modify the control flow of programs, nor to reason about 2-properties such as non-interference and continuity, because such properties consider runs of the program that do not follow the same control flow.

The challenge addressed in this paper is to provide a general notion of product programs which allows transforming relational verification tasks into standard ones, without the setbacks of cross-products or self-composition. In our setting, a product between two programs c_1 and c_2 is a program c which combines synchronous steps, in which instructions from c_1 and c_2 are executed in lockstep, with asynchronous steps, in which instructions from c_1 or c_2 are executed separately. Products combine the best of cross-products and self-composition: the ability of performing asynchronous steps recovers the flexibility and generality of self-composition, and make them applicable to programs with different control structures, whereas the ability of performing synchronous steps is the key to make the verification of c as effective as the verification of cross-products and significantly easier than the verification of the programs obtained by self-composition. Concretely, we demonstrate how product programs can be combined with off-the-shelf verification tools to carry relational reasoning on a wide range of examples, including: various forms of loop optimizations, static caching for incremental computation, SSE transformations for increasing performance on multi-core platforms, information flow and continuity analyses. All examples have been formally verified using the Why framework with its SMT back-end; in one case, involving complex summations on arrays, we used a combination of the SMT back-end and the Coq proof assistant back-end—however it is conceivable that the proof obligations could be discharged automatically by declaring suitable axioms in the SMT solver.

Contents. The paper is organized as follows: Section 2 introduces the product construction and shows the need for a generalization of cross-product and self-composition. Section 3 defines product programs and shows how they enable