Trends in Relational Program Verification

Bernhard Beckert and Mattias Ulbrich

1 Introduction

Relational program verification refers to the verification of relational properties, which relate different programs, different versions of the same program, or the same program for different inputs. For relational verification several program runs need to be compared. In this chapter, we give an overview of current trends in relational program verification.

There are many interesting program properties that are relational in nature. The most prominent examples are variants of program equivalence: Two (different) programs are equivalent if they terminate in equivalent program states whenever started equivalent states. Another important kind of a relational property is non-interference: If it is provable that any two runs of a system that differ in the initial value of some variable x result in the same output, then consequently the variable x does not interfere with the output (the system does not reveal information about the initial value of x).

Recently, there is a growing interest in relational properties, and relational verification is an important trend in formal methods. One of the main reasons for this trend is that relational properties avoid the bottleneck of having to write complex requirement specifications which impedes the use of formal verification in practice. Instead, the programs that are compared serve as specification of each other. For example, one may prove that – except for intended changes – a new program version behaves as the old one (regression verification).

Even though, intuitively, relating several program runs seems to be a more complex task than just investigating single runs, relational program verification is in many cases easier than functional verification. As explained in the course of this

Bernhard Beckert
Karlsruhe Institute of Technology, Germany, e-mail: beckert@kit.edu

Mattias Ulbrich
Karlsruhe Institute of Technology, Germany, e-mail: ulbrich@kit.edu

chapter, this is in particular true if the programs resp. program runs that are being compared are similar to each other, i.e., if they run (nearly) in lockstep, produce similar results, and/or are structurally similar.

For functional verification, the effort grows with the size and complexity of the program to be verified (and its specification), while for relational program verification, the effort mainly depends on the size of the *difference* between the programs resp. program executions (and the complexity of the relational property). Thus, if the difference is small, even large and complex programs can be handled. One can exploit the fact that differences are often local and only affect a small portion of a program.

Nevertheless, relational verification is not a trivial task. It may still require complex auxiliary specifications that describe the functionality of sub-components or detail the relation between the two systems (coupling invariants).

Relational properties are a very general concept. In this chapter, we focus on scenarios

1. where the programs being compared are written in the same language – as opposed to verifying translation relations where one of the programs is an executable specification, written in a abstract language and the other is the concrete system –, and
2. where concrete programs are proved to satisfy a relational property – as opposed to proving correctness for program transformations or program generation or compilation in general.

Structure of this chapter.

First, in Section 2, we introduce and define the basic concepts of relational properties and relational verification. Then, in Section 3, we describe the main scenarios where relational program verification is employed to ensure dependability of systems, including regression verification and proving non-interference properties. In Section 4, we discuss recent trends in how to use deductive verification to prove relational properties. Finally, we draw some conclusions in Section 5.

2 Basic Notions and Definitions

The following definitions are independent of a particular programming language. We assume a set $\mathscr{P}$ of programs is given. To simplify the presentation, we assume that all programs operate on the same state space $\mathscr{S}$. The semantics of a program $P \in \mathscr{P}$ is a relation $\overset{P}{\to} \subset \mathscr{S} \times \mathscr{S}$, where $s \overset{P}{\to} s'$ means that the program P, when started in state s, may terminate in state s'. If the programming language is deterministic, the relations $\overset{P}{\to}$ are partial functions on $\mathscr{S}$. The concrete structure of states depends the declarations in the programs and on the programming language; in particular, they may contain local variables, stacks, heaps etc. The value of a program variable x in a state $s \in \mathscr{S}$ is denoted by $s(x)$. We assume that there is a special state $\mathsf{err} \in \mathscr{S}$ to indicate a program has failed an assertion.

In the following, we define and discuss different types of functional and relational properties.

Definition 1 (Functional safety property). A *functional safety property* F is a set $F \subseteq (\mathscr{S} \times \mathscr{S})$, i.e., a set of state pairs.

A program $P \in \mathscr{P}$ *satisfies* F iff, for all $s, s' \in \mathscr{S}$, $s \xrightarrow{P} s'$ implies $(s, s') \in F$, i.e., $\xrightarrow{P} \subseteq F$.

Intuitively, a functional safety property F is the set of all those state pairs s, s' such that it is a correct program behaviour for P to terminate in state s' when started in s.

Example 1. The functional property that a program must either decrease the value of the variable x by 1 when started in a state with $x > 0$ or not terminate at all, can be formalised as

$$\{\, (s, s') \mid \text{if } \; s(x) > 0 \; \text{ then } \; s'(x) = s(x) + 1 \} \enspace .$$

Note that this functional property does not place any restrictions on how a program affects other parts of the state besides the variable x.

A functional property includes those state transitions that are considered “good” or “admissible” by the property. A program is judged against the property for every state transition separately. In contrast to that, a relational property sets transitions into relation. Satisfaction of a relational property by programs P_1, P_2 is judged by considering each of the transitions in $\xrightarrow{P_1}$ in the context spanned by the state transitions of $\xrightarrow{P_2}$ and vice versa.

Definition 2 (Relational safety property). A *relational safety property* R is a set $R \subseteq (\mathscr{S} \times \mathscr{S}) \times (\mathscr{S} \times \mathscr{S})$, i.e., a relation on state pairs.

Two programs $P_1, P_2 \in \mathscr{P}$ *satisfy* R iff, for all $s_1, s'_1, s_2, s'_2 \in \mathscr{S}$:

$$\text{if } \; s_1 \xrightarrow{P_1} s'_1 \; \text{ and } \; s_2 \xrightarrow{P_2} s'_2 \; \text{ then } \; \big((s_1, s'_1), (s_2, s'_2)\big) \in R \enspace .$$

Intuitively, a relational property R consists of those combinations of state transitions (s_i, s'_i) that – by definition of that property – are allowed to “co-exist” in the semantics of the two programs.

Example 2. One of the simplest but also most often used relational properties is program equivalence. If the two programs are started in the same initial state, then they terminate in the same state (if they terminate at all); if they are started in different states, their terminal state is not restricted:

$$F_{equiv} = \Big\{ \big((s_1, s'_1), \, (s_2, s'_2)\big) \;\Big|\; \text{if } \; s_1 = s_2 \; \text{ then } \; s'_1 = s'_2 \Big\}$$

Example 3. Another prominent relational property is non-interference (see Section 3.1.1). The requirement that the input variable h does not interfere with the output variable l can be expressed as (assuming that h, l are the only variables in the state):

$$F_{\textit{non-interference}} = \big\{ \big((s_1, s_1'), (s_2, s_2')\big) \;\big|\; \text{if } \; s_1(l) = s_2(l) \; \text{ then } \; s_1'(l) = s_2'(l) \big\}$$

Relational properties allow two dimensions of variation: transitions of the same program may be compared for *different initial states* or transitions of *different programs* may be compared for the same initial state; or both dimensions of variation may be combined. The case $P_1 = P_2$, where P_1 and P_2 are the same program, is a special case that is frequently considered in practice. Such *single-program relational properties*, which are also called *2-properties*, including non-interference, are discussed in Section 3.1.

The concept of relational properties is stronger and more expressive than functional properties. In fact, every functional property can also be represented as a relational property: For a functional property F, the relational property $R = F \times F$ is satisfied by a program $P = P_1 = P_2$ (Def. 2) iff F is satisfied by P (Def. 1).

The notion of relational safety properties as given in Def. 2 does not cover all interesting relational properties. Only those properties can be expressed that can be checked by looking at two state transitions at a time. However, there are many properties that require to compare three or more transitions – of the same program or of different programs.

Example 4. Consider a program *best* that chooses the “best” element from a set X according to some heuristic. Even if we may not want to or be able to specify which element is the best one in any situation, we may require *consistency of choice*: If a set X is split into two overlapping subsets X_1, X_2 (i.e., $X = X_1 \cup X_2$), and the program chooses the same element x from X_1 and from X_2 (i.e., $x = \textit{best}(X_1) = \textit{best}(X_2)$), then it must choose x from X as well ($x = \textit{best}(X)$).

Properties such as consistency, which can (only) be defined by the comparison of three transitions, are called 3-properties. This concept can be extended to k-properties for $k \in \mathbb{N}$ by generalising Definition 2.

However, k-properties do still not cover all interesting properties. For example, termination for all initial states is a rather simple property that is not a k-property for any k. Termination is a liveness property and is existential in nature, while all k-properties are universal in nature, requiring that *all* k-tuples of state transitions are “good” in some sense.

Example 5. The relational property that, whenever P_1 terminates when started in the initial state s, the program P_2 also terminates when started in s can be formalised as:

$$\text{for all } s, s_1' \in \mathscr{S} \text{ with } s \xrightarrow{P_1} s_1' \text{ there is an } s_2' \in \mathscr{S} \text{ with } s \xrightarrow{P_2} s_2'$$

As explained above, this property (called *mutual* or *relational termination*) cannot be expressed as a relational safety property. This parallels the functional case: Termination of a single program can also not be formulated as a (functional) safety property.

The taxonomy of properties can be further extended to more complex combinations of universal and existential quantification, though such complex property types are rare in practice. Clarkson et al (2008) introduce a taxonomy of relational properties, including *k*-safety and *k*-liveness properties, considering a program semantics with traces instead of pre-/post-state pairs, which is useful for reactive systems, process algebras, etc.

So far, we have discussed the concepts of relational properties and when they are satisfied by programs. For relational verification, i.e., to prove that programs satisfy a relational property, we need a further concept, namely *coupling properties*. A coupling property is a relation on states, which intuitively holds for corresponding states during the execution of two (or more) related programs. For example, if two programs run in lockstep (i.e., their traces are the same), then identity is a trivial coupling property. Coupling properties that hold throughout program execution are also called a *coupling invariant*. Typically coupling invariants do not need to hold in all states of a trace but only at certain synchronisation points.

A *coupling predicate* is a formula (in some logic used for deductive verification) that expresses a coupling property. Coupling predicates that are inductive, i.e., for which it is possible to prove by induction on the length of the traces that the property holds throughout the execution, are a powerful tool for verifying relational properties (see Section 4).

3 Application Scenarios

In this section, we describe the main scenarios where relational program verification is employed to ensure dependability of systems, including regression verification and proving non-interference properties.

3.1 Single-Program Properties

Single-program properties relate runs of the same program for different inputs. Typically, they require that, if the inputs are in some relation to each other, then the outputs must be in some (other) relation.

3.1.1 Non-interference and Information-flow Properties

For complex programs, one cannot easily tell how the input data is processed and how it affects the final state of the programs and flows into output variables. Secure information flow requires that the (public) output of a program does not depend on its (secret) inputs – resp. only to a certain degree. This is a relational safety property (Def. 2), which is also called *non-interference* as the secret input must not interfere with the public output. The non-interference property relates different runs of a program: For any two runs starting in states that only differ in the secret part of the initial state (the public part may be different), the observable (public) part of the final state must be the same (see Example 3). If a program is non-interferent, an attacker cannot learn the secret input by observing the public output – where we assume that the attacker knows the source code of the program and can control the public input.

Joshi and Leino (2000) and Amtoft and Banerjee (2004) were the first to give semantical definitions of information flow based on relational properties. A full definition of information flow in terms of the input-output function's equivalence kernel can be found, e.g., in Klebanov (2014). The survey paper by Sabelfeld and Myers (2003) gives an overview of language-based information-flow analyses.

While deductive verification of non-interference properties is a rather new development, there have been static security-enforcing techniques based on syntax or types for a long time. Type systems and program-dependency-graph-based analyses (e.g., by Hammer and Snelting (2009)) allow analysis of larger programs, but are less precise than deductive methods.

3.1.2 Symmetry Properties

An important kind of relational properties are symmetry properties, which express that, if two initial states are symmetric (or in some other way similar) to each other, they lead to symmetric (similar) final states.

If the number of possible inputs for a program is large, the effort for both testing and formal verification can be greatly reduced if the state space can be partitioned using a relational symmetry property. To exploit a symmetry relation S for verifying (or testing) a program P w.r.t. a functional property F (compatible with S), it suffices

1. to show that P has the relational symmetry property S,
2. to show that P satisfies F for a small subset $X \subseteq \mathscr{S}$ of representatives, and
3. to show that X reaches all states in $\mathscr{S}$ via symmetry S.

From this it can be concluded that P satisfies the property F for all $s \in \mathscr{S}$.

Example 6. A typical symmetry property is permutation-invariance S_{perm}. Assume for this example that all states are arrays of length N (i.e., $\mathscr{S} = \mathbb{N}^N$). A program P is called permutation-invariant if its result is the same for an array a and for a permuted array $\sigma(a)$ (for some permutation $\sigma \in \mathrm{Sym}_N$):

$$S_{perm} = \left\{ \left((s_1, s_1'), (s_2, s_2')\right) \,\middle|\, \text{if } \; s_1 = \sigma(s_2) \; \text{ for some } \sigma \in \mathrm{Sym}_N, \; \text{ then } \; s_1' = s_2' \right\}$$

The typical set X_{perm} of representatives is the set of sorted arrays. Hence, if a program P is permutation-invariant, it suffices to prove a functional property for sorted arrays only, which reduces the search space considerably.

For the example of verifying voting rules, the use of symmetry properties is discussed extensively by Beckert et al (2016). Voting rules are highly symmetric algorithms for fairness reasons; for example, the election result must be symmetric w.r.t. the order of voters and the order of candidates. There is also related work on breaking symmetries on the problem-specification level (e.g., by Mancini and Cadoli (2005); Cadoli and Mancini (2005)).

3.2 Multi-Program Properties

Multi-program relational properties compare two or more programs by their observable behaviour. Typically, the properties considered in practice are a variant of program equivalence. The programs may be required to be fully equivalent, or some relaxed version of equivalence may be used that allows for exceptions or replaces identity of results with similarity (e.g., using isomorphism of states).

Multi-program properties are not fundamentally different from single-program properties. In fact, Beckert et al (2015) show how the verification of a multi-program property can be reduced to verification of a single-program property by combining the input programs into a single program that comprises the possible behaviours of all original programs. While this reduction is theoretically possible, it is inefficient in practice.

3.2.1 Regression Verification

One of the main concerns during software evolution is to prevent the introduction of unwanted behaviour, commonly known as *regressions,* when implementing new features, fixing defects, or during optimisation. Undetected regressions can have severe consequences and incur high cost, in particular in late stages of development. Currently, the main quality assurance measure during software evolution is regression testing (Ammann and Offutt, 2008). Regression verification – a notion coined by Godlin and Strichman (2013) – is a complementary approach that attempts to achieve the same goals with program verification techniques. This means formally verifying that the two programs satisfy a relational equivalence property. In the basic form of regression verification, we try to prove that the two program versions terminate in identical final states for any initial state. In more sophisticated scenarios, we want to verify that the two programs (a) are equivalent only for some initial states, namely those not affected by the evolution step (conditional equivalence), or (b) differ in a formally specified way given by a relation on the final states that is different from the identity (relational equivalence). If regression verification is successful, it offers guaranteed coverage without requiring additional expenses to develop and maintain a test suite.

Interestingly, regression verification can be applied to information-flow properties. The goal then is proving that the new program version does not leak more secrets than the old one. This leads to a two-program 4-property; one of the rare cases of relational k-properties with $k > 3$ that are of practical interest.

3.2.2 Translation Validation

As said in the introduction, relational program verification focuses on scenarios where concrete programs are proved to satisfy a relational property. Ideally, one would like to prove correctness of a program transformation in general, i.e., to conduct a universal proof that implies correctness of the transformation for all programs. Then, no proof would be required for individual instances of the transformation.

But in many cases the general proof is too complex and impractical. Then, translation validation is a useful alternative, where we show for a concrete application of the program transformation that its result is equivalent to the original program. During the translation, useful information (like coupling predicates, information about applied loop unwinding, etc.) can be gathered (as witnesses) to aid the relational verification process. An example for this approach was presented by Lopes et al (2015) for proving correctness of optimisations in llvm code.

3.2.3 Contextual Equivalence

Contextual equivalence (Murawski et al, 2015), also called *backward compatibility* (Welsch and Poetzsch-Heffter, 2014), is a relational property of (sub-)programs requiring that they behave equivalently when included into any possible program context. This property, which is important for, e.g., evolution or refactoring of library functions, is an extension of basic program equivalence. Instead of (only) requiring two programs P_1, P_2 to be equivalent in the sense that

$$s \xrightarrow{P_1} s' \text{ iff } s \xrightarrow{P_2} s' \quad \text{for all } s, s' \in \mathscr{S} \quad ,$$

contextual equivalence requires that

$$s \xrightarrow{Q[P_1]} s' \text{ iff } s \xrightarrow{Q[P_2]} s' \quad \text{for all programs } Q[\cdot] \text{ and } s, s' \in \mathscr{S} \quad ,$$

where $Q[P_i]$ is the result of inserting the program P_i as a sub-program into $Q[\cdot]$.

Relational program verification to prove contextual equivalence requires an adequate semantic model for open programs. In the object-oriented setting, the program logic used for verification must account for features such as inheritance and callbacks. Welsch and Poetzsch-Heffter (2014, 2011) provide a solution for the context of Java and Boogie; Murawski et al (2015) show how the difficulties can be dealt with using game semantics.

3.2.4 Refinement

Refinement is a relational property between two behavioural descriptions, where usually one is more abstract and one is more concrete. Since this chapter focuses on relational program verification, we concentrate on the program verification aspects of refinement proofs. In algorithmic refinement, an abstract imperative algorithm description A is refined into a more detailed concrete description C, e.g., by defining how abstract concepts are turned into actual data structures. A distinguishing trait of refinement is that the algorithms may be non-deterministic: In the abstract program, the algorithm may have choices (like $\texttt{choose}\ x\ \texttt{such that}\ x > 0$) which are deliberately left open on the abstract level, whereas a refinement step may concretise the statement (e.g., into $x := 5$).

Refinement is not a relational safety property according to Def. 2 since it cannot be expressed using only universal quantification. As we have seen, program equivalence for deterministic programs, which requires that two programs started in the same state terminate in the same state (if they both terminate), can be formalised with universal quantification using the relational safety property F_{equiv} (see Example 2):

$$\text{for all } s_1, s_2, s_1', s_2' \in \mathscr{S}\text{: if } s_1 \xrightarrow{C} s_1' \text{ and } s_2 \xrightarrow{A} s_2'\ , \text{ then } \big((s_1, s_1'), (s_2, s_2')\big) \in F_{equiv}$$

In the presence of non-deterministic behaviour, however, this property scheme is not adequate: The abstract program A may have several different terminal states, and they cannot be all equal to the terminal state(s) of C. Instead, it must be required that s_1' is a *possible* terminal state for P_2 using a different property scheme

$$\text{for all } s_1, s_1' \in \mathscr{S} \text{ exist } s_2, s_2' \in \mathscr{S} \text{ such that:}$$
$$\text{if } s_1 \xrightarrow{C} s_1',\ \text{ then } s_2 \xrightarrow{A} s_2' \text{ and } \big((s_1, s_1'), (s_2, s_2')\big) \in R$$

for some refinement relation $R \subseteq (\mathscr{S} \times \mathscr{S}) \times (\mathscr{S} \times \mathscr{S})$. If the input and output coupling predicate for this refinement step is the identity (like for equivalence), then the refinement relation $R_{equiv} = \big\{\big((s,s'),(s,s')\big) \bigm| s, s' \in \mathscr{S}\big\}$ must be used. Interestingly, mutual termination (see Example 5) falls also into this property scheme using the relation $R_{term} = \big\{\big((s,s_1),(s,s_2)\big) \bigm| s, s_1, s_2 \in \mathscr{S}\big\}$.

Ulbrich (2013) uses a dynamic logic to formulate and discharge refinement proof obligations. Dynamic logic (Harel et al, 2001) is a program logic like Hoare logic. But it is more general as it supports nesting of statements and supports “forall” and “exists” terminal state operators.

4 Verification of Relational Properties

Two trends in research for the verification of relational properties can be observed that lift existing successes in program verification to relational questions:

1. *Reduction of relational properties to functional safety properties that can be verified using off-the-shelf functional program verification tools.*
   This makes recent and future technological advances in functional verification automatically available for relational verification, too. Issues like modularisation, loop-handling, framing, etc. that are similar for functional and relational verification can be left to the existing functional program verification machinery.
2. *Exploiting similarities between intermediate states in the compared program runs.*
   Examining the program runs to be compared individually has serious disadvantages in many cases. For many practical application scenarios for relational verification, the program runs are related in the sense that intermediate synchronisation points in the compared programs can be identified at which the states of their runs are similar. This can ease the verification burden considerably.

These trends are not dependent on each other. But it can be observed that often they both play a rote in relational verification methods.

4.1 Reduction to Functional Verification

Early approaches to verifying relational properties devised specialised logics for handling relational questions. Benton (2004) first introduced the theory of a relational Hoare calculus to reason about relational properties, and Yang (2007) introduced a relational separation logic with special syntactic extensions that allow the specification of separating conjunctions on two heaps instead of only one.

The seminal paper by Barthe et al (2011) introduced a general notion of *product programs* that supports a direct reduction of relational verification to standard functional verification. Product programs are not tied to a particular application scenario. To make use of single-program functional verification tools for relational two-program verification, functional verification is applied to a product program constructed from the two programs to be compared. For the construction, it is important that the two programs cannot interfere with each other. We hence assume that the two programs P_1 and P_2 operate on disjoint sets of variables. In case this assumption does not hold, e.g., for the analysis of a single-program relational property where the programs are identical, disjointness can easily achieved by variable renaming. Syntactically, a product program P operates on both the variables (heaps, stacks, memories, etc.) of P_1 and those of P_2 and consists of the statements of both P_1 and P_2. Semantically, a product program operates on the state space $\mathscr{S} \times \mathscr{S}$ and satisfies for all $s_1, s_1', s_2, s_2' \in \mathscr{S}$:

$$\text{if} \quad s_1 \xrightarrow{P_1} s_1' \text{ and } s_2 \xrightarrow{P_2} s_2' \qquad \text{then} \qquad (s_1, s_2) \xrightarrow{P} (s_1', s_2') \text{ or } (s_1, s_2) \xrightarrow{P} \mathsf{err} \ ,$$

i.e., the result state of P consists of the result states s_1 and s_2 that would arise if P_1 and P_2 were run separately – or P may terminate in the special error state err, i.e., fail an assertion.

The concession that a product program is allowed to fail in more cases than the original programs allows us to combine programs into product programs more liberally. Assumptions about intermediate states or the synchronisation of P_1 and P_2 can be added to P using assertions in the code of P. During a proof for P, these assertions are to be proved correct, thus justifying the assumptions made during the construction of P. We will encounter product programs with additional assertions in Section 4.2.

Product programs pave the way for using functional program calculi to formalise and discharge relational properties. Instead of a special relational Hoare logic and calculus using quadruples $\{\phi\}P_1 \sim P_2\{\psi\}$ to talk about two programs (Benton, 2004), the traditional Hoare triple $\{\phi\}P\{\psi\}$ can be analysed using a standard Hoare calculus:

$$\models \{\phi\}P\{\psi\} \quad \text{implies} \quad \models \{\phi\}P_1 \sim P_2\{\psi\} \tag{1}$$

The challenge now is to find good principles for the construction of product programs. Most obviously, the sequential composition $(P_1\,;P_2)$ is a valid product program. In fact, the "implies" in implication (1) turns into an "iff" for sequential composition. Accordingly, to formulate the non-interference property for a program P as a functional proof obligation, Barthe et al (2004) and Darvas et al (2005) suggested to employ *self-composition*, i.e., to sequentially compose two copies of P.

However, this simple direct sequential composition has substantial drawbacks, as already reported by Terauchi and Aiken (2005) soon after introduction of the technique of self-composition. The problem can be visualised by the following thought experiment:

Example 7. Take a (non-trivial) deterministic program P operating on a single variable x that consists of a single while-loop. In order to verify that P is equivalent to a clone P_c (with the same code, but the variable is renamed to x_c), one can try to prove the Hoare triple $\{x = x_c\}(P\,;P_c)\{x = x_c\}$ based on simple sequential composition. To deal with the loops, sufficiently strong loop invariants must be found for the two consecutive loops. It turns out that the *strongest possible* functional loop invariant that together with the negated guard is satisfied only by a single value (depending on the initial value of x) must be used. Any weaker loop invariant leaving freedom of choice for the values of x and x_c would not suffice to imply the required equality $x = x_c$ in the final state of $(P;P_c)$.

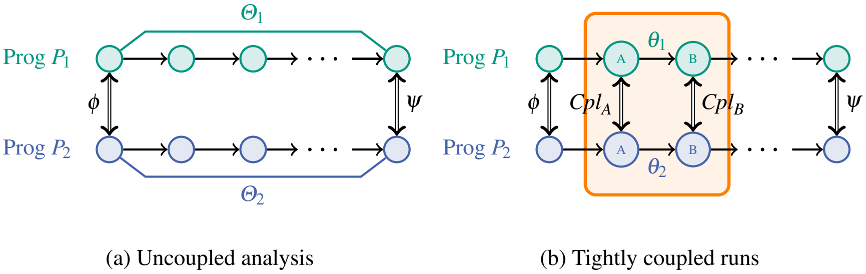
Loop invariants are difficult to find, automatically or manually. Hence, it is of vital importance to find better ways to construct product programs if one wants make the technique accessible to state-of-the-art automatic or interactive verification tools.

4.2 Exploiting Similarity between Program Runs

The key to solving the problem of sequential composition exposed in Example 7 is to exploit similarities between program states that occur in the runs that are compared during the verification. This allows to simplify the steps of the verification.

Revisiting Example 7, we see how a different execution policy makes things a lot easier: Assume the two loops were not executed consecutively but *alternatingly*, i.e., executing one iteration of the loop of P, then one of P_c, then again one of P, and so on. Then, whenever an iteration of P_c finishes, (a) both loops have iterated equally often and (b) $x = x_c$ holds. This is indeed a sufficiently strong *coupling invariant* to complete the trivial proof. Regardless of the complexity of the result that P computes, the simple coupling predicate suffices for an inductive proof.

The idea of similarity exploitation is to identify locations (e.g., line numbers) in both programs such that, when they are reached, the corresponding states of the programs are coupled. Pairs of such locations are called synchronisation points. In principle any two states can be coupled, and there is no formal definition of when two states are “similar” or “coupled”. A synchronisation is well chosen (in the context of a proof) if there is a simple enough coupling predicate that can be used in program verification to abstract the states at the synchronisation point.



(a) Uncoupled analysis

(b) Tightly coupled runs

Fig. 1: Visualisation of relational verification and trace similarities.

Figure 1 illustrates the idea of coupling states for relational verification. Figure 1a depicts the verification task $\{\phi\}(P_1\,;P_2)\{\psi\}$ for the sequential composition. The two programs must be handled separately. One needs to do verification steps to the effect of extracting functional before-after-predicates Θ_1 for P_1 and Θ_2 for P_2 (using, e.g., a weakest precondition or a Hoare calculus) and then reason that $\phi \wedge \Theta_1 \wedge \Theta_2 \to \psi$. Figure 1b shows how this can be avoided: Instead of considering a program run as one state transition, it is broken down into segments leading from synchronisation point to synchronisation point. Every segment (framed block in Fig. 1b) is verified individually. If Cpl_A, Cpl_B are coupling predicates that capture the relation between states at the synchronisation points A resp. B and θ_1, θ_2 are before-after-predicates for the transition from synchronisation point A to point B in the respective programs, then the verification condition for the segment is $Cpl_A \wedge \theta_1 \wedge \theta_2 \rightarrow Cpl_B$. The validity of the entire proof condition then follows inductively, in very much the same way as loop invariant proofs for functional programs.

A challenge here is to identify good locations at which to set synchronisation points. Most natural candidates are those points where functional program verification also applies abstraction in form of loop invariants: the entry points of loops (loop heads).

```
while cnd1 do
  body1
end;
while cnd2 do
  body2
end
```

(a) Sequential composition

```
assert cnd1 == cnd2 ;
while cnd1 do
  body1;
  body2;
  assert cnd1 == cnd2
end
```

(b) perfect synchronisation

```

while cnd1 or cnd2 do
  if cnd1 then body1;
  if cnd2 then body2

end
```

(c) loose synchronisation

Fig. 2: Different product programs for two loops.

Technically, the coupling can be implemented in different ways. One possibility is to produce product programs in which the loops are iterated alternatingly. Fig. 2 shows how programs in a simple while programming language (with side-effect-free expressions) can be woven into a product program. Three possibly ways to weave two loops are presented: Fig. 2a is the direct sequential composition which does not make use of synchronisation points. In Fig. 2b, the loop bodies are iterated alternatingly and always in pairs. Verification using this product program requires that the loops iterate equally often because otherwise the assertion that the loop conditions `cnd1` and `cnd2` evaluate equivalently will fail. Finally, Fig. 2c shows how the limitation of equal number of iterations can be lifted by adding conditionals that check the loop guards individually. This product loop first executes the loop bodies alternatingly and then, when one program has terminated the loop, finishes the remaining loop in an uncoupled fashion.

It is noteworthy that the verification of the perfect synchronisation scheme from Fig. 2b also implies mutual termination (see Example 5) of the two loops. The added assertion ensures that both loops iterate equally often for any input, which is sufficient for mutual termination (assuming the loop bodies have no further loops). Hence, although mutual termination is not a relational k-property, sufficient conditions can be encoded into product programs that imply the property. Elenbogen et al (2015) and Hawblitzel et al (2013) formulate more sophisticated relational safety properties that entail mutual termination.

Relational proofs for fully synchronised loops can already be conducted using the early relational calculi presented by Benton (2004) and Yang (2007).

4.3 More Elaborate Synchronisation Schemes

Verification by means of coupling predicates works well in many situations – but only if the two programs under inspection reach related states whenever they reach a synchronisation point. If that premise does not hold, e.g., when comparing two completely different sorting algorithms for equivalence (say, to show that the result of mergesort is the same as that of bubblesort), looking for good synchronisation points would be futile since the inner structure of the algorithms is so different that specifying the relationship between states would be more difficult than verifying that both are stable sorting functions.

The strict lockstep combination in which exactly one loop iteration of the first program corresponds to exactly one loop iteration of the second program is, in many cases, not flexible enough to account for all typical synchronization scenarios. It may very well be that for one program, a loop needs to first be unwound a number of times before entering a lockstep synchronous part; or each loop iteration of one program does not correspond to one but to k iterations of the other program; and, of course, many other synchronisation schemes are possible. Fig. 3 shows a schematic example of loosely coupled loops in which program states are not in a 1:1 relationship.

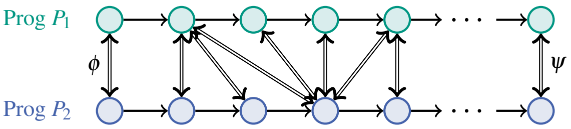


Fig. 3: Loosely coupled runs.

When dealing with relational program verification by reduction to functional verification, a flexible synchronisation mechanism can be achieved by defining a set of transformation rules that can be used to weave two programs into a single program, which is a product program by construction. This has originally been suggested by Barthe et al (2011) for weaving two programs, a generalisation (that supports several programs and modular verification) has been presented by Eilers et al (2018). Banerjee et al (2016) present a relational Hoare logic supporting flexible program weaving without necessarily producing a product program in the course.

Relational verification has a high potential for automation: Regardless of the complexity of the actual computation, the compared program runs in a relational verification task can be very closely related. For instance, in a regression verification scenario, two programs may be identical but for the assignments to a single variable x. In many cases, it is enough to specify relationally how the variable x is treated differently and assume that nothing else has changed.

If the coupling predicates between corresponding loop iterations is close to equality, it is indeed a good candidate to not have it specified by the user but to infer it automatically using inference techniques for loop invariants. Felsing et al (2014) have shown that relational coupling predicates can be inferred automatically for regression verification of non-trivial programs. The approach supports loosely coupled synchronization points and uses dynamic exploration to find loop unwinding ratios that promise a good synchronisation between programs (Kiefer et al, 2018). In this approach, coupling predicates are modelled as uninterpreted predicates within constrained Horn clauses (Rümmer et al, 2013). Modern SMT solvers are used to infer sufficiently strong inductive predicates such that the relational post-condition is satisfied.

4.4 Alternative Approaches

While in the previous sections, we have presented a general technique to deal with relational properties in deductive verification, it should be mentioned that product program construction (or similar techniques relating symbolic program executions) is not the only way to deal with relational properties deductively. For single-program properties, it is sometimes possible to formulate verification conditions that only require a *single* invocation of the program. However, this requires that program constructs are embedded into the logic more deeply, in particular that programs can occur in the scope of quantifiers. In Hoare logic, formulas like $\forall x\,\{\phi\}P\{\psi\}$ are syntactically not allowed. But, in dynamic logic (Harel et al, 2001), such a quantification is possible.

Darvas et al (2005) noted that non-interference of a (secret) variable h with a (public) variable l for a program P can be expressed using a single program invocation of P in dynamic logic:

$$\forall l \exists r \forall h\ \langle P \rangle r = l$$

Because of the order of quantifiers in this formula, it expresses that, for all input values of l, there is a *single* result value r for the public variable l that is independent of the input value for h.

5 Conclusions

Relational properties are ubiquitous, and there are many application scenarios in which they play a rule. The effort for relational program verification mainly grows with the size and complexity of the difference between the compared programs resp. program runs. As the size of the difference is a different dimension than size and complexity of the programs themselves, relational verification has the potential to be usable in cases where functional verification is infeasible. In particular, the need for writing requirement specifications can often be avoided. Thus, relational verification can extend the reach of formal methods to new application scenarios.

While – on a theoretical level – the expressiveness of relational properties is the same as that of functional properties, experience shows that some new techniques are needed in practice for relational verification, which moreover differ for different use cases. In particular, the heuristics needed to automatically find synchronisation points and coupling invariants depend on the application scenario.

References

Ammann P, Offutt J (2008) Introduction to software testing. Cambridge University Press

Amtoft T, Banerjee A (2004) Information flow analysis in logical form. In: Giacobazzi R (ed) Static Analysis, 11th International Symposium, SAS 2004, Verona, Italy, August 26-28, 2004, Proceedings, Springer, Lecture Notes in Computer Science, vol 3148, pp 100–115, DOI 10.1007/978-3-540-27864-1_10

Banerjee A, Naumann DA, Nikouei M (2016) Relational logic with framing and hypotheses. In: Lal A, Akshay S, Saurabh S, Sen S (eds) 36th IARCS Annual Conference on Foundations of Software Technology and Theoretical Computer Science, FSTTCS 2016, December 13-15, 2016, Chennai, India, Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, LIPIcs, vol 65, pp 11:1–11:16, DOI 10.4230/LIPIcs.FSTTCS.2016.11

Barthe G, D'Argenio PR, Rezk T (2004) Secure information flow by self-composition. In: 17th IEEE Computer Security Foundations Workshop, (CSFW-17 2004), 28-30 June 2004, Pacific Grove, CA, USA, IEEE Computer Society, pp 100–114, DOI 10.1109/CSFW.2004.17

Barthe G, Crespo JM, Kunz C (2011) Relational verification using product programs. In: Butler MJ, Schulte W (eds) FM 2011: Formal Methods - 17th International Symposium on Formal Methods, Limerick, Ireland, June 20-24, 2011. Proceedings, Springer, Lecture Notes in Computer Science, vol 6664, pp 200–214, DOI 10.1007/978-3-642-21437-0_17

Beckert B, Klebanov V, Ulbrich M (2015) Regression verification for Java using a secure information flow calculus. In: Monahan R (ed) Proceedings of the 17th Workshop on Formal Techniques for Java-like Programs, FTfJP 2015, Prague, Czech Republic, July 7, 2015, ACM, pp 6:1–6:6, DOI 10.1145/2786536.2786544

Beckert B, Bormer T, Kirsten M, Neuber T, Ulbrich M (2016) Automated verification for functional and relational properties of voting rules. In: Sixth International Workshop on Computational Social Choice (COMSOC 2016)

Benton N (2004) Simple relational correctness proofs for static analyses and program transformations. In: Jones ND, Leroy X (eds) Proceedings of the 31st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2004, Venice, Italy, January 14-16, 2004, ACM, pp 14–25, DOI 10.1145/964001.964003

Cadoli M, Mancini T (2005) Using a theorem prover for reasoning on constraint problems. In: AI* IA 2005: Advances in Artificial Intelligence, Springer

Clarkson MR, Chong S, Myers AC (2008) Civitas: Toward a secure voting system. In: 2008 IEEE Symposium on Security and Privacy (S&P 2008), 18-21 May 2008, Oakland, California, USA, IEEE Computer Society, pp 354–368, DOI 10.1109/SP.2008.32

Darvas Á, Hähnle R, Sands D (2005) A theorem proving approach to analysis of secure information flow. In: Hutter D, Ullmann M (eds) Security in Pervasive Computing, Second International Conference, SPC 2005, Boppard, Germany, April 6-8, 2005, Proceedings, Springer, Lecture Notes in Computer Science, vol 3450, pp 193–209, DOI 10.1007/978-3-540-32004-3_20

Eilers M, Müller P, Hitz S (2018) Modular product programs. In: Ahmed A (ed) Programming Languages and Systems - 27th European Symposium on Programming, ESOP 2018, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2018, Thessaloniki, Greece, April 14-20, 2018, Proceedings, Springer, Lecture Notes in Computer Science, vol 10801, pp 502–529, DOI 10.1007/978-3-319-89884-1_18

Elenbogen D, Katz S, Strichman O (2015) Proving mutual termination. Formal Methods in System Design 47(2):204–229, DOI 10.1007/s10703-015-0234-3

Felsing D, Grebing S, Klebanov V, Rümmer P, Ulbrich M (2014) Automating regression verification. In: Crnkovic I, Chechik M, Grünbacher P (eds) ACM/IEEE International Conference on Automated Software Engineering, ASE '14, Vasteras, Sweden - September 15 - 19, 2014, ACM, pp 349–360, DOI 10.1145/2642937.2642987

Godlin B, Strichman O (2013) Regression verification: proving the equivalence of similar programs. Softw Test, Verif Reliab 23(3):241–258, DOI 10.1002/stvr.1472

Hammer C, Snelting G (2009) Flow-sensitive, context-sensitive, and object-sensitive information flow control based on program dependence graphs. Int J Inf Sec 8(6):399–422, DOI 10.1007/s10207-009-0086-1

Harel D, Kozen D, Tiuryn J (2001) Dynamic logic. SIGACT News 32(1):66–69, DOI 10.1145/568438.568456

Hawblitzel C, Kawaguchi M, Lahiri SK, Rebêlo H (2013) Towards modularly comparing programs using automated theorem provers. In: Bonacina MP (ed) Automated Deduction - CADE-24 - 24th International Conference on Automated Deduction, Lake Placid, NY, USA, June 9-14, 2013. Proceedings, Springer, Lecture Notes in Computer Science, vol 7898, pp 282–299, DOI 10.1007/978-3-642-38574-2_20

Joshi R, Leino KRM (2000) A semantic approach to secure information flow. Sci Comput Program 37(1-3):113–138, DOI 10.1016/S0167-6423(99)00024-6

Kiefer M, Klebanov V, Ulbrich M (2018) Relational program reasoning using compiler IR - combining static verification and dynamic analysis. J Autom Reasoning 60(3):337–363, DOI 10.1007/s10817-017-9433-5

Klebanov V (2014) Precise quantitative information flow analysis - a symbolic approach. Theor Comput Sci 538:124–139, DOI 10.1016/j.tcs.2014.04.022

Lopes NP, Menendez D, Nagarakatte S, Regehr J (2015) Provably correct peephole optimizations with alive. In: Grove D, Blackburn S (eds) Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation, Portland, OR, USA, June 15-17, 2015, ACM, pp 22–32, DOI 10.1145/2737924.2737965

Mancini T, Cadoli M (2005) Detecting and breaking symmetries by reasoning on problem specifications. In: Abstraction, Reformulation and Approximation, Springer

Murawski AS, Ramsay SJ, Tzevelekos N (2015) Game semantic analysis of equivalence in IMJ. In: Finkbeiner B, Pu G, Zhang L (eds) Automated Technology for Verification and Analysis - 13th International Symposium, ATVA 2015, Shanghai, China, October 12-15, 2015, Proceedings, Springer, Lecture Notes in Computer Science, vol 9364, pp 411–428, DOI 10.1007/978-3-319-24953-7_30

Rümmer P, Hojjat H, Kuncak V (2013) Classifying and solving Horn clauses for verification. In: Cohen E, Rybalchenko A (eds) Verified Software: Theories, Tools, Experiments - 5th International Conference, VSTTE 2013, Menlo Park, CA, USA, May 17-19, 2013, Revised Selected Papers, Springer, Lecture Notes in Computer Science, vol 8164, pp 1–21, DOI 10.1007/978-3-642-54108-7_1

Sabelfeld A, Myers AC (2003) Language-based information-flow security. IEEE Journal on Selected Areas in Communications 21(1):5–19, DOI 10.1109/JSAC.2002.806121

Terauchi T, Aiken A (2005) Secure information flow as a safety problem. In: Hankin C, Siveroni I (eds) Static Analysis, 12th International Symposium, SAS 2005, London, UK, September 7-9, 2005, Proceedings, Springer, Lecture Notes in Computer Science, vol 3672, pp 352–367, DOI 10.1007/11547662_24

Ulbrich M (2013) Dynamic logic for an intermediate language: Verification, interaction and refinement. PhD thesis, Karlsruhe Institute of Technology

Welsch Y, Poetzsch-Heffter A (2011) Full abstraction at package boundaries of object-oriented languages. In: da Silva Simão A, Morgan C (eds) Formal Methods, Foundations and Applications - 14th Brazilian Symposium, SBMF 2011, São Paulo, Brazil, September 26-30, 2011, Revised Selected Papers, Springer, Lecture Notes in Computer Science, vol 7021, pp 28–43, DOI 10.1007/978-3-642-25032-3_3

Welsch Y, Poetzsch-Heffter A (2014) A fully abstract trace-based semantics for reasoning about backward compatibility of class libraries. Sci Comput Program 92:129–161, DOI 10.1016/j.scico.2013.10.002

Yang H (2007) Relational separation logic. Theor Comput Sci 375(1-3):308–334, DOI 10.1016/j.tcs.2006.12.036