An Efficient VCGen-based Modular Verification of Relational Properties

Lionel Blatter[1,2](0000-0001-9058-2005), Nikolai Kosmatov[3,4](0000-0003-1557-2813), Virgile Prevosto[3](0000-0002-7203-0968), and Pascale Le Gall[5](0000-0002-8955-6835)

[1] Karlsruhe Institute of Technology, 76131, Karlsruhe, Germany
`firstname.lastname@kit.edu`
[2] Max Planck Institute for Security and Privacy, 44799, Bochum, Germany
[3] Université Paris-Saclay, CEA, List, 91120, Palaiseau, France
`firstname.lastname@cea.fr`
[4] Thales Research & Technology, 91120, Palaiseau, France
[5] CentraleSupélec, Université Paris-Saclay, 91190 Gif-sur-Yvette France
`firstname.lastname@centralesupelec.fr`

Abstract Deductive verification typically relies on function contracts that specify the behavior of each function for a single function call. *Relational properties* link several function calls together within a single specification. They can express more advanced properties of a given function or relate calls to different functions, possibly run in parallel. However, relational properties cannot be expressed and verified directly in the traditional setting of modular deductive verification. Recent work proposed a new technique for relational property verification that relies on a verification condition generator to produce logical formulas that must be verified to ensure a given relational property. This paper presents an overview of this approach and proposes important enhancements. We integrate an optimized verification condition generator and extend the underlying theory to show how relational properties can be proved in a modular way, where one relational property can be used to prove another one, like in modular verification of function contracts. Our results have been fully formalized and proved sound in the Coq proof assistant.

1 Introduction

Modular deductive verification [19] is used to prove that every function f of a given program respects its *contract*. Such a contract is, basically, an implication: if the given *precondition* is true before a call to f and the call terminates[6], the given *postcondition* is true when f returns control to the caller. However, some kinds of properties are not easily reducible to a single function call. Indeed, it is often necessary to express a property that involves several functions, possibly executed in parallel, or relates the results of several calls to the same function for different arguments. Such properties are known as *relational properties* [6].

[6] Termination can be either assumed (partial correctness) or proved separately (full correctness) in a classical way [16]; for the purpose of this paper we can assume it.

```
//Command c_sum:
if x1 < x2 then {
    x3 := x3 + x1;
    x1 := x1 + 1;
    call(y_sum)
} else { skip }
```

Relational property $\mathcal{R}_1$ between commands c^1_ω and c^2_ω:

$$\{\, x_2\langle 1\rangle = x_2\langle 2\rangle \,\}\ \begin{array}{l} //\ c^1_\omega\text{:} \\ x_1 := 1; \\ x_3 := 0; \\ \mathbf{call}(y_{\text{sum}}) \end{array}\ \langle 1\rangle \sim \begin{array}{l} //\ c^2_\omega\text{:} \\ x_1 := 0; \\ x_3 := 0; \\ \mathbf{call}(y_{\text{sum}}) \end{array}\ \langle 2\rangle\ \{\, x_3\langle 1\rangle = x_3\langle 2\rangle \,\}$$

Figure 1: Recursive command c_{sum}, associated as a body with procedure name y_{sum}, and relational property $\mathcal{R}_1$ between two commands, denoted c^1_ω and c^2_ω, involving a call to this procedure.

Examples of such relational properties include monotonicity (i.e. $x \leq y \Rightarrow f(x) \leq f(y)$), involving 2 calls, or transitivity ($\text{cmp}(x, y) \geq 0 \wedge \text{cmp}(y, z) \geq 0 \Rightarrow \text{cmp}(x, z) \geq 0$), involving 3 calls. In secure information flow [3], *non-interference* is also a relational property. Namely, given a partition of program variables between high-security variables and low-security variables, a program is said to be non-interferent if any two executions starting from states in which the low-security variables have the same initial values will end up in a final state where the low-security variables have the same values. In other words, high-security variables cannot interfere with low-security ones.

Motivation. Lack of support for relational properties in verification tools was already faced by industrial users (e.g. in [8] for C programs). The usual way to deal with this limitation is to use *self-composition* [3,30,9], product programs [2] or other self-composition variants [31]. Those techniques are based on code transformations that are relatively tedious and error-prone. Moreover, they are hardly applicable in practice to real-life programs with pointers like in C. Namely, self-composition requires that the compared executions operate on completely separated (i.e. disjoint) memory areas, which might be extremely difficult to ensure for complex programs with pointers. Modular verification of relational properties is another important feature: the user may want to rely on some relational properties in order to verify some other ones.

Example 1 (relational property). Figure 1 shows an example of a recursive command (that is, program) c_{sum}. We clearly distinguish the name and the body of a procedure. The procedure named y_{sum} is assumed to have command c_{sum} as its body, so that c_{sum} recursively calls itself. Given three global integer variables x_1, x_2 and x_3, command c_{sum} adds to x_3 (used as an accumulator) the sum $x_1 + (x_1 + 1) + \cdots + (x_2 - 1)$ if $x_1 < x_2$, and has no effect otherwise.

Figure 1 also shows an example of a relational property $\mathcal{R}_1$ (inspired by [2]) stating the equivalence of two commands c^1_ω and c^2_ω (assumed to be run on separate memory states), which assign x_1 and x_3 before calling y_{sum}. The relational property is written here in Benton's notation [6]: tags $\langle 1\rangle$ and $\langle 2\rangle$ are used to distinguish the programs linked by the property. When variables of the linked programs have the same names, such a tag after a variable name also helps to distinguish the instance of the variable used in the relational precondition and postcondition (written in curly braces, resp., on the left and on the right). Property $\mathcal{R}_1$ states that if x_2 has the same value before the execution of c^1_ω and before the execution of c_ω^2, then x_3 will have the same value after their executions. Indeed, c_ω^1 will compute in x_3 the sum $1+2+\cdots+(x_2-1)$, while c_ω^2 will compute in x_3 the sum $0+1+2+\cdots+(x_2-1)$.

In this paper, we show how relational property $\mathcal{R}_1$ can be verified using another relational property $\mathcal{R}_3$ linking two runs of c_{sum} rather than using a full functional contract of c_{sum}. More precisely, $\mathcal{R}_3$ (that will be formally defined below in Fig. 5) generalizes the situation of $\mathcal{R}_1$ and states that the resulting value of x_3 after two runs of c_{sum} will be the same if the initial state of the second run is exactly one iteration of c_{sum} behind that of the first run. □

Approach. Our recent work [11] proposed an alternative to self-composition that is not based on code transformation or relational rules. It directly relies on a standard verification condition generator (VCGen) to produce logical formulas to be verified (typically, with an automated prover) to ensure a given relational property. This approach requires no extra code processing (such as sequential composition of programs or variable renaming). Moreover, no additional separation hypotheses are required. The locations of each program are separated by construction: each program has its own memory state. This approach has been formalized on a minimal language L, representative of the main issues relevant for relational property verification. L is a standard While language extended with annotations, procedures and pointers. Notably, the presence of dereferences and address-of operations makes it representative of various aliasing problems with (possibly, multiple) pointer dereferences of a real-life language like C. An example of a relational property for programs with pointers was given in [11]. We formalize the proposed approach and prove its soundness in the Coq proof assistant [33]. Our Coq development[7] contains about 3700 lines.

Contributions. We give an overview of the VCGen-based approach for relational property verification (presented in [11]) and enhance the underlying theory with several new features. The new technical contributions of this paper include:

– a Coq formalization and proof of soundness of an optimized VCGen for language L, and its extension to the verification of relational properties;
– an extension of the framework allowing not only to *prove* relational properties, but also to *use* them as hypotheses in the following proofs;
– a Coq formalization of the extended theory.

We also provide an illustrative example and, as another minor extension, add the capacity to refer to old values of variables in postconditions.

Outline. Section 2 introduces the imperative language L used in this work. Functional correctness is defined in Section 3. The extension of functional correctness to relational properties is presented in Section 4. Then, we prove the soundness of an optimized VCGen in Section 5, and show how it can be soundly extended to verify relational properties in Section 6. Finally, we present related work in Section 7 and concluding remarks in Section 8.

[7] Available at `https://github.com/lyonel2017/Relational-Spec/`.

2 Syntax and Semantics of the Considered Language L

2.1 Locations, States, and Procedure Contracts

We denote by $\mathbb{N} = \{0, 1, 2, \dots\}$ the set of natural numbers, by $\mathbb{N}^* = \{1, 2, \dots\}$ the set of nonzero natural numbers, and by $\mathbb{B} = \{\text{True}, \text{False}\}$ the set of Boolean values. Let $\mathbb{X}$ be the set of program *locations* and $\mathbb{Y}$ the set of *program (procedure) names*, and let $x, x', x_1, \dots$ and $y, y', y_1, \dots$ denote metavariables ranging over those respective sets. We assume that there exists a bijective function $\mathbb{N} \to \mathbb{X}$, so that $\mathbb{X} = \{x_i \,|\, i \in \mathbb{N}\}$. Intuitively, we can see i as the *address* of location x_i.

Let Σ be the set of functions $\sigma : \mathbb{N} \to \mathbb{N}$, called *memory states*, and let $\sigma, \sigma', \sigma_1, \dots$ denote metavariables ranging over Σ. A state σ maps a location to a value using its address: location x_i has value $\sigma(i)$.

We define the *update* operation of a memory state $set(\sigma, i, n)$, also denoted by $\sigma[i/n]$, as the memory state σ' mapping each address to the same value as σ, except for i, bound to n. Formally, $set(\sigma, i, n)$ is defined by the following rules:

$$\forall \sigma \in \Sigma, x_i \in \mathbb{X}, n \in \mathbb{N}, x_j \in \mathbb{X}.\ i = j \Rightarrow \sigma[i/n](j) = n, \tag{1}$$

$$\forall \sigma \in \Sigma, x_i \in \mathbb{X}, n \in \mathbb{N}, x_j \in \mathbb{X}.\ i \neq j \Rightarrow \sigma[i/n](j) = \sigma(j). \tag{2}$$

Let Ψ be the set of functions $\psi : \mathbb{Y} \to \mathbb{C}$, called *procedure environments*, mapping program names to commands (defined below), and let $\psi, \psi_1, \dots$ denote metavariables ranging over Ψ. We write $\text{body}_\psi(y)$ to refer to $\psi(y)$, the commands (or *body*) of procedure y in a given procedure environment ψ. An example of a procedure environment ψ_{sum} is given in Fig. 5, where $\text{body}_{\psi_{\text{sum}}}(y_{\text{sum}}) = c_{\text{sum}}$.

Preconditions (or *assertions*) are predicates of arity one, taking as parameter a memory state and returning an equational first-order logic formula. Let metavariables $P, P_1, \dots$ range over the set $\mathbb{P}$ of preconditions. For instance, using λ-notation, precondition P assessing that location x_3 is bound to 2 can be defined by $P \triangleq \lambda\sigma.\sigma(3) = 2$. This form will be more convenient for relational properties (than e.g. $x_3 = 2$) as it makes explicit the memory states on which a property is evaluated.

Postconditions are predicates of arity two, taking as parameters two memory states and returning an equational first-order logic formula. Its two arguments refer to the initial and the final state. For instance, postcondition Q assessing that location x_1 was incremented (that is, $x_1 = \text{old}(x_1) + 1$) can be defined in λ-notation by $Q \triangleq \lambda\sigma\sigma'.\,\sigma'(1) = \sigma(1) + 1$. Let metavariables $Q, Q_2, \dots$ range over the set $\mathbb{Q}$ of postconditions.

Finally, we define the set Φ of *contract environments* $\phi : \mathbb{Y} \to \mathbb{P} \times \mathbb{Q}$, and metavariables $\phi, \phi_1, \dots$ to range over Φ. More precisely, ϕ maps a procedure name y to the associated (procedure) *contract* $\phi(y) = (\text{pre}_\phi(y), \text{post}_\phi(y))$, composed of a pre- and a postcondition for procedure y. As usual, a procedure contract will allow us to specify the behavior of a single call to the corresponding procedure, that is, if we start executing y in a memory state satisfying $\text{pre}_\phi(y)$, and the evaluation terminates, the pair composed of the initial and final states will satisfy $\text{post}_\phi(y)$.

$a ::= n$	natural const.
$\mid x$	location
$\mid *x$	dereference
$\mid \&x$	address
$\mid a_1 \ op_a \ a_2$	arithm. oper.

$b ::= true \mid false$	Boolean const.
$\mid a_1 \ op_b \ a_2$	comparison
$\mid b_1 \ op_l \ b_2 \mid \neg b_1$	logic oper.

$c ::= \textbf{skip}$	do nothing
$\mid x := a$	direct assignment
$\mid *x := a$	indirect assignment
$\mid c_1; c_2$	sequence
$\mid \textbf{assert}(P)$	assertion
$\mid \textbf{if } b \textbf{ then } \{c_1\} \textbf{ else } \{c_2\}$	condition
$\mid \textbf{while } b \textbf{ inv } P \textbf{ do } \{c_1\}$	loop
$\mid \textbf{call}(y)$	procedure call

Figure 2: Syntax of arithmetic and Boolean expressions and commands in L.

$$\xi_a[\![n]\!]\sigma \triangleq n \qquad \xi_a[\![x_i]\!]\sigma \triangleq \sigma(i) \qquad \xi_a[\![*x_i]\!]\sigma \triangleq \sigma(\sigma(i)) \qquad \xi_a[\![\&x_i]\!]\sigma \triangleq i$$

Figure 3: Evaluation of expressions in L (selected rules).

2.2 Syntax for Expressions and Commands

Let $\mathbb{E}_a$, $\mathbb{E}_b$ and $\mathbb{C}$ denote respectively the sets of arithmetic expressions, Boolean expressions and commands. We denote by $a, a_1, \ldots; b, b_1, \ldots$ and $c, c_1, \ldots$ metavariables ranging, respectively, over those sets. Syntax of arithmetic and Boolean expressions is given in Fig. 2. Constants are natural numbers or Boolean values. Expressions use standard arithmetic, comparison and logic binary operators, denoted respectively $op_a ::= \{+, \times, -\}$, $op_b ::= \{\leqslant, =, \ldots\}$, $op_l ::= \{\vee, \wedge\}$. Since we use natural values, the subtraction is bounded by 0, as in Coq: if $n' > n$, the result of $n - n'$ is considered to be 0. Expressions also include locations, possibly with a dereference or an address operator.

Figure 2 also presents the syntax of commands in L. Sequences, skip and conditions are standard. An assignment can be done to a location directly or after a dereference. Recall that a location x_i contains as a value a natural number, say v, that can be seen in turn as the address of a location, namely x_v, so the assignment $*x_i := a$ writes the value of expression a to the location x_v, while the address operation $\&x_i$ computes the address i of x_i. An assertion command **assert**(P) indicates that an assertion P should be valid at the point where the command occurs. The loop command **while** b **inv** P **do** $\{c_1\}$ is always annotated with an invariant P. As usual, this invariant should hold when we reach the command and be preserved by each loop step. Command **call**(y) is a procedure call. All annotations (assertions, loop invariants and procedure contracts) will be ignored during the program execution and will be relevant only for program verification in Section 5. Procedures do not have explicit parameters and return values (hence we use the term *procedure call* rather than *function call*). Instead, as in assembly code [23], parameters and return value(s) are shared implicitly between the caller and the callee through memory locations: the caller must put/read the right values at the right locations before/after the call. Finally, to avoid ambiguity, we group sequences of commands with { }.

$$\langle \mathbf{assert}(P), \sigma\rangle \xrightarrow{\psi} \sigma \qquad \frac{\xi_a[\![a]\!]\sigma = n}{\langle x_i := a, \sigma\rangle \xrightarrow{\psi} \sigma[i/n]} \qquad \frac{\xi_a[\![a]\!]\sigma = n}{\langle *x_i := a, \sigma\rangle \xrightarrow{\psi} \sigma[\sigma(i)/n]} \qquad \frac{\langle \mathrm{body}_\psi(y), \sigma_1\rangle \xrightarrow{\psi} \sigma_2}{\langle \mathbf{call}(y), \sigma_1\rangle \xrightarrow{\psi} \sigma_2}$$

Figure 4: Operational semantics of commands in L (selected rules).

Procedure environment: $\psi_{\mathrm{sum}} \triangleq \{y_{\mathrm{sum}} \to c_{\mathrm{sum}}\}$

Hoare triple $\mathcal{R}_2$: $\psi_{\mathrm{sum}} : \big\{\,\mathrm{True}\big\}\, c_{\mathrm{sum}}\, \big\{\,\mathrm{old}(x_1) \geqslant \mathrm{old}(x_2) \Rightarrow \mathrm{old}(x_3) = x_3\big\}$

Relational property $\mathcal{R}_3$:

$$\psi_{\mathrm{sum}} : \left\{\begin{array}{l} x_1\langle 2\rangle < x_2\langle 2\rangle \ \wedge \\ x_2\langle 1\rangle = x_2\langle 2\rangle \ \wedge \\ x_1\langle 1\rangle = x_1\langle 2\rangle + 1 \ \wedge \\ x_3\langle 1\rangle = x_3\langle 2\rangle + x_1\langle 2\rangle \end{array}\right\} c_{\mathrm{sum}}\langle 1\rangle \sim c_{\mathrm{sum}}\langle 2\rangle \,\big\{\, x_3\langle 1\rangle = x_3\langle 2\rangle \,\big\}$$

Figure 5: A procedure environment ψ_{sum} associating procedure name y_{sum} with its body c_{sum} (see Fig. 1), a Hoare triple $\mathcal{R}_2$ for command c_{sum}, and a relational property $\mathcal{R}_3$ linking two runs of c_{sum}.

2.3 Operational Semantics

Evaluation of arithmetic and Boolean expressions in L is defined by functions ξ_a and ξ_b. Selected evaluation rules for arithmetic expressions are shown in Fig. 3. Operations $*x_i$ and $\&x_i$ have a semantics similar to the C language, i.e. dereferencing and address-of. Semantics of Boolean expressions is standard [36].

Based on these evaluation functions, we can define the operational semantics of commands in a given procedure environment ψ. Selected evaluation rules[8] are shown in Fig. 4. As said above, both assertions and loop invariants can be seen as program annotations that do not influence the execution of the program itself. Hence, command $\mathbf{assert}(P)$ is equivalent to a skip. Likewise, loop invariant P has no influence on the semantics of **while** b **inv** P **do** $\{c\}$.

We write $\Vdash \langle c, \sigma\rangle \xrightarrow{\psi} \sigma'$ to denote that $\langle c, \sigma\rangle \xrightarrow{\psi} \sigma'$ can be derived from the rules of Fig. 4. Our Coq formalization, inspired by [29], provides a deep embedding of L, with an associated parser, in files `Aexp.v`, `Bexp.v` and `Com.v`.

3 Functional Correctness

We define functional correctness in a similar way to the original *Hoare triple* definition [19], except that we also need a procedure environment ψ, leading to a quadruple denoted $\psi : \{P\}c\{Q\}$. We will however still refer by the term "Hoare triple" to the corresponding program property, formally defined as follows.

Definition 1 (Hoare triple). *Let c be a command, ψ a procedure environment, and P and Q two assertions. We define a Hoare triple $\psi : \{P\}c\{Q\}$ as follows:*

$$\psi : \{P\}c\{Q\} \triangleq \forall \sigma, \sigma' \in \Sigma.\ P(\sigma) \wedge (\Vdash \langle c, \sigma\rangle \xrightarrow{\psi} \sigma') \Rightarrow Q(\sigma, \sigma').$$

Informally, our definition states that, for a given ψ, if a state σ satisfies P and the execution of c on σ terminates in a state σ', then (σ, σ') satisfies Q.

[8] Full versions of Fig. 3, 4 are given in Appendix A.

Example 2. Figure 5 gives an example of a Hoare triple denoted $\mathcal{R}_2$. □

Next, we introduce notation $CV(\psi, \phi)$ to denote the fact that, for the given ψ and ϕ every procedure satisfies its contract.

Definition 2 (Contract Validity). *Let ψ be a procedure environment and ϕ a contract environment. We define contract validity $CV(\psi, \phi)$ as follows:*

$$CV(\psi, \phi) \triangleq \forall y \in \mathbb{Y}.\ \psi : \{\mathrm{pre}_\phi(y)\}\boldsymbol{call}(y)\{\mathrm{post}_\phi(y)\}).$$

The notion of contract validity is at the heart of modular verification, since it allows assuming that the contracts of the callees are satisfied during the verification of a Hoare triple. More precisely, to state the validity of procedure contracts without assuming anything about their bodies in our formalization, we will consider an arbitrary choice of implementations ψ' of procedures that satisfy the contracts, like in the first assumption of Theorem 1 below. This theorem, taken from [1, Th. 4.2] and reformulated for L in [11], states that $\psi : \{P\}c\{Q\}$ holds if we can prove the contract of (the bodies in ψ of) all procedures in an arbitrary environment ψ' respecting the contracts, and if the validity of contracts of ϕ for ψ implies the Hoare triple itself. This theorem is the basis for modular verification of Hoare Triples, as done for instance in Hoare Logic [19,36] or verification condition generation.

Theorem 1 (Recursion). *Given a procedure environment ψ and a contract environment ϕ such that the following two assumptions hold:*

$$\forall \psi' \in \Psi.\ CV(\psi', \phi) \Rightarrow \forall y \in \mathbb{Y}, \psi' : \{\mathrm{pre}_\phi(y)\}\mathrm{body}_\psi(y)\{\mathrm{post}_\phi(y)\},$$
$$CV(\psi, \phi) \Rightarrow \psi : \{P\}c\{Q\},$$

we have $\psi : \{P\}c\{Q\}$.

We refer the reader to the Coq development, more precisely the results `recursive_proc` and `recursive_hoare_triple` in file `Hoare_Triple.v` for a complete proof of Theorem 1.

4 Relational Functional Correctness

Relational properties can be seen as an extension of Hoare triples. But, instead of linking one program with two properties, the pre- and postconditions, relational properties link n programs to two properties, called *relational precondition* and *relational postcondition*. A *relational precondition* or *assertion* (resp., *relational postcondition*) for n programs is a predicate taking a sequence of n (resp., $2n$) memory states and returning a first-order logic formula. Metavariables $\widehat{P}$, $\widehat{P'}, \dots$ (resp., $\widehat{Q}$, $\widehat{Q'}$, ...) range over the corresponding sets. As a simple example, the relational postcondition of $\mathcal{R}_1$ (written in Fig. 1 in Benton's notation) can be stated in λ-notation as follows: $\lambda\sigma_1, \sigma_2, \sigma'_1, \sigma'_2\ .\ \sigma'_1(3) = \sigma'_2(3)$.

A *relational property* is a property about n programs $c_1, \dots, c_n$, stating that if each program c_i starts in a state σ_i and ends in a state σ'_i such that $\widehat{P}(\sigma_1, \dots, \sigma_n)$

holds, then $\widehat{Q}(\sigma_1, \ldots, \sigma_n, \sigma'_1, \ldots, \sigma'_n)$ holds, where $\widehat{P}$ is a relational precondition and $\widehat{Q}$ is a relational postcondition. We formally define relational correctness similarly to functional correctness (cf. Def. 1), except that we now use sequences of commands and memory states. We abbreviate by $(u_k)^n$ a sequence of elements $(u_k)_{k=1}^n = (u_1, \ldots, u_n)$, where k ranges from 1 to n. If $n \leq 0$, $(u_k)^n$ is the empty sequence denoted [ ]. If $n = 1$, $(u)^1$ is the singleton sequence (u).

Definition 3 (Relational Hoare Triple). *Let ψ be a procedure environment, $(c_k)^n$ a sequence of n commands ($n \in \mathbb{N}^*$), $\widehat{P}$ and $\widehat{Q}$ relational pre- and postcondition for n commands. The relational correctness of $(c_k)^n$ with respect to $\widehat{P}$ and $\widehat{Q}$, denoted $\psi : \{\widehat{P}\}(c_k)^n\{\widehat{Q}\}$, is defined as follows:*

$$\psi : \{\widehat{P}\}(c_k)^n\{\widehat{Q}\} \triangleq$$
$$\forall (\sigma_k)^n, (\sigma'_k)^n.\ \widehat{P}((\sigma_k)^n) \wedge (\bigwedge_{i=1}^{n} \Vdash \langle c_i, \sigma_i \rangle \xrightarrow{\psi} {\sigma'}_i) \Rightarrow \widehat{Q}((\sigma_k)^n, (\sigma'_k)^n).$$

For $n = 1$, this notion defines a Hoare triple. It also generalizes Benton's notation [6] for two commands: $\psi : \{\widehat{P}\}c_1 \sim c_2\{\widehat{Q}\}$. As Benton's work mostly focused on comparing equivalent programs, using symbol $\sim$ was quite natural.

Example 3. Relational property $\mathcal{R}_3$ introduced in Ex. 1 is formalized (in Benton's notation) in Fig. 5. Below, we will illustrate modular verification of relational properties by deducing $\mathcal{R}_1$ from $\mathcal{R}_3$ and partial contract $\mathcal{R}_2$ of c_{sum}. □

We will now extend Theorem 1 to relational contract environments. A *relational contract environment* $\widehat{\phi}$ maps a sequence of program names $(y_k)^n$ to a *relational contract*, composed of a relational pre- and postcondition, denoted $\widehat{\phi}((y_k)^n) = (\,\widehat{\text{pre}}_{\widehat{\phi}}((y_k)^n), \widehat{\text{post}}_{\widehat{\phi}}((y_k)^n)\,)$. Practical applications require only a finite number of properties, so the relational contract can be assumed trivial for all except a finite number of sequences. A relational contract environment generalizes a contract environment, since a standard procedure contract is a relational contract (for a sequence of exactly one element). Notice that $\widehat{\phi}$ considers only one relational property for a given sequence $(y_k)^n$: this is not a limitation since several properties can be encoded in one contract. We define the set of relational contract environments $\widehat{\Phi}$, and metavariables $\widehat{\phi}, \widehat{\phi}_0, \widehat{\phi}_1, \ldots$ will range over $\widehat{\Phi}$.

We introduce notation $CV_r(\psi, \widehat{\phi})$ to denote the fact that all procedures defined in ψ satisfy the relational contracts in which they are involved in $\widehat{\phi}$.

Definition 4 (Relational Contract Validity). *Let ψ be a procedure environment and $\widehat{\phi}$ a relational contract environment. We define $CV_r(\psi, \widehat{\phi})$ as follows:*

$$CV_r(\psi, \widehat{\phi}) \triangleq \forall (y_k)^n \in \text{dom}(\widehat{\phi}),\ n > 0 \Rightarrow \psi : \{\widehat{\text{pre}}_{\widehat{\phi}}((y_k)^n)\}(\textbf{\textit{call}}(y_k))_{k=1}^n\{\widehat{\text{post}}_{\widehat{\phi}}((y_k)^n)\}.$$

Theorem 2 (Relational Recursion). *Given a procedure environment ψ and a relational contract environment $\widehat{\phi}$ such that the following two assumptions hold:*

$$\forall \psi' \in \Psi.\ CV_r(\psi', \widehat{\phi}) \Rightarrow$$
$$\forall (y_k)^n \in \text{dom}(\widehat{\phi}), \psi' : \{\widehat{\text{pre}}_{\widehat{\phi}}((y_k)^n)\}(\text{body}_{\psi}(y_k))_{k=1}^n\{\widehat{\text{post}}_{\widehat{\phi}}((y_k)^n)\},$$

$$CV_r(\psi, \widehat{\phi}) \Rightarrow \psi : \{\widehat{P}\}(c_k)^n\{\widehat{Q}\}$$

then we have $\psi : \{\widehat{P}\}(c_k)^n\{\widehat{Q}\}$.

The Coq proof (which is a straightforward extension of the proof of Theorem 1) is available in `Rela.v`, Theorem `recursion_relational`.

5 Optimized Verification Condition Generator

A standard way [16] for verifying that a Hoare triple holds is to use a verification condition generator (VCGen). In this section, we formalize a VCGen for Hoare triples such that if all verification conditions that it generates are valid, then the Hoare triple is valid according to Def. 1. The VCGen described in this section is based on optimizations introduced in [15]. Such optimizations allow the VCGen to return formulas whose size is linear with respect to the size of the program itself, and are now part of any state-of-the-art deductive verification tool. The key idea is to avoid splitting verification condition generation into two separated sub-generation at each conditional. The definition is formalized in Coq in the file `Vcg_Opt.v`, where we also prove that the verification conditions of this optimized VCGen imply those of the naive VCGen presented in [11]. This will allow us to use the optimized VCGen (or more generally any VCGen satisfying the properties stated in Theorem 3 below) for the verification of relational properties as well (see Section 6).

5.1 Verification Condition Generator

When defining the naive VCGen in [11], we proposed a modular definition. Namely, we divided it into three functions $\mathcal{T}_c$, $\mathcal{T}_a$ and $\mathcal{T}_f$. Here, we follow the same approach for the optimized VCGen, using three new functions $\mathcal{T}_c^{\triangleright}$, $\mathcal{T}_a^{\triangleright}$, and $\mathcal{T}_f^{\triangleright}$:

- function $\mathcal{T}_c^{\triangleright}$ generates the main verification condition, expressing that the postcondition holds in the final state, assuming auxiliary annotations hold;
- function $\mathcal{T}_a^{\triangleright}$ generates auxiliary verification conditions stemming from assertions, loop invariants, and preconditions of called procedures;
- finally, function $\mathcal{T}_f^{\triangleright}$ generates verification conditions for the auxiliary procedures that are called by the main program, to ensure that their bodies respect their contracts.

Definition 5 (Function $\mathcal{T}_c^{\triangleright}$ generating the main verification condition). *Given a command c, two memory states σ and σ', a contract environment ϕ, and a function f taking a formula as argument and returning a formula, function $\mathcal{T}_c^{\triangleright}$ returns a formula defined by case analysis on c as shown in Fig. 6.*

State σ represents the state before executing the command, while σ' represents the state after it. Intuitively, the argument that gets passed to f is the formula that relates σ and σ' according to c itself. Thus, if f is of the form

$$
\begin{aligned}
\mathcal{T}_c^{\triangleright}[\![\mathbf{skip}]\!](\sigma,\sigma',\phi,f) &\triangleq f(\sigma=\sigma')\\
\mathcal{T}_c^{\triangleright}[\![x_i := a]\!](\sigma,\sigma',\phi,f) &\triangleq f(\sigma' = set(\sigma,i,\xi_a[\![a]\!]\sigma))\\
\mathcal{T}_c^{\triangleright}[\![*x_i := a]\!](\sigma,\sigma',\phi,f) &\triangleq f(\sigma' = set(\sigma,\sigma(i),\xi_a[\![a]\!]\sigma))\\
\mathcal{T}_c^{\triangleright}[\![\mathbf{assert}(P)]\!](\sigma,\sigma',\phi,f) &\triangleq f(P(\sigma)\wedge\sigma=\sigma')\\
\mathcal{T}_c^{\triangleright}[\![c_0;c_1]\!](\sigma,\sigma',\phi,f) &\triangleq \forall\sigma'', \mathcal{T}_c^{\triangleright}[\![c_0]\!](\sigma,\sigma'',\phi,\lambda p_1.\\
&\qquad \mathcal{T}_c^{\triangleright}[\![c_1]\!](\sigma'',\sigma',\phi,\lambda p_2.f(p_1\wedge p_2)))\\
\mathcal{T}_c^{\triangleright}[\![\mathbf{if}\ b\ \mathbf{then}\ \{c_0\}\ \mathbf{else}\ \{c_1\}]\!](\sigma,\sigma',\phi,f) &\triangleq \mathcal{T}_c^{\triangleright}[\![c_0]\!](\sigma,\sigma',\phi,\lambda p_1.\\
&\qquad \mathcal{T}_c^{\triangleright}[\![c_1]\!](\sigma,\sigma',\phi,\lambda p_2.\\
&\qquad f((b\equiv \text{True}\Rightarrow p_1)\wedge(b\equiv\text{False}\Rightarrow p_2))))\\
\mathcal{T}_c^{\triangleright}[\![\mathbf{call}(y)]\!](\sigma,\sigma',\phi,f) &\triangleq f(\text{pre}_\phi(y)(\sigma)\wedge\text{post}_\phi(y)(\sigma,\sigma'))\\
\mathcal{T}_c^{\triangleright}[\![\mathbf{while}\ b\ \mathbf{inv}\ inv\ \mathbf{do}\ \{c\}]\!](\sigma,\sigma',\phi,f) &\triangleq f(inv\ \sigma\wedge inv\ \sigma'\wedge\neg(\xi_b[\![b]\!]\sigma'))
\end{aligned}
$$

Figure 6: Definition of function $\mathcal{T}_c^{\triangleright}$ generating the main verification condition.

$\lambda p.p \Rightarrow Q(\sigma,\sigma')$, as in Theorem 3 below, the resulting formula is a verification condition for post-condition Q to hold.

For **skip**, which does nothing, both states are identical. For assignments, σ' is simply the update of σ. An assertion introduces a hypothesis over σ but leaves it unchanged. For a sequence, a fresh memory state σ'' is introduced, and we compose the VCGen. For a conditional, if the condition evaluates to True, we select the condition from the *then* branch, and otherwise from the *else* branch. Note that, contrary to the naive VCGen, we perform a single call to f, ensuring the linearity of the formula.

The rule for calls simply assumes that before the call σ satisfies $\text{pre}_\phi(y)$ and after the call σ and σ' satisfy $\text{post}_\phi(y)$. Finally, $\mathcal{T}_c^{\triangleright}$ assumes that, for a loop, both the initial state σ and the final one σ' satisfy the loop invariant. Additionally, in σ' the loop condition evaluates to False. As for an assertion, the callee's precondition and the loop invariant are just assumed to be true; function $\mathcal{T}_a^{\triangleright}$, defined below, generates the corresponding proof obligations.

Example 4. For $c \triangleq$ **if** False **then** {**skip**} **else** $\{x_1 := 2\}$ we have:

$$\mathcal{T}_c^{\triangleright}[\![c]\!](\sigma,\sigma',\phi,\lambda p.\ p\Rightarrow\sigma'(1)=2)\equiv$$
$$(\text{False}\equiv\text{True}\Rightarrow\sigma=\sigma')\wedge(\text{False}\equiv\text{False}\Rightarrow\sigma'=set(\sigma,1,2))\Rightarrow\sigma'(1)=2.\quad\square$$

Lemma 1 establishes a relation between functions $\mathcal{T}_c^{\triangleright}$ and $\mathcal{T}_c$: the formulas generated by $\mathcal{T}_c^{\triangleright}$ imply the formulas generated by $\mathcal{T}_c$.

Lemma 1. *Given a program c, a procedure contract environment ϕ, a memory state σ and an assertion P, if we have $\forall\sigma'\in\Sigma,\ \mathcal{T}_c^{\triangleright}[\![c]\!](\sigma,\sigma',\phi,\lambda p.\,p\Rightarrow P(\sigma'))$, then we have $\mathcal{T}_c[\![c]\!](\sigma,\phi,P)$.*

Proof. By structural induction over c. $\square$

$$
\begin{aligned}
\mathcal{T}_a^{\triangleright}[\![\mathbf{skip}]\!](\sigma,\phi) &\triangleq \text{True}\\
\mathcal{T}_a^{\triangleright}[\![x := a]\!](\sigma,\phi) &\triangleq \text{True}\\
\mathcal{T}_a^{\triangleright}[\![*x := a]\!](\sigma,\phi) &\triangleq \text{True}\\
\mathcal{T}_a^{\triangleright}[\![\mathbf{assert}(P)]\!](\sigma,\phi) &\triangleq P(\sigma)\\
\mathcal{T}_a^{\triangleright}[\![c_0; c_1]\!](\sigma,\phi) &\triangleq \mathcal{T}_a^{\triangleright}[\![c_0]\!](\sigma,\phi) \wedge\\
&\quad\ \forall\sigma',\ \mathcal{T}_c^{\triangleright}[\![c_0]\!](\sigma,\, \sigma',\, \phi,\, \lambda p.\, p \Rightarrow \mathcal{T}_a^{\triangleright}[\![c_1]\!](\sigma',\phi))\\
\mathcal{T}_a^{\triangleright}[\![\mathbf{if}\ b\ \mathbf{then}\ \{c_0\}\ \mathbf{else}\ \{c_1\}]\!](\sigma,\phi) &\triangleq (\xi_b[\![b]\!]\sigma' \Rightarrow \mathcal{T}_a^{\triangleright}[\![c_0]\!](\sigma,\phi)) \wedge\\
&\quad\ (\neg(\xi_b[\![b]\!]\sigma') \Rightarrow \mathcal{T}_a^{\triangleright}[\![c_1]\!](\sigma,\phi))\\
\mathcal{T}_a^{\triangleright}[\![\mathbf{call}(y)]\!](\sigma,\phi) &\triangleq \text{pre}_{\phi}(y)(\sigma)\\
\mathcal{T}_a^{\triangleright}[\![\mathbf{while}\ b\ \mathbf{inv}\ inv\ \mathbf{do}\ \{c\}]\!](\sigma,\phi) &\triangleq inv(\sigma) \wedge\\
&\quad\ (\forall\sigma',\ inv(\sigma')\ \Rightarrow\ \xi_b[\![b]\!]\sigma' \Rightarrow \mathcal{T}_a^{\triangleright}[\![c]\!](\sigma',\phi)) \wedge\\
&\quad\ (\forall\sigma'\sigma'',\ inv(\sigma')\ \Rightarrow\ \mathcal{T}_c^{\triangleright}[\![c]\!](\sigma',\sigma'',\phi,\lambda p.\, p \Rightarrow inv(\sigma'')))
\end{aligned}
$$

Figure 7: Definition of function $\mathcal{T}_a^{\triangleright}$ generating auxiliary verification conditions.

Definition 6 (Function $\mathcal{T}_a^{\triangleright}$ generating the auxiliary verification condition). *Given a command c, a memory state σ representing the state before the command, and a contract environment ϕ, function $\mathcal{T}_a$ returns a formula defined by case analysis on c as shown in Fig. 7.*

Basically, $\mathcal{T}_a^{\triangleright}$ collects all assertions, preconditions of called procedures, as well as invariant establishment and preservation, and lifts the corresponding formulas to constraints on the initial state σ through the use of $\mathcal{T}_c^{\triangleright}$.

As for $\mathcal{T}_c^{\triangleright}$, the formulas generated by $\mathcal{T}_a^{\triangleright}$ imply those generated by $\mathcal{T}_a$.

Lemma 2. *For a given program c, a procedure contract environment ϕ, and a memory state σ, if we have* $\mathcal{T}_a^{\triangleright}[\![c]\!](\sigma,\phi)$, *then we have* $\mathcal{T}_a[\![c]\!](\sigma,\phi)$.

Proof. By structural induction over c. □

Finally, we define the function for generating the conditions for verifying that the body of each procedure defined in ψ respects its contract defined in ϕ.

Definition 7 (Function $\mathcal{T}_f^{\triangleright}$ generating the procedure verification condition). *$\mathcal{T}_f^{\triangleright}$ takes as argument two environments ψ and ϕ and returns a formula:*

$$
\begin{aligned}
\mathcal{T}_f^{\triangleright}(\phi,\psi) \triangleq \forall y,\sigma,\sigma'.\ &\text{pre}_{\phi}(y)(\sigma)\ \Rightarrow\\
&\mathcal{T}_a^{\triangleright}[\![\text{body}_{\psi}(y)]\!](\sigma,\phi) \wedge \mathcal{T}_c^{\triangleright}[\![\text{body}_{\psi}(y)]\!](\sigma,\sigma',\phi,\lambda p.p \Rightarrow \text{post}_{\phi}(y)(\sigma,\sigma')).
\end{aligned}
$$

Finally, the formulas generated by $\mathcal{T}_f^{\triangleright}$ imply those generated by $\mathcal{T}_f$.

Lemma 3. *For a given procedure environment ψ, and a procedure contract environment ϕ, if we have* $\mathcal{T}_f^{\triangleright}(\phi,\psi)$, *then we have* $\mathcal{T}_f(\phi,\psi)$.

Proof. Using Lemmas 1 and 2. □

The definition of the optimized VCGen and its link to the naive version can be found in file `Vcg_Opt.v` of the Coq development.

5.2 Hoare Triple Verification

Using the VCGen defined in Sec. 5.1, we can state the theorem establishing how a Hoare Triple can be verified. The proof can be found in file `Correct.v` of the Coq development.

Theorem 3 (Soundness of VCGen). *Assume that we have $\mathcal{T}_f^{\triangleright}(\phi, \psi)$ and*

$$\forall \sigma.\ P(\sigma) \Rightarrow \mathcal{T}_a^{\triangleright}[\![c]\!](\sigma, \phi),$$
$$\forall \sigma, \sigma'.\ P(\sigma) \Rightarrow \mathcal{T}_c^{\triangleright}[\![c]\!](\sigma,\ \sigma',\ \phi,\ \lambda p.\, p\ \Rightarrow\ Q(\sigma, \sigma')).$$

Then we have $\psi : \{P\}c\{Q\}$.

Proof. By soundness of the naive VCGen [11, Th. 3] and Lemmas 1, 2, 3. □

6 Modular Verification of Relational Properties

In this section, we propose a modular verification method for relational properties (defined in Section 4) using the optimized VCGen defined in Section 5 (or, more generally, any VCGen respecting Theorem 3). First, we define the function $\mathcal{T}_{cr}^{\triangleright}$ for the recursive call of $\mathcal{T}_c^{\triangleright}$ on a sequence of commands and memory states.

Definition 8 (Function $\mathcal{T}_{cr}^{\triangleright}$). *Given a sequence of commands $(c_k)^n$ and a sequence of memory states $(\sigma_k)^n$, a contract environment ϕ and a function f taking as argument a formula and returning a formula, function $\mathcal{T}_{cr}^{\triangleright}$ is defined by induction on n for the basis ($n = 0$) and inductive case ($n \in \mathbb{N}^*$) as follows:*

$$\mathcal{T}_{cr}^{\triangleright}([\,], [\,], [\,], \phi, f) \triangleq f(True),$$
$$\mathcal{T}_{cr}^{\triangleright}((c_k)^n, (\sigma_k)^n, (\sigma'_k)^n, \phi, f) \triangleq$$
$$\mathcal{T}_c^{\triangleright}[\![c_n]\!](\sigma_n, \sigma'_n, \phi,\ \lambda p_n.\ \mathcal{T}_{cr}^{\triangleright}((c_k)^{n-1}, (\sigma_k)^{n-1}, (\sigma'_k)^{n-1}, \phi, \lambda p_{n-1}.\ f(p_n \wedge p_{n-1}))).$$

Intuitively, like in Def. 5, the argument that gets passed to f is the formula that relates the n pre-states $(\sigma_k)^n$ to the n post-states $(\sigma'_k)^n$ when all $(c_k)^n$ are executed. Again, if f is of the form $\lambda p.p \Rightarrow \widehat{Q}((\sigma_k)^n, (\sigma'_k)^n)$, the resulting formula is a verification condition for the relational postcondition $\widehat{Q}$ to hold. More concretely, for $n = 2$, and f as above, we obtain:

$$\mathcal{T}_{cr}^{\triangleright}((c_1, c_2), (\sigma_1, \sigma_2), (\sigma'_1, \sigma'_2), \phi, \lambda p.p \Rightarrow \widehat{Q}((\sigma_1, \sigma_2), (\sigma'_1, \sigma'_2))) \equiv$$
$$\mathcal{T}_c^{\triangleright}[\![c_2]\!](\sigma_2, \sigma'_2, \phi,\ \lambda p_2.\mathcal{T}_c^{\triangleright}[\![c_1]\!](\sigma_1, \sigma'_1, \phi,\ \lambda p_1.p_2 \wedge p_1 \Rightarrow \widehat{Q}((\sigma_1, \sigma_2), (\sigma'_1, \sigma'_2)))).$$

We similarly define a notation for the auxiliary verification conditions for a sequence of n commands. Basically, this is the conjunction of the auxiliary verification conditions generated by $\mathcal{T}_a^{\triangleright}$ on each individual command.

Definition 9 (Function $\mathcal{T}_{ar}^{\triangleright}$). *Given a sequence of commands $(c_k)^n$ and a sequence of memory states $(\sigma_k)^n$, we define function $\mathcal{T}_{ar}^{\triangleright}$ as follows:*

$$\mathcal{T}_{ar}^{\triangleright}((c_k)^n, (\sigma_k)^n, \phi) \triangleq \bigwedge_{i=1}^{n} \mathcal{T}_a^{\triangleright}[\![c_i]\!](\sigma_i, \phi).$$

A standard contract over a single procedure y can be used directly whenever there is a call to y. For a relational contract over $(y_k)^n$, things are more complicated: there is not a single program point where we can apply the relational contract. Instead, we have to somehow track in the generated formulas all the calls that have been made, and to guard the application of the relational contract by a constraint stating that all the appropriate calls have indeed taken place. In order to achieve that, we start by defining a notation for the conjunction of a sequence of procedure calls and associated memory states:

Definition 10 (Functions $\mathcal{P}_{call}$ and $\mathcal{P}_{pred}$).

$$\mathcal{P}_{call}(y, \sigma, \sigma', \psi) \triangleq \Vdash \langle \boldsymbol{call}(y), \sigma\rangle \xrightarrow{\psi} \sigma',$$
$$\mathcal{P}_{pred}((y_k)^n, (\sigma_k)^n, (\sigma'_k)^n, \psi) \triangleq \bigwedge_{i=1}^{n} \mathcal{P}_{call}(y_i, \sigma_i, \sigma'_i, \psi).$$

Then, we can define function $\mathcal{T}_{pr}$ translating relational contracts into a logical formula, using $\mathcal{P}_{pred}$ to guard its application with tracked calls.

Definition 11 (Function $\mathcal{T}_{pr}$).

$$\begin{gathered}\mathcal{T}_{pr}(\widehat{\phi}, \psi) \triangleq \\ \forall (y_k)^n, (\sigma_k)^n, (\sigma'_k)^n,\ n > 0 \ \Rightarrow\ \mathcal{P}_{pred}((y_k)^n, (\sigma_k)^n, (\sigma'_k)^n, \psi)\ \Rightarrow \\ \widehat{\mathrm{pre}}_{\widehat{\phi}}((y_k)^n)(\sigma_k)^n \ \Rightarrow\ \widehat{\mathrm{post}}_{\widehat{\phi}}((y_k)^n)(\sigma_k)^n(\sigma'_k)^n.\end{gathered}$$

We now define function $\mathcal{L}$ to lift a relational procedure contract with an associated tracked call predicate and reduce it to a standard contract.

$$\mathcal{L}(\widehat{\phi}, \psi) \triangleq \lambda y.(\lambda\sigma.\widehat{\mathrm{pre}}_{\widehat{\phi}}((y)^1)(\sigma)^1, \lambda\sigma\sigma'.\widehat{\mathrm{post}}_{\widehat{\phi}}((y)^1)(\sigma)^1(\sigma')^1 \wedge \mathcal{P}_{call}(y, \sigma, \sigma', \psi)).$$

Finally, using function $\mathcal{T}_{pr}$ and $\mathcal{L}$, we can define function $\mathcal{T}_{fr}^{\triangleright}$ for generating the verification condition for verifying that the bodies of each sequence of procedures respect the relational contract defined in $\widehat{\phi}$: thanks to $\mathcal{L}$, each call instruction will result in a corresponding $\mathcal{P}_{call}$ occurrence in the generated formula, so that it will be possible to make use of the relational contracts hypotheses in $\mathcal{T}_{pr}$ when the appropriate sequences of calls occur.

Definition 12 (Function $\mathcal{T}_{fr}^{\triangleright}$).

$$\begin{gathered}\mathcal{T}_{fr}^{\triangleright}(\widehat{\phi}, \psi) \triangleq \\ \forall (y_k)^n, (\sigma_k)^n, (\sigma'_k)^n, \psi',\ \ \widehat{\mathrm{pre}}_{\widehat{\phi}}((y_k)^n) \ \Rightarrow\ \mathcal{T}_{pr}(\widehat{\phi}, \psi') \ \Rightarrow \\ \mathcal{T}_{ar}^{\triangleright}((\mathrm{body}_{\psi}(y_k))_{k=1}^n,\ (\sigma_k)^n,\ \mathcal{L}(\widehat{\phi}, \psi')) \wedge \\ \mathcal{T}_{cr}^{\triangleright}((\mathrm{body}_{\psi}(y_k))_{k=1}^n,\ (\sigma_k)^n,\ (\sigma'_k)^n,\ \mathcal{L}(\widehat{\phi}, \psi'),\ \lambda p.p \ \Rightarrow\ \widehat{\mathrm{post}}_{\widehat{\phi}}((y_k)^n))).\end{gathered}$$

Using functions $\mathcal{T}_{cr}^{\triangleright}$, $\mathcal{T}_{ar}^{\triangleright}$ and $\mathcal{T}_{fr}^{\triangleright}$, we can now give the main result of this paper, i.e. that the verification of relational properties with the VCGen is correct.

Theorem 4 (Soundness of relational VCGen). *For any sequence of commands* $(c_k)^n$*, contract environment* $\widehat{\phi}$*, procedure environment* ψ*, and relational pre- and postcondition* $\widehat{P}$ *and* $\widehat{Q}$*, if the following three properties hold:*

$$\mathcal{T}_{fr}^{\triangleright}(\widehat{\phi}, \psi), \tag{3}$$

$$\forall (\sigma_k)^n, \psi', \quad \widehat{P}((\sigma_k)^n) \wedge \mathcal{T}_{pr}(\widehat{\phi}, \psi) \;\Rightarrow\; \mathcal{T}_{ar}^{\triangleright}((c_k)^n, (\sigma_k)^n, \mathcal{L}(\widehat{\phi}, \psi')), \tag{4}$$

$$\forall (\sigma_k)^n, (\sigma'_k)^n, \psi', \quad \widehat{P}((\sigma_k)^n) \wedge \mathcal{T}_{pr}(\widehat{\phi}, \psi) \;\Rightarrow$$
$$\mathcal{T}_{cr}^{\triangleright}((c_k)^n, (\sigma_k)^n, (\sigma'_k)^n, \mathcal{L}(\widehat{\phi}, \psi'), \; \lambda p.\, p \Rightarrow \widehat{Q}((\sigma_k)^n, (\sigma'_k)^n)), \tag{5}$$

then we have $\psi : \{\widehat{P}\}(c_k)^n\{\widehat{Q}\}$.

In other words, a relational property is valid if all relational procedure contracts are valid, and, assuming the relational precondition holds, both the auxiliary verification conditions and the main relational verification condition hold. The corresponding Coq formalization is available in file `Rela.v`, and the Coq proof of Theorem 4 is in file `Correct_Rela.v`.

Example 5. Consider $\psi = \psi_{\text{sum}}$ and $\widehat{\phi}$ which encodes $\mathcal{R}_2$ and $\mathcal{R}_3$. The relational property $\mathcal{R}_1$ of Fig. 1 can now be proven valid in a modular way, using $\mathcal{R}_2$ and $\mathcal{R}_3$, by the proposed technique based on Theorem 4 (see file `Examples.v` of the Coq development). For instance, (5) becomes the formula of Fig. 8. There, the relational precondition is given by (6), while the simplified (instantiated for sequence $(y_{\text{sum}}, y_{\text{sum}})$) translation of the relational contracts $\mathcal{T}_{pr}(\widehat{\phi}, \psi)$ is given by (7). Finally, (8) gives the main verification condition:

$$\mathcal{T}_{cr}^{\triangleright}((c_\omega^1, c_\omega^2), (\sigma_1, \sigma_2), (\sigma'_1, \sigma'_2), \mathcal{L}(\widehat{\phi}, \psi'), \lambda p.p \Rightarrow \sigma'_1[3] = \sigma'_2[3]), \text{ where } \mathcal{L}(\widehat{\phi}, \psi') =$$
$$\{y_{\text{sum}} \rightarrow (\lambda\sigma.\,\text{True}, \lambda\sigma, \sigma'.\, \sigma[1] \geqslant \sigma[2] \Rightarrow \sigma[3] = \sigma'[3] \wedge \mathcal{P}_{call}(y_{\text{sum}}, \sigma, \sigma', \psi'))\}.$$

Long for a manual proof, such formulas are well-treated by solvers. □

7 Related Work

Relational Property Verification. Significant work has been done on relational program verification (see [27,26] for a detailed state of the art). We discuss below some of the efforts the most closely related to our work.

Various relational logics have been designed as extensions to Hoare Logic, such as Relational Hoare Logic [6] and Cartesian Hoare Logic [32]. As our approach, those logics consider for each command a set of associated memory states in the very rules of the system, thus avoiding additional separation assumptions. Limitations of these logics are often the absence of support for aliasing or a limited form of relational properties. For instance, Relational Hoare Logic supports only relational properties with two commands and Cartesian Hoare Logic supports only k-safety properties (relational properties on the same command). Our method has an advanced support of aliasing and supports a very general definition of relational properties, possibly between several dissimilar commands.

$$\forall \sigma_1, \sigma_2, \sigma'_1, \sigma'_2, \psi.$$

$$\boxed{\sigma_1(1) = \sigma_2(1)} \tag{6}$$

$$\wedge$$

$$\boxed{\begin{array}{c} (\forall \sigma_1, \sigma_2, \sigma'_1, \sigma'_2. \\ \mathcal{P}_{call}(y_{\text{sum}}, \sigma_1, \sigma'_1, \psi) \wedge \mathcal{P}_{call}(y_{\text{sum}}, \sigma_2, \sigma'_2, \psi) \wedge \\ \sigma_2(1) < \sigma_2(2) \wedge \sigma_1(2) = \sigma_2(2) \wedge \\ \sigma_1(1) = \sigma_2(1) + 1 \wedge \sigma_1(3) = \sigma_2(3) + \sigma_2(1) \\ \Rightarrow \\ \sigma'_1(3) = \sigma'_2(3)) \end{array}} \tag{7}$$

$$\Rightarrow$$

$$\boxed{\begin{array}{c} \forall \sigma''_1, \sigma'''_1, \sigma''_2, \sigma'''_2. \\ \sigma''_1 = set(\sigma_1, 1, 1) \wedge \sigma'''_1 = set(\sigma''_1, 3, 0) \wedge \\ ((\sigma'''_1(1) \geqslant \sigma'''_1(2) \Rightarrow \sigma'''_1(3) = \sigma'_1(3)) \wedge \mathcal{P}_{call}(y_{\text{sum}}, \sigma'''_1, \sigma'_1, \psi)) \wedge \\ \sigma''_2 = set(\sigma_2, 1, 0) \wedge \sigma'''_2 = set(\sigma''_2, 3, 0) \wedge \\ ((\sigma'''_2(1) \geqslant \sigma'''_2(2) \Rightarrow \sigma'''_2(3) = \sigma'_2(3)) \wedge \mathcal{P}_{call}(y_{\text{sum}}, \sigma'''_2, \sigma'_2, \psi)) \\ \Rightarrow \\ \sigma'_1(3) = \sigma'_2(3) \end{array}} \tag{8}$$

Figure 8: Assumption (5) of Theorem 4 illustrated for property $\mathcal{R}_1$ of Fig. 1.

Self-compositon [3,30,9] and its derivations [2,31,14] are well-known approaches to deal with relational properties. This is in particular due to their flexibility: self-composition methods can be applied as a preprocessing step to different verification approaches. For example, self-composition is used in combination with symbolic execution and model checking for verification of voting functions [5]. Other examples are the use of self-composition in combination with verification condition generation in the context of the Java language [13] or the C language [9,10]. In general, the support of aliasing of C programs in these last efforts is very limited due the problems mentioned earlier. Compared to these techniques, where self-composition is applied before the generation of verification conditions (and therefore requires taking care about separation of memory states of the considered programs), our method can be seen as relating the considered programs' semantics directly at the level of the verification conditions, where separation of their memory states is already ensured, thus avoiding the need to take care of this separation explicitly.

Finally, another advanced approach for relational verification is the translation of the relational problem into Horn clauses and their proof using constraint solving [22,34]. The benefit of constraint solving lies in the ability to automatically find relational invariants and complex self-composition derivations. Moreover, the translation of programs into Horn clauses, done by tools like REVE[9], results in formulas similar to those generated by our VCGen. Therefore, like our approach, relational verification with constraint solving requires no additional separation hypothesis in presence of aliasing.

Certified Verification Condition Generation. In a broad sense, this work continues previous efforts in formalization and mechanized proof of program language semantics, analyzers and compilers, such as [29,25,18,7,20,21,35,24,12,28]. Generation of certificates (in Isabelle) for the BOOGIE verifier is presented in [28]. The certified deductive verification tool WhyCert [18] comes with a similar soundness result for its verification condition generator. Its formalization follows an alternative proof approach, based on co-induction, while our proof relies on induction. WhyCert is syntactically closer to the C language and the ACSL specification language [4], while our proof uses a simplified language, but with a richer aliasing model. Furthermore, we provide a formalization and a soundness proof for relational verification, which was not considered in WhyCert or in [28].

Our previous work [11] presented a method for relational property verification based on a naive VCGen. To the best of our knowledge, the present work is the first proposal of *modular* relational property verification based on an *optimized* VCGen for a representative language with procedure calls and aliases with a full mechanized formalization and proof of soundness in COQ.

8 Conclusion

We have presented in this paper an overview of a method for modular verification of relational properties using an optimized verification condition generator, without relying on code transformations (such as self-composition) or making additional separation hypotheses in case of aliasing. This method has been fully formalized in COQ, and the soundness of recursive relational verification using a verification condition generator (itself formally proved correct) for a simple language with procedure calls and aliasing has been formally established.

This work opens the door for interesting future work. Currently, for relational properties, product programs [2] or other self-composition optimizations [31] are the standard approach to deal with complex loop constructions. We expect that user-provided coupling invariants and loop properties can avoid having to rely on code transformation methods. Showing this in our framework is the next step, before the investigation of termination and co-termination [17],[34] for extending the modularity of relational contracts.

References

1. Apt, K., de Boer, F., Olderog, E.: Verification of Sequential and Concurrent Programs. Texts in Computer Science, Springer (2009). https://doi.org/10.1007/978-1-84882-745-5

[9] `https://formal.kastel.kit.edu/projects/improve/reve/`

2. Barthe, G., Crespo, J.M., Kunz, C.: Relational verification using product programs. In: Proc. of the 17th International Symposium on Formal Methods (FM 2011). LNCS, vol. 6664, pp. 200–214. Springer (2011). https://doi.org/10.1007/978-3-642-21437-0_17
3. Barthe, G., D'Argenio, P.R., Rezk, T.: Secure information flow by self-composition. J. of Mathematical Structures in Computer Science **21**(6), 1207–1252 (2011). https://doi.org/10.1017/S0960129511000193
4. Baudin, P., Cuoq, P., Filliâtre, J.C., Marché, C., Monate, B., Moy, Y., Prevosto, V.: ACSL: ANSI/ISO C Specification Language (2021), `https://frama-c.com/html/acsl.html`
5. Beckert, B., Bormer, T., Kirsten, M., Neuber, T., Ulbrich, M.: Automated verification for functional and relational properties of voting rules. In: Proc. of the 6th International Workshop on Computational Social Choice (COMSOC 2016) (2016)
6. Benton, N.: Simple relational correctness proofs for static analyses and program transformations. In: Proc. of the 31st ACM SIGPLAN-SIGACT Symposium on of Programming Languages (POPL 2004). pp. 14–25. ACM (2004). https://doi.org/10.1145/964001.964003
7. Beringer, L., Appel, A.W.: Abstraction and subsumption in modular verification of C programs. In: Proc. of the Third World Congress on Formal Methods - (FM 2019). LNCS, vol. 11800, pp. 573–590. Springer (2019). https://doi.org/10.1007/978-3-030-30942-8_34
8. Bishop, P.G., Bloomfield, R.E., Cyra, L.: Combining testing and proof to gain high assurance in software: A case study. In: Proc. of the 24th International Symposium on Software Reliability Engineering (ISSRE 2013). pp. 248–257. IEEE (2013). https://doi.org/10.1109/ISSRE.2013.6698924
9. Blatter, L., Kosmatov, N., Le Gall, P., Prevosto, V.: RPP: automatic proof of relational properties by self-composition. In: Proc. of the 23rd International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2017). LNCS, vol. 10205, pp. 391–397. Springer (2017). https://doi.org/10.1007/978-3-662-54577-5_22
10. Blatter, L., Kosmatov, N., Le Gall, P., Prevosto, V., Petiot, G.: Static and dynamic verification of relational properties on self-composed C code. In: Proc. of the 12th International Conference on Tests and Proofs (TAP 2018). LNCS, vol. 10889, pp. 44–62. Springer (2018). https://doi.org/10.1007/978-3-319-92994-1_3
11. Blatter, L., Kosmatov, N., Prevosto, V., Le Gall, P.: Certified verification of relational properties. In: Proc. of the 17th International Conference on integrated Formal Methods (iFM 2022). LNCS, Springer (Jun 2022), to appear
12. Blazy, S., Maroneze, A., Pichardie, D.: Verified validation of program slicing. In: Proc. of the 2015 Conference on Certified Programs and Proofs (CPP 2015). pp. 109–117. ACM (2015). https://doi.org/10.1145/2676724.2693169
13. Dufay, G., Felty, A.P., Matwin, S.: Privacy-sensitive information flow with JML. In: Proc. of the 20th Conference on Automated Deduction (CADE 2005). LNCS, vol. 3632, pp. 116–130. Springer (2005). https://doi.org/10.1007/11532231_9
14. Eilers, M., Müller, P., Hitz, S.: Modular product programs. In: Proc. of the 27th European Symposium on Programming (ESOP 2018). LNCS, vol. 10801, pp. 502–529. Springer (2018). https://doi.org/10.1007/978-3-319-89884-1_18
15. Flanagan, C., Saxe, J.B.: Avoiding exponential explosion: generating compact verification conditions. In: Proc. of the 28th ACM SIGPLAN Symposium on Principles of Programming Languages (POPL 2001). pp. 193–205. ACM (2001). https://doi.org/10.1145/360204.360220

16. Floyd, R.W.: Assigning meanings to programs. In: Proc. of Symposia in Applied Mathematics. vol. 19 (Mathematical Aspects of Computer Science), p. 19–32 (1967). https://doi.org/10.1090/psapm/019/0235771
17. Hawblitzel, C., Kawaguchi, M., Lahiri, S.K., Rebêlo, H.: Towards modularly comparing programs using automated theorem provers. In: Proc. of the 24th International Conference on Automated Deduction (CADE 2013). LNCS, vol. 7898, pp. 282–299. Springer (2013). https://doi.org/10.1007/978-3-642-38574-2_20
18. Herms, P.: Certification of a Tool Chain for Deductive Program Verification. Phd thesis, Université Paris Sud - Paris XI (Jan 2013), https://tel.archives-ouvertes.fr/tel-00789543
19. Hoare, C.A.R.: An axiomatic basis for computer programming. Communications of the ACM **12**(10), 576–580 (1969). https://doi.org/10.1145/363235.363259
20. Jourdan, J., Laporte, V., Blazy, S., Leroy, X., Pichardie, D.: A formally-verified C static analyzer. In: Proc. of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL 2015). pp. 247–259. ACM (2015). https://doi.org/10.1145/2676726.2676966
21. Jung, R., Krebbers, R., Jourdan, J., Bizjak, A., Birkedal, L., Dreyer, D.: Iris from the ground up: A modular foundation for higher-order concurrent separation logic. J. Funct. Program. **28**, e20 (2018). https://doi.org/10.1017/S0956796818000151
22. Kiefer, M., Klebanov, V., Ulbrich, M.: Relational program reasoning using compiler IR - combining static verification and dynamic analysis. J. of Automated Reasoning **60**(3), 337–363 (2018). https://doi.org/10.1007/s10817-017-9433-5
23. Kip, I.: Assembly Language for x86 Processors. Prentice Hall Press, 7th edn. (2014)
24. Krebbers, R., Leroy, X., Wiedijk, F.: Formal C semantics: CompCert and the C standard. In: Proc. of the 5th International Conference on Interactive Theorem Proving (ITP 2014), Held as Part of the Vienna Summer of Logic (VSL 2014). LNCS, vol. 8558, pp. 543–548. Springer (2014). https://doi.org/10.1007/978-3-319-08970-6_36
25. Leroy, X., Blazy, S.: Formal verification of a C-like memory model and its uses for verifying program transformations. Journal of Automated Reasoning **41**(1), 1–31 (2008)
26. Maillard, K., Hritcu, C., Rivas, E., Van Muylder, A.: The next 700 relational program logics. In: Proc. of the 47th ACM SIGPLAN Symposium on Principles of Programming Languages (POPL 2020). vol. 4, pp. 4:1–4:33 (2020). https://doi.org/10.1145/3371072
27. Naumann, D.A.: Thirty-seven years of relational Hoare logic: Remarks on its principles and history. In: Proc. of the 9th International Symposium on Leveraging Applications of Formal Methods (ISoLA 2020). LNCS, vol. 12477, pp. 93–116. Springer (2020). https://doi.org/10.1007/978-3-030-61470-6_7
28. Parthasarathy, G., Müller, P., Summers, A.J.: Formally validating a practical verification condition generator. In: Proc. of the 33rd International Conference on Computer Aided Verification (CAV 2021). LNCS, vol. 12760, pp. 704–727. Springer (2021). https://doi.org/10.1007/978-3-030-81688-9_33
29. Pierce, B.C., Azevedo de Amorim, A., Casinghino, C., Gaboardi, M., Greenberg, M., Hriţcu, C., Sjöberg, V., Yorgey, B.: Logical Foundations. Software Foundations series, volume 1, Electronic textbook (2018), http://www.cis.upenn.edu/~bcpierce/sf
30. Scheben, C., Schmitt, P.H.: Efficient self-composition for weakest precondition calculi. In: Proc. of the 19th International Symposium on Formal Methods (FM 2014). LNCS, vol. 8442, pp. 579–594. Springer (2014). https://doi.org/10.1007/978-3-319-06410-9_39

31. Shemer, R., Gurfinkel, A., Shoham, S., Vizel, Y.: Property directed self composition. In: Proc. of the 31th International Conference on Computer Aided Verification (CAV 2019). LNCS, vol. 11561, pp. 161–179. Springer (2019). https://doi.org/10.1007/978-3-030-25540-4_9
32. Sousa, M., Dillig, I.: Cartesian Hoare Logic for Verifying k-safety Properties. In: Proc. of the 37th Conference on Programming Language Design and Implementation (PLDI 2016). pp. 57–69. ACM (2016). https://doi.org/10.1145/2908080.2908092
33. The Coq Development Team: The Coq Proof Assistant (2021), `https://coq.inria.fr/`
34. Unno, H., Terauchi, T., Koskinen, E.: Constraint-based relational verification. In: Proc. of the 33th International Conference on Computer Aided Verification (CAV 2021). LNCS, vol. 12759, pp. 742–766. Springer (2021). https://doi.org/10.1007/978-3-030-81685-8_35
35. Wils, S., Jacobs, B.: Certifying C program correctness with respect to compcert with verifast. CoRR **abs/2110.11034** (2021), `https://arxiv.org/abs/2110.11034`
36. Winskel, G.: The formal semantics of programming languages - an introduction. Foundation of computing series, MIT Press (1993)

Appendix

A Complete Semantics of Language L

A.1 Evaluation of Arithmetic and Boolean Expressions in L

We provide a complete list of rules for evaluation of arithmetic and Boolean expressions in L in Fig. 9. Evaluation of arithmetic and Boolean expressions in $\mathcal{L}$ is defined by functions ξ_a and ξ_b. As mentioned above, the subtraction is lower-bounded by 0. Operations $*x_i$ and $\&x_i$ have a semantics similar to the C language, i.e. dereferencing and address-of. Semantics of Boolean expressions is standard [36].

$$
\begin{aligned}
\xi_a[\![n]\!]\sigma &\triangleq n & \xi_b[\![true]\!]\sigma &\triangleq \text{True} \\
\xi_a[\![x_i]\!]\sigma &\triangleq \sigma(i) & \xi_b[\![false]\!]\sigma &\triangleq \text{False} \\
\xi_a[\![*x_i]\!]\sigma &\triangleq \sigma(\sigma(i)) & \xi_b[\![a_1\ op_b\ a_2]\!]\sigma &\triangleq \xi_a[\![a_1]\!]\sigma\ op_a\ \xi_a[\![a_2]\!]\sigma \\
\xi_a[\![\&x_i]\!]\sigma &\triangleq i & \xi_b[\![b_1\ op_l\ b_2]\!]\sigma &\triangleq \xi_b[\![b_1]\!]\sigma\ op_l\ \xi_b[\![b_2]\!]\sigma \\
\xi_a[\![a_1\ op_a\ a_2]\!]\sigma &\triangleq \xi_a[\![a_1]\!]\sigma\ op_a\ \xi_a[\![a_2]\!]\sigma & \xi_b[\![\neg b]\!]\sigma &\triangleq \neg\xi_b[\![b]\!]\sigma
\end{aligned}
$$

Figure 9: Evaluation of arithmetic and Boolean expressions in L.

A.2 Operational Semantics of Commands in L in L

We provide a complete operational semantics of commands in L in Fig. 10.

$$\langle \mathbf{skip}, \sigma \rangle \xrightarrow{\psi} \sigma \qquad \frac{\xi_a[\![a]\!]\sigma = n}{\langle x_i := a, \sigma \rangle \xrightarrow{\psi} \sigma[i/n]} \qquad \frac{\xi_a[\![a]\!]\sigma = n}{\langle *x_i := a, \sigma \rangle \xrightarrow{\psi} \sigma[\sigma(i)/n]}$$

$$\langle \mathbf{assert}(P), \sigma \rangle \xrightarrow{\psi} \sigma \qquad \frac{\xi_b[\![b]\!]\sigma = \text{True} \qquad \langle c_1, \sigma_1 \rangle \xrightarrow{\psi} \sigma_2}{\langle \mathbf{if}\ b\ \mathbf{then}\ \{c_1\}\ \mathbf{else}\ \{c_2\}, \sigma_1 \rangle \xrightarrow{\psi} \sigma_2}$$

$$\frac{\langle c_1, \sigma_1 \rangle \xrightarrow{\psi} \sigma_2 \qquad \langle c_2, \sigma_2 \rangle \xrightarrow{\psi} \sigma_3}{\langle c_1; c_2, \sigma_1 \rangle \xrightarrow{\psi} \sigma_3} \qquad \frac{\xi_b[\![b]\!]\sigma = \text{False} \qquad \langle c_2, \sigma_1 \rangle \xrightarrow{\psi} \sigma_2}{\langle \mathbf{if}\ b\ \mathbf{then}\ \{c_1\}\ \mathbf{else}\ \{c_2\}, \sigma_1 \rangle \xrightarrow{\psi} \sigma_2}$$

$$\frac{\xi_b[\![b]\!]\sigma_1 = \text{True} \qquad \langle c_1, \sigma_1 \rangle \xrightarrow{\psi} \sigma_2 \qquad \langle \mathbf{while}\ b\ \mathbf{inv}\ P\ \mathbf{do}\ \{c\}, \sigma_2 \rangle \xrightarrow{\psi} \sigma_3}{\langle \mathbf{while}\ b\ \mathbf{inv}\ P\ \mathbf{do}\ \{c\}, \sigma_1 \rangle \xrightarrow{\psi} \sigma_3}$$

$$\frac{\xi_b[\![b]\!]\sigma = \text{False}}{\langle \mathbf{while}\ b\ \mathbf{inv}\ P\ \mathbf{do}\ \{c\}, \sigma \rangle \xrightarrow{\psi} \sigma} \qquad \frac{\langle \text{body}_\psi(y), \sigma_1 \rangle \xrightarrow{\psi} \sigma_2}{\langle \mathbf{call}(y), \sigma_1 \rangle \xrightarrow{\psi} \sigma_2}$$

Figure 10: Operational semantics of commands in L.