Certified Verification of Relational Properties[⋆]

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

[1] Karlsruhe Institute of Technology
`firstname.lastname@kit.edu`
[2] Université Paris-Saclay, CEA, List, 91120, Palaiseau, France
`firstname.lastname@cea.fr`
[3] Thales Research & Technology, 91120, Palaiseau, France
[4] CentraleSupélec, Université Paris-Saclay, 91190 Gif-sur-Yvette France
`firstname.lastname@centralesupelec.fr`

Abstract The use of function contracts to specify the behavior of functions often remains limited to the scope of 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, such as non-interference, continuity, or monotonicity. They can also relate calls to different functions, for instance, to show that an optimized implementation is equivalent to its original counterpart. However, relational properties cannot be expressed and verified directly in the traditional setting of modular deductive verification. Self-composition has been proposed to overcome this limitation, but it requires complex transformations and additional separation hypotheses for real-life languages with pointers. We propose a novel approach that is not based on code transformation and avoids those drawbacks. It directly applies a verification condition generator to produce logical formulas that must be verified to ensure a given relational property. The approach has been fully formalized and proved sound in the Coq proof assistant.

1 Introduction

Modular deductive verification [18] allows the user to prove that a function respects its formal specification. More precisely, for a given function f, any individual call to f can be proved to respect the *contract* of f, that is, basically an implication: if the given *precondition* is true before the call and the call terminates[5], the given *postcondition* is true after it. However, some kinds of properties are not easily reducible to a single function call. Indeed, it is frequently necessary to express a property that involves several functions or relates the results of several calls to the same function for different arguments. Such properties are known as *relational properties* [6].

⋆ Part of this work was funded by the AESC project supported by the Ministry of Science, Research and Arts Baden-Württemberg (Ref: 33-7533.-9-10/20/1).

[5] Termination can be assumed (partial correctness) or proved separately (full correctness) in a well-known way [15]; for the purpose of this paper we can assume it.

```
//C program C_sw1 :
x3  = *x1;
*x1 = *x2;
*x2 = x3;
//C program C_sw2 :
*x1 = *x1 + *x2;
*x2 = *x1 - *x2;
*x1 = *x1 - *x2;
```

```
//Composed C program C_sw3 :
x3_1  = *x1_1;
*x1_1 = *x2_1;
*x2_1 = x3_1;

*x1_2 = *x1_2 + *x2_2;
*x2_2 = *x1_2 - *x2_2;
*x1_2 = *x1_2 - *x2_2;
```

$$c_{\text{sw1}} \triangleq \begin{array}{l} x_3 := *x_1; \\ *x_1 := *x_2; \\ *x_2 := x_3; \end{array}$$

$$c_{\text{sw2}} \triangleq \begin{array}{l} *x_1 := *x_1 + *x_2; \\ *x_2 := *x_1 - *x_2; \\ *x_1 := *x_1 - *x_2 \end{array}$$

Figure 1: Two C programs $\mathcal{C}_{\text{sw1}}$ and $\mathcal{C}_{\text{sw2}}$ swapping `*x1` and `*x2`, their composition $\mathcal{C}_{\text{sw3}}$, and their counterparts c_{sw1} and c_{sw2} in language $\mathcal{L}$ (defined below).

Examples of such relational properties include monotonicity (i.e. $x \leq y \Rightarrow f(x) \leq f(y)$), involving two calls, or transitivity ($cmp(x, y) \geq 0 \wedge cmp(y, z) \geq 0 \Rightarrow cmp(x, z) \geq 0$), involving three 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.

Relational properties can also relate calls to different functions. For instance, in the verification of voting rules [5], relational properties are used for defining specific properties (such as monotonicity, anonymity or consistency). Notably, applying the voting rule to a sequence of ballots and a permutation of the same sequence of ballots must lead to the same result, i.e. the order in which the ballots are passed to the voting function should not have any impact on the outcome.

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 program [2] or other self-composition optimizations [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.

Example 1 (Motivating Example). Figure 1 shows an example of two simple C programs performing a swap of the values referred to by pointers `x1` and `x2` (of type `int*`). Program $\mathcal{C}_{\text{sw1}}$ uses an auxiliary variable `x3` (of type `int`), while $\mathcal{C}_{\text{sw2}}$ performs an in-place swap using arithmetic operations. As usual in that case, to work correctly, each of these programs needs some separation hypotheses: pointers `x1` and `x2` should be *separated* (that is, point to disjoint memory locations) and must not point to `x1`, `x2` themselves and, for $\mathcal{C}_{\text{sw1}}$, to `x3`.

Consider a relational property, denoted $\mathcal{R}_{\mathrm{sw}}$, stating that both programs, executed from two states in which each of `*x1` and `*x2` has the same value, will end up in two states also having the same values in these locations. To prove this relational property using self-composition, one typically has to generate a new C program $\mathcal{C}_{\mathrm{sw3}}$ (see Fig. 1) composing $\mathcal{C}_{\mathrm{sw1}}$ and $\mathcal{C}_{\mathrm{sw2}}$. To avoid name conflicts, we rename their variables by adding, resp., suffixes "`_1`" and "`_2`". The relational property $\mathcal{R}_{\mathrm{sw}}$ is then expressed by a contract of $\mathcal{C}_{\mathrm{sw3}}$ with a precondition P and a postcondition Q. Obviously, both P and Q must include the equalities: `*x1_1==*x1_2` and `*x2_1==*x2_2`, and P must also require the aforementioned separation hypotheses necessary for each function. But for programs with pointers and aliasing, this is not sufficient: the user also has to specify additional separation hypotheses[6] between variables coming from the different programs, that is, in our example, that each of `x1_1` and `x2_1` is separated from each of `x1_2` and `x2_2`. Without such hypotheses, a deductive verification tool cannot show, for example, that a modification of `*x1_1` does not impact `*x1_2` in the composed program $\mathcal{C}_{\mathrm{sw3}}$, and is thus unable to deduce the required property. For real-life programs, such separation hypotheses can be hard to specify or generate. It can become even more complicated for programs with double or multiple indirections. □

Approach. This paper proposes an alternative approach that is not based on code transformation or relational rules. It directly uses a verification condition generator (VCGen) to produce logical formulas to be verified (typically, with an automated prover) to ensure a given relational property. It requires no extra code processing (such as sequential composition of programs or variable renaming). Moreover, no additional separation hypotheses—in addition to those that are anyway needed for each function to work—are required. The locations of each program are separated by construction: each program has its own memory state. The language $\mathcal{L}$ considered in this work was chosen as a minimal language representative of the main issues relevant for relational property verification: it is a standard WHILE language enriched with annotations, procedures and pointers (see programs c_{sw1} and c_{sw2} in Fig. 1 for examples; we use a lower-case letter c for $\mathcal{L}$ programs and a capital letter $\mathcal{C}$ for C programs). Notably, the presence of dereferencing and address-of operations makes it representative of various aliasing problems with (possibly, multiple) pointer dereferences of a real-life language like C. We formalize the proposed approach and prove[7] its soundness in the COQ proof assistant [33]. Our COQ development contains about 3400 lines.

Contributions. The contributions of this paper include:

– a COQ formalization and proof of soundness of recursive Hoare triple verification with a verification condition generator on a representative language with procedures and aliasing;

[6] For convenience of the reader, P and Q are defined in detail in Appendix A.

[7] The COQ development is at `https://github.com/lyonel2017/Relational-Spec`, where the version corresponding to this paper is tagged iFM2022.

- a novel method for verifying relational properties using a verification condition generator, without relying on code transformation (such as self-composition) or making additional separation hypotheses in case of aliasing;
- a Coq formalization and proof of soundness of the proposed method of relational property verification for the considered language.

Outline. Section 2 introduces an imperative language $\mathcal{L}$ used in this work. Functional correctness is defined in Section 3, and relational properties in Section 4. Then, we prove the soundness of a 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.

2 Syntax and Semantics of the $\mathcal{L}$ Language

2.1 Notation for 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 $x, x', x_1, ...$ and $y, y', y_1, ...$ denote metavariables ranging over those respective sets. We assume that there exists a bijective function $\mathbb{N} \to \mathbb{X}$, so that $\mathbb{X} = \{x_i \mid i \in \mathbb{N}\}$. 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 $\sigma, \sigma', \sigma_1, ...$ denote metavariables ranging over the set. 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 $\psi, \psi_1, ...$ denote metavariables ranging over Ψ. We write $\text{body}_\psi(y)$ to refer to $\psi(y)$, the commands (or *body*) of procedure y for a given procedure environment ψ.

Assertions are predicates of arity one, taking as parameter a memory state and returning an equational first-order logic formula. Let metavariables $P, Q, ...$ range over the set $\mathbb{A}$ of assertions. For instance, using λ-notation, assertion 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.

Finally, we define the set Φ of *contract environments* $\phi : \mathbb{Y} \to \mathbb{A} \times \mathbb{A}$, and metavariables $\phi, \phi_1, ...$ 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 procedure call, that is, if we start executing y in a memory state satisfying $\text{pre}_\phi(y)$, and the evaluation terminates, the final state satisfies $\text{post}_\phi(y)$.

$a ::= n$	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 $\mathcal{L}$.

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 $a, a_1, ...; b, b_1, ...$ and $c, c_1, ...$ 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 $op_a ::= \{+, \times, -\}$, $op_b ::= \{<=, =\}$, $op_l ::= \{\vee, \wedge\}$. 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 address operators.

Figure 2 also presents the syntax of commands in $\mathcal{L}$. 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 [22], 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 delimit sequences of commands with { }.

Example 2. Figure 3 shows an example of a command c_{rec} and a procedure environment ψ where procedure y_1 points to a recursive command, called in c_{rec}.

$$c_{\text{rec}} \triangleq \begin{array}{l} x_1 := x_4; \\ x_2 := 0; \\ \textbf{call}(y_1) \end{array} \qquad \psi = \left\{ y_1 \rightarrow \begin{array}{l} \textbf{if } x_1 > 0 \textbf{ then } \{ \\ \quad x_2 := x_2 + x_3; \\ \quad x_1 := x_1 - 1; \\ \quad \textbf{call}(y_1) \\ \} \textbf{ else } \{ \\ \quad \textbf{skip} \\ \} \end{array} , \; \dots \right\}$$

$$\phi = \left\{ y_1 \rightarrow \begin{pmatrix} \lambda\sigma.\sigma(2) = \sigma(3) \times (\sigma(4) - \sigma(1)) \land 0 \leq \sigma(1) \land \sigma(1) \leq \sigma(4), \\ \lambda\sigma.\sigma(2) = \sigma(3) \times \sigma(4) \end{pmatrix}, \; \dots \right\}.$$

Figure 3: Example of an $\mathcal{L}$ program c_{rec} with its environments.

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

Figure 4: Evaluation of expressions in $\mathcal{L}$ (selected rules).

With the semantics of Sec. 2.3, from any initial state, the command will return a state in which $x_2 = x_3 \times x_4$. Procedure y_1 returns a state where $x_2 = x_3 \times x_4$ if the initial state satisfies $x_2 = x_3 \times (x_4 - x_1) \land 0 \leq x_1 \land x_1 \leq x_4$. This can be expressed by the contract environment ϕ given (in λ-notation) in Fig. 3. $\square$

2.3 Operational Semantics

Evaluation of arithmetic and Boolean expressions in $\mathcal{L}$ is defined by functions ξ_a and ξ_b. Selected evaluation rules for arithmetic expressions are shown in Fig. 4. 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. 5. 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 **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. 5. Our Coq formalization, inspired by [29], provides a deep embedding of $\mathcal{L}$, 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 [18], 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.

[8] For convenience of the reader, full versions of Fig. 4, 5 are given in Appendix B.

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

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

Figure 5: Operational semantics of commands in $\mathcal{L}$ (selected rules).

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:*

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

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' satisfies Q.

Next, we introduce notation $CV(\psi, \phi)$ to denote the fact that, for the given ϕ and ψ, 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 assumption (3) in Lemma 1. This technical lemma, taken from [1, Equation (4.6)], gives an alternative criterion for validity of procedure contracts: if, under the assumption that the contracts in ϕ hold, we can prove for each procedure y that its body satisfies its contract, then the contracts are valid.

Lemma 1 (Adequacy of contracts). *Given a procedure environment ψ and a contract environment ϕ such that*

$$\forall \psi' \in \Psi.\ CV(\psi', \phi) \Rightarrow \forall y \in \mathbb{Y}, \psi' : \{\mathrm{pre}_\phi(y)\}\mathrm{body}_\psi(y)\{\mathrm{post}_\phi(y)\}, \tag{3}$$

we have $CV(\psi, \phi)$.

Proof. Any given terminating execution traverses a finite number of procedure calls (over all procedures) that can be replaced by inlining the bodies a sufficient number of times. We first formalize a theory of k-inliners (that inline procedure bodies a finite number of times $k \geq 0$ and replace deeper calls by nonterminating loops) and prove their properties. Relying on this elegant theory, the proof of the lemma proceeds by induction on the number of procedure inlinings. $\square$

From that, we can establish the main result of this section. Theorem 1, taken from [1, Th. 4.2] states that $\psi : \{P\}c\{Q\}$ holds if assumption (3) holds 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 [18,36] or verification condition generation.

Theorem 1 (Recursion). *Given a procedure environment ψ and a contract environment ϕ such that*

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

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

Proof. By Lemma 1. □

We refer the reader to the Coq development, more precisely the results `recursive_proc` and `recursive_hoare_triple` in file `Hoare_Triple.v` for complete proofs of Lemma 1 and Theorem 1 for $\mathcal{L}$. To the best of our knowledge, this is the first mechanized proof of these classical results.

An interesting corollary can be deduced from Theorem 1.

Corollary 1 (Procedure Recursion) *Given a procedure environment ψ and a contract environment ϕ such that*

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

we have $\forall y \in \mathbb{Y}.\ \psi : \{\mathrm{pre}_\phi(y)\}\mathrm{body}_\psi(y)\{\mathrm{post}_\phi(y)\}$.

4 Relational Properties

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 assertions. We define a *relational assertion* as a predicate taking a sequence of memory states and returning a first-order logic formula. We use metavariables $\widehat{P}, \widehat{Q}, ...$ to range over the set of relational assertions, denoted $\widehat{\mathbb{A}}$. As a simple example of a relational assertion, we might say that two states bind location x_3 to the same value. This would be stated as follows: $\lambda(\sigma_1, \sigma_2).\sigma_1(3) = \sigma_2(3)$.

A *relational property* is a property about n programs $c_1, ..., c_n$, stating that if each program c_i starts in a state σ_i and ends in a state σ'_i such that $\widehat{P}(\sigma_1, ..., \sigma_n)$ holds, then $\widehat{Q}(\sigma'_1, ..., \sigma'_n)$ holds, where $\widehat{P}$ and $\widehat{Q}$ are relational assertions over n memory states.

We formally define relational correctness similarly to functional correctness (cf. Def. 1), except that we now use sequences of memory states and commands of equal length. We denote 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 [ ].

$$\psi : \{\widehat{P}\}\ c_{\mathrm{sw1}}\ \sim\ c_{\mathrm{sw2}}\ \{\widehat{Q}\},$$

$$\widehat{P} \triangleq\ \lambda\sigma_1\sigma_2.\ \sigma_1(\sigma_1(1)) = \sigma_2(\sigma_2(1)) \wedge \sigma_1(\sigma_1(2)) = \sigma_2(\sigma_2(2)) \wedge$$
$$\sigma_1(1) \neq \sigma_1(2) \wedge \sigma_2(1) \neq \sigma_2(2) \wedge \sigma_1(1) > 3 \wedge \sigma_1(2) > 3 \wedge \sigma_2(1) > 2 \wedge \sigma_2(2) > 2,$$

$$\widehat{Q} \triangleq \lambda\sigma'_1\sigma'_2.\ \sigma'_1(\sigma'_1(1)) = \sigma'_2(\sigma'_2(1)) \wedge \sigma'_1(\sigma'_1(2)) = \sigma'_2(\sigma'_2(2)).$$

Figure 6: A relational property for $\mathcal{L}$ programs c_{sw1} and c_{sw2} of Fig. 1.

Definition 3 (Relational Correctness). *Let ψ be a procedure environment, $(c_k)^n$ a sequence of n commands ($n \in \mathbb{N}^*$), and $\widehat{P}$ and $\widehat{Q}$ two relational assertions over n states. 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:*

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

This notation generalizes the one proposed by Benton [6] for relational properties linking 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. In particular, Benton's work would not be practical for verification of relational properties with several calls such as transitivity mentioned in Sec. 1.

Example 3 (Relational property). Figure 6 formalizes the relational property $\mathcal{R}_{\mathrm{sw}}$ for $\mathcal{L}$ programs c_{sw1} and c_{sw2} discussed in Ex. 1. Recall that $\mathcal{R}_{\mathrm{sw}}$ (written in Fig. 6 in Benton's notation) states that both programs executed from two states named σ_1 and σ_2 having the same values in $*x_1$ and $*x_2$ will end up in two states σ'_1 and σ'_2 also having the same values in these locations. Notice that the initial state of each program needs separation hypotheses (cf. the second line of the definition of $\widehat{P}$). Namely, x_1 and x_2 must point to different locations and must not point to x_1, x_2 or, for c_{sw1}, to x_3 for the property to hold. This relational property is formalized in the Coq development in file `Examples.v`. □

5 Verification Condition Generation for Hoare Triples

A standard way [15] 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 and show that it is correct, in the sense that if all verification conditions that it generates are valid, then the Hoare triple is valid according to Def. 1.

5.1 Verification Condition Generator

We have chosen to split the VCGen in three steps, as it is commonly done [23]:

– function $\mathcal{T}_c$ generates the main verification condition, expressing that the postcondition holds in the final state, assuming auxiliary annotations hold;

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

Figure 7: Definition of function $\mathcal{T}_c$ generating the main verification condition.

- function $\mathcal{T}_a$ generates auxiliary verification conditions stemming from assertions, loop invariants, and preconditions of called procedures;
- finally, function $\mathcal{T}_f$ generates verification conditions for the auxiliary procedures that are called by the main program, to ensure that their bodies respect their contracts.

Definition 4 (Function $\mathcal{T}_c$ generating the main verification condition). *Given a command c, a memory state σ representing the state before the command, a contract environment ϕ, and an assertion f, function $\mathcal{T}_c$ returns a formula defined by case analysis on c as shown in Fig. 7.*

Assertion f represents the postcondition we want to verify after the command executed from state σ. For each command, except sequence and branch, a fresh memory state σ' is introduced and related to the current memory state σ. The new memory state is given as parameter to f. 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, we simply compose the conditions, that is, we check that the final state of c_0 is such that f will be verified after executing c_1. For a conditional, we check that if the condition evaluates to true, the *then* branch will ensure the postcondition, and that otherwise the *else* branch will ensure the postcondition. The rule for calls simply assumes that σ' verifies $\mathrm{post}_\phi(y)$. Finally, $\mathcal{T}_c$ assumes that, after a loop, σ' is a state where the loop condition is false and the loop invariant holds. As for an assertion, the callee's precondition and the loop invariant are just assumed to be true; function $\mathcal{T}_a$, defined below, generates the corresponding proof obligations.

Example 4. For $c \triangleq \mathbf{skip}; x_1 := 2$, and $f \triangleq \lambda\sigma.\ \sigma(1) = 2$, we have:

$$\mathcal{T}_c[\![c]\!](\sigma,\phi,f) \equiv \forall\sigma'_1.\sigma=\sigma'_1 \Rightarrow (\forall\sigma'_2.\sigma'_2 = \mathit{set}(\sigma'_1,1,2) \Rightarrow \sigma'_2(1)=2). \qquad \square$$

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

Figure 8: Definition of function $\mathcal{T}_a$ generating auxiliary verification conditions.

Definition 5 (Function $\mathcal{T}_a$ generating the auxiliary verification conditions). *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. 8.*

Basically, $\mathcal{T}_a$ 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 $\mathcal{T}_c$.

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 6 (Function $\mathcal{T}_f$ generating the procedure verification condition). *Given two environments ψ and ϕ, $\mathcal{T}_f$ returns the following formula:*

$$
\begin{aligned}
\mathcal{T}_f(\phi,\psi) \triangleq \forall y,\sigma.\ \mathrm{pre}_\phi(y)(\sigma) \Rightarrow \mathcal{T}_a[\![\mathrm{body}_\psi(y)]\!](\sigma,\phi)\wedge\\
\mathcal{T}_c[\![\mathrm{body}_\psi(y)]\!](\sigma,\phi,\mathrm{post}_\phi(y)).
\end{aligned}
$$

The VCGen is defined in file `Vcg.v` of the Coq development. Interested readers will also find a proof (in file `Vcg_Opt.v`) of a VCGen optimization (not detailed here), which prevents the size of the generated formulas from becoming exponential in the number of conditions in the program [14], which is a classical problem for "naive" VCGens.

5.2 Hoare Triple Verification

We can now state the theorems establishing correctness of the VCGen. Their proof can be found in file `Correct.v` of the Coq development.

First, Lemma 2 shows that, under the assumption of the procedure contracts, a Hoare triple is valid if for all memory states satisfying the precondition, the main verification condition and the auxiliary verification conditions hold.

Lemma 2. *Assume the following two properties hold:*

$$\forall \sigma \in \Sigma, P(\sigma) \Rightarrow \mathcal{T}_a[\![c]\!](\sigma, \phi),$$
$$\forall \sigma \in \Sigma, P(\sigma) \Rightarrow \mathcal{T}_c[\![c]\!](\sigma, \phi, Q).$$

Then we have $CV(\psi, \phi) \Rightarrow \psi : \{P\}c\{Q\}$.

Proof. By structural induction over c. □

Next, we prove in Lemma 3 that if $\mathcal{T}_f(\phi, \psi)$ holds, then for an arbitrary choice of implementations ψ' of procedures respecting the procedure contracts, the body of each procedure y respects its contract.

Lemma 3. *Assume that the formula $\mathcal{T}_f(\phi, \psi)$ is satisfied. Then we have*

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

Proof. By Lemma 2. □

Finally, we can establish the main theorem of this section, stating that the VCGen is correct with respect to our definition of Hoare triples.

Theorem 2 (Soundness of VCGen). *Assume that we have $\mathcal{T}_f(\phi, \psi)$ and*

$$\forall \sigma \in \Sigma, P(\sigma) \Rightarrow \mathcal{T}_a[\![c]\!](\sigma, \phi),$$
$$\forall \sigma \in \Sigma, P(\sigma) \Rightarrow \mathcal{T}_c[\![c]\!](\sigma, \phi, Q).$$

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

Proof. By Theorem 1 and Lemmas 2 and 3. □

Example 5. Consider again the command c_{rec}, procedure environment ψ, and contract environment ϕ of Ex. 2 (presented in Fig. 3). We can apply Theorem 2 to prove its functional correctness expressed by the following Hoare triple:

$$\psi : \{\lambda\sigma.True\}\ \ c_{\text{rec}}\ \ \{\lambda\sigma.\sigma(2) = \sigma(4) \times \sigma(3)\}$$

(see command `com_rec` in file `Examples.v`). □

6 Verification of Relational Properties

In this section, we propose a verification method for relational properties (defined in Section 4) using the VCGen defined in Section 5 (or, more generally, any VCGen respecting Theorem 2). First, we define the notation $\mathcal{T}_{cr}$ for the recursive call of function $\mathcal{T}_c$ on a sequence of commands and memory states:

Definition 7 (Function $\mathcal{T}_{cr}$). *Given a sequence of commands $(c_k)^n$ and a sequence of memory states $(\sigma_k)^n$, a contract environment ϕ and a predicate $\widehat{Q}$ over n states, function $\mathcal{T}_{cr}$ is defined by induction on n as follows.*

– *Basis:* $n = 0$.

$$\mathcal{T}_{cr}([\,],[\,],\phi,\widehat{Q}) \triangleq \widehat{Q}([\,]).$$

– *Inductive:* $n \in \mathbb{N}^*$.

$$\mathcal{T}_{cr}((c_k)^n,(\sigma_k)^n,\phi,\widehat{Q}) \triangleq$$
$$\mathcal{T}_c[\![c_n]\!](\sigma_n,\phi,\ \lambda\sigma'_n.\mathcal{T}_{cr}((c_k)^{n-1},(\sigma_k)^{n-1},\phi,\ \lambda({\sigma'}_k)^{n-1}.\widehat{Q}(({\sigma'}_k)^n))).$$

Intuitively, for $n = 2$, $\mathcal{T}_{cr}$ gives the weakest relational condition that σ_1 and σ_2 must fulfill in order for $\widehat{Q}$ to hold after executing c_1 from σ_1 and c_2 from σ_2: $\mathcal{T}_{cr}((c_1,c_2),(\sigma_1,\sigma_2),\phi,\widehat{Q}) \equiv \mathcal{T}_c[\![c_2]\!](\sigma_2,\phi,\ \lambda\sigma'_2.\mathcal{T}_c[\![c_1]\!](\sigma_1,\phi,\ \lambda\sigma'_1.\widehat{Q}(\sigma'_1,\sigma'_2)))$.

Remark 1. Assume we have $n > 0$, a command c_n, a sequence of commands $(c_k)^{n-1}$, and a sequence of memory states $(\sigma_k)^{n-1}$. From Def. 1, it follows that

$$\forall \sigma_n,\sigma'_n.\ \widehat{P}((\sigma_k)^n) \land (\Vdash \langle c_n,\sigma_n\rangle \xrightarrow{\psi} \sigma'_n) \Rightarrow$$
$$\mathcal{T}_{cr}((c_k)^{n-1},(\sigma_k)^{n-1},\phi,\lambda({\sigma'}_k)^{n-1}.\widehat{Q}(({\sigma'}_k)^n))$$

is equivalent to

$$\psi : \{\lambda\sigma_n.\widehat{P}((\sigma_k)^n)\}c_n\{\lambda\sigma'_n.\mathcal{T}_{cr}((c_k)^{n-1},(\sigma_k)^{n-1},\phi,\lambda({\sigma'}_k)^{n-1}.\widehat{Q}(({\sigma'}_k)^n))\}.$$

Example 6 (Relational verification condition). In order to make things more concrete, we can go back to the relational property $\mathcal{R}_{\text{sw}}$ between two implementations c_{sw1} and c_{sw2} of `swap` defined in Ex. 1 and examine what would be the main verification condition generated by $\mathcal{T}_{cr}$. Let $\widehat{P}$ and $\widehat{Q}$ be defined as in Ex. 3. In this particular case, we have $n = 2$, and ϕ is empty (since we do not have any function call), thus Def. 7 becomes:

$$\mathcal{T}_{cr}((c_{\text{sw1}},c_{\text{sw2}}),(\sigma_1,\sigma_2),\emptyset,\widehat{Q}) = \mathcal{T}_c[\![c_{\text{sw2}}]\!](\sigma_2,\emptyset,\lambda\sigma'_2.\mathcal{T}_c[\![c_{\text{sw1}}]\!](\sigma_1,\emptyset,\lambda\sigma'_1.\widehat{Q}(\sigma'_1,\sigma'_2))).$$

We thus start by applying $\mathcal{T}_c$ over c_{sw1}, to obtain, using the rules of Def. 4 for sequence and assignment, the following intermediate formula:

$$\begin{aligned}
&\mathcal{T}_{cr}((c_{\text{sw1}},c_{\text{sw2}}),(\sigma_1,\sigma_2),\emptyset,\widehat{Q}) = \\
&\quad \mathcal{T}_c(c_{\text{sw2}},\sigma_2,\emptyset, \\
&\qquad \lambda\sigma'_2.\forall\sigma_3,\sigma_5,\sigma_7. \\
&\qquad\quad \sigma_3 = \sigma_1[3/\sigma_1(\sigma_1(1))] \Rightarrow \\
&\qquad\quad \sigma_5 = \sigma_3[\sigma_3(1)/\sigma_3(\sigma_3(2))] \Rightarrow \\
&\qquad\quad \sigma_7 = \sigma_5[\sigma_5(2)/\sigma_5(3)] \Rightarrow \widehat{Q}(\sigma_7,\sigma'_2).
\end{aligned}$$

We can then do the same with c_{sw2} to obtain the final formula:

$$\begin{aligned}
&\mathcal{T}_{cr}((c_{\text{sw1}}, c_{\text{sw2}}), (\sigma_1, \sigma_2), \emptyset, \widehat{Q}) = \\
&\quad \forall (\sigma_k)^8. \\
&\qquad \sigma_4 = \sigma_2[\sigma_2(1)/\sigma_2(\sigma_2(1)) + \sigma_2(\sigma_2(2))] \Rightarrow \\
&\qquad \sigma_6 = \sigma_4[\sigma_4(2)/\sigma_4(\sigma_4(1)) - \sigma_4(\sigma_4(2))] \Rightarrow \\
&\qquad \sigma_8 = \sigma_6[\sigma_6(1)/\sigma_6(\sigma_6(1)) - \sigma_6(\sigma_6(2))] \Rightarrow \\
&\qquad \sigma_3 = \sigma_1[3/\sigma_1(\sigma_1(1))] \Rightarrow \\
&\qquad \sigma_5 = \sigma_3[\sigma_3(1)/\sigma_3(\sigma_3(2))] \Rightarrow \\
&\qquad \sigma_7 = \sigma_5[\sigma_5(2)/\sigma_5(3)] \Rightarrow \widehat{Q}(\sigma_7, \sigma_8).
\end{aligned}$$

Here, σ_k with odd (resp., even) indices result from $\mathcal{T}_c$ for c_{sw1} (resp., c_{sw2}). □

We similarly define a notation for the auxiliary verification conditions for a sequence of n commands.

Definition 8 (Function $\mathcal{T}_{ar}$). *Given a sequence of commands $(c_k)^n$ and a sequence of memory states $(\sigma_k)^n$, we define function $\mathcal{T}_{ar}$ as follows:*

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

Remark 2. For $n > 0$, it trivially follows from Def. 8 that:

$$\mathcal{T}_{ar}((c_k)^n, (\sigma_k)^n, \phi) \equiv \mathcal{T}_a[\![c_n]\!](\sigma_n, \phi) \wedge \mathcal{T}_{ar}((c_k)^{n-1}, (\sigma_k)^{n-1}, \phi).$$

Using functions $\mathcal{T}_{cr}$ and $\mathcal{T}_{ar}$, we can now give the main result of this paper: it states that the verification of relational properties using the VCGen is correct.

Theorem 3 (Soundness of relational VCGen). *For any sequence of commands $(c_k)^n$, contract environment ϕ, procedure environment ψ, and relational assertions over n states $\widehat{P}$ and $\widehat{Q}$, if the following three properties hold:*

$$\mathcal{T}_f(\phi, \psi), \tag{4}$$

$$\forall (\sigma_k)^n, \widehat{P}((\sigma_k)^n) \Rightarrow \mathcal{T}_{ar}((c_k)^n, (\sigma_k)^n, \phi), \tag{5}$$

$$\forall (\sigma_k)^n, \widehat{P}((\sigma_k)^n) \Rightarrow \mathcal{T}_{cr}((c_k)^n, (\sigma_k)^n, \phi, \widehat{Q}), \tag{6}$$

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

In other words, a relational property is valid if all procedure contracts are valid, and, assuming the relational precondition holds, both the auxiliary verification conditions and the main relational verification condition hold. We give the main steps of the proof below. The corresponding Coq formalization is available in file `Rela.v`, and the Coq proof of Theorem 3 is in file `Correct_Rela.v`.

Proof. By induction on the length n of the sequence of commands $(c_k)^n$.

– Induction basis: $n = 0$. By Def. 3, our goal becomes:

$$\psi : \{\widehat{P}\}(c_k)^0\{\widehat{Q}\} \;\equiv\; \widehat{P}([\,]) \Rightarrow \widehat{Q}([\,]).$$

Indeed, by definition of $\mathcal{T}_{cr}$ and Hypothesis (6), $\widehat{P}([\,]) \Rightarrow \widehat{Q}([\,])$ holds.

– Induction step: assuming the result for n, we prove it for $n+1$. So, assume we have a sequence of commands $(c_k)^{n+1}$, relational assertions and environments respecting (4), (5), (6) (stated for sequences of $n+1$ elements). We have to prove $\psi : \{\widehat{P}\}(c_k)^{n+1}\{\widehat{Q}\}$, which, by Def. 3, is equivalent to:

$$\forall(\sigma_k)^{n+1}, ({\sigma'}_k)^{n+1}.\ \widehat{P}((\sigma_k)^{n+1}) \wedge (\bigwedge_{i=1}^{n+1} \Vdash \langle c_i, \sigma_i\rangle \xrightarrow{\psi} {\sigma'}_i) \Rightarrow \widehat{Q}(({\sigma'}_k)^{n+1}). \quad (7)$$

First, we can deduce from Hypothesis (5) and Remark 2:

$$\forall(\sigma_k)^{n+1},\ \widehat{P}((\sigma_k)^{n+1}) \Rightarrow \mathcal{T}_a[\![c_{n+1}]\!](\sigma_{n+1}, \phi), \quad (8)$$

$$\forall(\sigma_k)^{n+1},\ \widehat{P}((\sigma_k)^{n+1}) \Rightarrow \mathcal{T}_{ar}((c_k)^n, (\sigma_k)^n, \phi). \quad (9)$$

By Hypothesis (6) and Def. 7, we have

$$\begin{gathered}\forall(\sigma_k)^{n+1},\ \widehat{P}((\sigma_k)^{n+1}) \Rightarrow \\ \mathcal{T}_c[\![c_{n+1}]\!](\sigma_{n+1}, \phi, \lambda\sigma'_{n+1}.\mathcal{T}_{cr}((c_k)^n, (\sigma_k)^n, \phi, \lambda({\sigma'}_k)^n.\widehat{Q}((\sigma_k)^{n+1}))). \quad (10)\end{gathered}$$

Using (4), (8) and (10), we can now apply Theorem 2 (for an arbitrary subsequence $(\sigma_k)^n$, that we can thus put in an external universal quantifier) to obtain:

$$\begin{gathered}\forall(\sigma_k)^n. \\ \psi : \{\lambda\sigma_{n+1}.\ \widehat{P}((\sigma_k)^{n+1})\}c_{n+1}\{\lambda\sigma'_{n+1}.\ \mathcal{T}_{cr}((c_k)^n, (\sigma_k)^n, \phi, \lambda({\sigma'}_k)^n.\widehat{Q}(({\sigma'}_k)^{n+1}))\}. \quad (11)\end{gathered}$$

Using Remark 1 and by rearranging the quantifiers and implications, we can rewrite (11) into:

$$\begin{aligned}&\forall\sigma_{n+1}, \sigma'_{n+1}.\ \Vdash \langle c_{n+1}, \sigma_{n+1}\rangle \xrightarrow{\psi} \sigma'_{n+1} \Rightarrow \\ &\qquad \forall(\sigma_k)^n.\widehat{P}((\sigma_k)^{n+1}) \Rightarrow \mathcal{T}_{cr}((c_k)^n, (\sigma_k)^n, \phi, \lambda({\sigma'}_k)^n.\widehat{Q}(({\sigma'}_k)^{n+1})). \quad (12)\end{aligned}$$

For arbitrary states σ_{n+1} and σ'_{n+1} such that $\Vdash \langle c_{n+1}, \sigma_{n+1}\rangle \xrightarrow{\psi} \sigma'_{n+1}$, using (4), (9) and (12), we can apply the induction hypothesis, and obtain:

$$\begin{aligned}&\forall\sigma_{n+1}, \sigma'_{n+1}.\ \Vdash \langle c_{n+1}, \sigma_{n+1}\rangle \xrightarrow{\psi} \sigma'_{n+1} \Rightarrow \\ &\qquad \psi : \{\lambda(\sigma_k)^n.\widehat{P}((\sigma_k)^{n+1})\}(c_k)^n\{\lambda({\sigma'}_k)^n.\widehat{Q}(({\sigma'}_k)^{n+1})\}.\end{aligned}$$

Finally, by Def. 3 and by rearranging the quantifiers, we deduce (7). □

Example 7. The relational property of Ex. 3 is proven valid using the proposed technique based on Theorem 3 in file `Examples.v` of the Coq development. For instance, (6) becomes $\forall \sigma_1, \sigma_2. \widehat{P}(\sigma_1, \sigma_2) \Rightarrow \mathcal{T}_{cr}((c_{\mathrm{sw1}}, c_{\mathrm{sw2}}), (\sigma_1, \sigma_2), \emptyset, \widehat{Q})$, where the last expression was computed in Ex. 6. Such formulas—long for a manual proof—are well-treated by automatic solvers.

Notice that in this example we do not need any code transformations or extra separation hypotheses in addition to those anyway needed for the swap functions while both programs manipulate the same locations x_1, x_2, and—even worse—the unknown locations pointed by them can be any locations x_i, $i > 3$. $\square$

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.

Self-compositon [3,30,9] and its derivations [2,31,13] 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 [12] 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 [21,34]. 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],

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

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 [29,25,17,7,19,20,35,24,11,28]. Generation of certificates (in Isabelle) for the BOOGIE verifier is presented in [28]. The certified deductive verification tool WhyCert [17] 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].

To the best of our knowledge, the present work is the first proposal of relational property verification based on verification condition generation realized 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 a method for verifying relational properties using a 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 Hoare triple verification using a verification condition generator (itself formally proved correct) for a simple language with procedure calls and aliasing has been formally established. Our formalization is well-adapted for proving possible optimizations of a VCGen and for using optimized VCGen versions for relational property verification.

This work sets up a basis for the formalization of modular verification of relational properties using verification condition generation. We plan to extend it with more features such as the possibility to refer to the values of variables before a function call in the postcondition (in order to relate them to the values after the call) and the capacity to rely on relational properties during the proof of other properties. Future work also includes an implementation of this technique inside a tool like RPP [9] in order to integrate it with SMT solvers and to evaluate it on benchmarks. The final objective would be to obtain a system similar to the verification of Hoare triples, namely, having relational procedure contracts, relational assertions, and relational invariants. 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. Moreover, we expect termination and co-termination [16],[34] to be used to extend 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
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. 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
12. 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
13. 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
14. 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
15. 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
16. 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
17. 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`
18. 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
19. 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
20. 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
21. 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
22. Kip, I.: Assembly Language for x86 Processors. Prentice Hall Press, 7th edn. (2014)
23. Kirchner, F., Kosmatov, N., Prevosto, V., Signoles, J., Yakobowski, B.: Frama-C: A software analysis perspective. Formal Aspects of Computing **27**(3), 573–609 (2015). https://doi.org/10.1007/s00165-014-0326-7
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 Founda-

tions 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 Detailed Motivating Example

Figure 9 provides a more detailed version of the motivating example presented in Section 1 and Fig. 1. Programs $\mathcal{C}_{\text{sw1}}$ and $\mathcal{C}_{\text{sw2}}$ contain, resp., C functions `sw1` and `sw2`, where pointers `x1` and `x2` are function parameters and variable `x3` in $\mathcal{C}_{\text{sw1}}$ is a local variable. This choice is most natural in C.

Recall that we consider a relational property $\mathcal{R}_{\text{sw}}$:

$\mathcal{R}_{\text{sw}}$: both programs, executed from two states in which `*x1` has the same value for both programs and `*x2` has the same value for both programs, will end up in two states in which each of these locations also has the same value.

To prove the target relational property, a new composed C program $\mathcal{C}_{\text{sw3}}$ with a C function `sw3` is created (see Fig. 9) by composing the code of both functions. To distinguish variables of different programs, variables coming from $\mathcal{C}_{\text{sw1}}$ and $\mathcal{C}_{\text{sw1}}$ are marked, resp., with a suffix "`_1`" or "`_2`".

In the self-composition based approach, the target relational property for the composed program $\mathcal{C}_{\text{sw3}}$ is proved by the Hoare triple $\{P\}\,\mathcal{C}_{\text{sw3}}\,\{Q\}$, where precondition P and postcondition Q are defined in Fig. 9. The definitions are expressed in the ACSL specification language [4]. Lines 5–6 in the definition of P state that each of `*x1` and `*x2` has the same value in the states before the execution of `sw1` and `sw2`. Similarly, lines 5–6 in the definition of Q state the same properties after the execution of `sw1` and `sw2`. However, the precondition must also include additional constraints. Lines 9–11 in the definition of P provide usual preconditions for the swap function `sw1` to be executed correctly: the input pointers must be valid and separated. For instance, validity of pointer `x1_1` means that `*x1_1` can be safely read and written. The separation property `\separated(x1_1,x2_1)` means that the locations `*x1_1` and `*x2_1` are disjoint, that is, do not share any byte[10]. Lines 14–16 in the definition of P provide similar preconditions for the swap function `sw2`. For simplicity, we ignore arithmetic overflows in `sw2`: the specification and verification of properties about the absence of arithmetic overflows are straightforward and orthogonal to the purpose of this paper.

Notice that thanks to the choice of having pointers `x1` and `x2` as function parameters and variable `x3` in $\mathcal{C}_{\text{sw1}}$ as a local variable, for this version we do not need to state explicitly other separation hypotheses stating that `x1` and `x2` do not

[10] Notice that this separation property is stronger in C than the non-equality constraint `x1_1 != x2_1`, which does not exclude that both locations have some bytes in common (if the pointers are not aligned). For simplicity, byte-related data representation and alignment constraints are not modeled in $\mathcal{L}$, where the separation can be simply represented by non-equality constraints. This does not restrict the representativity of $\mathcal{L}$ for the purpose of our study.

refer to `x1`, `x2` and, for `sw1`, `x3` themselves. Indeed, these separation hypotheses[11] are already ensured by the fact that `x1`, `x2` and, for `sw1`, `x3` are allocated during the call to the C function (and the pointers `x1` and `x2` are valid before the call).

The aforementioned parts of P and Q naturally come from the relational property $\mathcal{R}_{\text{sw}}$ and the preconditions of the considered functions: in this sense, they are expected. However, they are not sufficient: in a real-life language with possible aliasing like C, to model the behavior of both programs correctly within the composed program and to prove the expected relational property, additional separation hypotheses between the variables coming from both programs $\mathcal{C}_{\text{sw1}}$ and $\mathcal{C}_{\text{sw2}}$ are required. They are expressed by lines 20–23 in the definition of P in Fig. 9.

Such additional separation hypotheses become even more complex for real-life programs, in particular in C, with a greater number of pointers and/or in the presence of multiple pointers (such as double pointers, for instance, `int **p`). Indeed, the required separation hypotheses for the composed program rapidly become extremely hard to specify (or to generate) in order to ensure a sound proof of relational properties on the composed program.

With this definition of precondition P and postcondition Q, the code of $\mathcal{C}_{\text{sw3}}$ can be proved to satisfy its contract by the deductive verification plugin Wp of Frama-C [23].

[11] In $\mathcal{L}$, for simplicity, we consider only global variables, therefore, in the counterparts c_{sw1} and c_{sw2} in language $\mathcal{L}$, these additional separation hypotheses must be explicit (as we show in Ex. 3 and Fig. 6). This slight difference of modeling is intentional in order to show the most natural version of these functions in C with function parameters and local variables rather than with global variables only.

```
//C program $\mathcal{C}_{\text{sw1}}$ :
void sw1(int *x1,int *x2){
  int x3;
  x3  = *x1;
  *x1 = *x2;
  *x2 = x3;
}
```

```
//C program $\mathcal{C}_{\text{sw2}}$ :
void sw2(int *x1,int *x2){
  *x1 = *x1 + *x2;
  *x2 = *x1 - *x2;
  *x1 = *x1 - *x2;
}
```

$\{P\}$

```
//Composed C program $\mathcal{C}_{\text{sw3}}$ :
void sw3(int *x1_1,int *x2_1
    int *x1_2,int *x2_2){

//Code simulating $\mathcal{C}_{\text{sw1}}$ :
  int x3_1;
  x3_1  = *x1_1;
  *x1_1 = *x2_1;
  *x2_1 = x3_1;

//Code simulating $\mathcal{C}_{\text{sw2}}$ :
  *x1_2 = *x1_2 + *x2_2;
  *x2_2 = *x1_2 - *x2_2;
  *x1_2 = *x1_2 - *x2_2;
}
```

$\{Q\}$

```
//$P$ is defined as follows:

//Relation between initial
//values of $\mathcal{C}_{\text{sw1}}$ and $\mathcal{C}_{\text{sw2}}$ :
*x1_1 == *x1_2 &&
*x2_1 == *x2_2 &&

//Preconditions for $\mathcal{C}_{\text{sw1}}$ :
\valid(x1_1) &&
\valid(x2_1) &&
\separated(x1_1,x2_1) &&

//Preconditions for $\mathcal{C}_{\text{sw2}}$ :
\valid(x1_2) &&
\valid(x2_2) &&
\separated(x1_2,x2_2) &&

//Extra hypotheses for
//a correct simulation by $\mathcal{C}_{\text{sw3}}$ :
\separated(x1_1,x1_2) &&
\separated(x1_1,x2_2) &&
\separated(x2_1,x1_2) &&
\separated(x2_1,x2_2)
```

```
//$Q$ is defined as follows:

//Relation between resulting
//values of $\mathcal{C}_{\text{sw1}}$ and $\mathcal{C}_{\text{sw2}}$ :
*x1_1 == *x1_2 &&
*x1_1 == *x1_2
```

Figure 9: Two C programs $\mathcal{C}_{\text{sw1}}$ and $\mathcal{C}_{\text{sw2}}$ swapping `*x1` and `*x2` and the Hoare triple $\{P\}\,\mathcal{C}_{\text{sw3}}\,\{Q\}$ to prove a relational property between them using their composition in C program $\mathcal{C}_{\text{sw3}}$, as well as definitions of precondition P and postcondition Q of $\mathcal{C}_{\text{sw3}}$.

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

Figure 10: Evaluation of arithmetic and Boolean expressions in $\mathcal{L}$.

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

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

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

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

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

Figure 11: Operational semantics of commands in $\mathcal{L}$.

B Complete Semantics of Language $\mathcal{L}$

B.1 Evaluation of Arithmetic and Boolean Expressions in $\mathcal{L}$

We provide a complete list of rules for evaluation of arithmetic and Boolean expressions in $\mathcal{L}$ in Fig. 10. 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].

B.2 Operational Semantics of Commands in $\mathcal{L}$ in $\mathcal{L}$

We provide a complete operational semantics of commands in $\mathcal{L}$ in Fig. 11.