Relational properties for specification and verification of C programs in Frama-C

Lionel Blatter

université PARIS-SACLAY

CentraleSupélec

list CEA tech

NNT : 2019SACLC065

Thèse de doctorat

Relational properties for specification and verification of C programs in Frama-C

Thèse de doctorat de l'Université Paris-Saclay
préparée à CentraleSupélec

École doctorale n°573 Interfaces : Approches Interdisciplinaire : Fondements, Applications et Innovations
Spécialité de doctorat : Informatique

Thèse présentée et soutenue à Palaiseau, le 26 septembre 2019, par

Lionel Blatter

Composition du jury :

Jean-Christophe Filliâtre Directeur de Recherche, Université Paris Sud	Président
Alain Giorgetti Maître de conférences HDR, Université de Franche-Comté	Rapporteur
Mattias Ulbrich Chercheur, Karlsruhe Institute of Technology	Rapporteur
Mihaela Sighireanu Maître de conférences HDR, Université Paris Diderot	Examinatrice
Nikolai Kosmatov Ingénieur-chercheur HDR, CEA List	Co-encadrant
Pascale Le Gall Professeur, CentraleSupélec	Directrice de thèse
Virgile Prevosto Ingénieur-chercheur, CEA List	Co-encadrant

Remerciements

Tout d'abord je voudrais remercier Alain Giorgetti et Mattias Ulbrich d'avoir accepté de rapporter ma thèse. Je remercie également l'examinatrice Mihaela Sighireanu et le président du jury Jean-Christophe Filliâtre. Je remercie l'ensemble des membres du jury d'avoir accepté d'assister à la présentation de ce travail et de m'avoir permis de corriger un certain nombre d'imprécisions et d'erreurs contenus dans le manuscrit.

Je voudrais remercier tout particulièrement ma directrice de thèse Pascale Le Gall et mes encadrants du CEA Nikolai Kosmatov et Virgile Prevosto. Ils ont toujours été disponible, à l'écoute de mes nombreuses questions, et toujours intéressé à l'avancée de mes travaux. Les nombreuses discussions que nous avons eues, ainsi que leurs conseils, m'ont toujours été utile. Leurs rigueurs, leur esprit d'analyse et leurs capacités à résoudre des problèmes et leurs connaissances, m'ont permis de progresser. Enfin, leurs relectures et corrections de cette thèse ont été très appréciables.

Je remercie toutes les personnes qui ont permis de créer un cadre agréable au CEA. En premier lieu mes voisins de bureau : Bernard, Jean-Christophe, Vincent et Christophe. Je remercie tous les doctorants du laboratoire LSL : David, Allan, Steven, Hugo, Benjamin, Quentin, Alexandre, Yaëlle, Maxime, Lesly, Virgile et Frédéric. Je remercie également des personnes du laboratoire LISE : Imen et Ngo Minh Thang. Je voudrais remercier Loïc, François, Patrick et Nicky pour leur aide. Je remercie également les membres du laboratoire MICS de CentraleSupélec et du LRI de l'Université Paris-Sud qui m'ont accueilli pendant ma dernière année de thèse.

Je remercie enfin celles et ceux qui me sont chers. Leurs attentions et encouragements m'ont accompagné tout au long de ces années. Je remercie mes parents pour leur soutien moral et matériel.

Résumé étendu en français

Les techniques de vérification déductive fournissent des méthodes puissantes pour la vérification formelle des propriétés exprimées dans la Logique de Hoare. Dans cette formalisation, également connue sous le nom de sémantique axiomatique, un programme est considéré comme un transformateur de prédicat, où chaque programme c exécuté sur un état vérifiant une propriété P conduit à un état vérifiant une autre propriété Q.

Cependant, il est fréquent qu'on veuille parler d'une propriété mettant en jeu l'exécution de plusieurs fonctions, ou comparer les résultats d'une même fonction sur différents paramètres. Des groupes de fonctions sont fréquemment liés par des spécifications algébriques précisant leurs relations. On parle dans ce cas de propriétés relationnelles, liant un ensemble de programmes à deux propriétés. Plus précisément, une propriété relationnelle est une propriété concernant n programmes $c_1, ..., c_n$, indiquant que si chaque programme c_i commence dans un état s_i et termine dans un état s'_i tel que $P(s_1, ..., s_n)$ soit vérifié, alors $Q(s'_1, ..., s'_n)$ est vérifié. Ainsi, les propriétés relationnelles invoquent un nombre fini d'exécutions de programmes éventuellement dissemblables.

Il en résulte que les méthodes déductives classiques se prêtent mal à la spécification et à la vérification de telles propriétés. On retrouve dans la littérature différentes méthodes qui permettent de répondre au problème de la vérification de propriétés relationnelles. Les méthodes les plus classiques de vérification déductive de propriétés relationnelles sont : la Logique de Hoare Relationnelles qui est une extension de la sémantique axiomatique traditionnel et permet de vérifier des propriétés concernant 2 programmes. L'approche par Self-Composition et par Produit de Programme, des approches consistantes à réduire le problème de vérification des propriétés relationnelles portant sur n programmes à un problème de vérification standard de programme. L'approche est fondée sur la construction d'un nouveau programme simulant les appels des programmes reliés par la propriété. Ces méthodes présentent la limitation de ne pas supporter les appels de procédures et de ne pas permettre d'utiliser les propriétés relationnelles comme hypothèses. Cette thèse apporte deux solutions à cette problématique. Les deux approches permettent de prouver une propriété relationnelle et de l'utiliser comme hypothèse dans des vérifications ultérieures.

La première solution proposée consiste à étendre la méthode de Self-Composition afin de pouvoir vérifier et utiliser les propriétés relationnelles dans le contexte d'appels de procédures. L'utilisation des propriétés relationnelles est basée sur une axiomatisation en logique du premier ordre des propriétés. Cette solution est implémentée dans le contexte du langage de programmation C, du langage de spécification ACSL et du plugin de vérification déductive WP, dans la

plate-forme FRAMA-C. Nous avons étendu le langage de spécification ACSL afin de pouvoir exprimer les propriétés et ainsi pouvoir implémenter notre méthode de vérification dans un plugin Frama-C nommé RPP. L'outil permet de spécifier une propriété relationnelle, de la prouver et de l'utiliser comme hypothèse dans la preuve d'autres propriétés en utilisant la vérification déductive classique. Afin de tester notre démonstrateur, nous avons créé un ensemble de benchmarks afin de valider l'outil. L'outil nous permet de traiter un large ensemble de propriétés relationnelles de manière automatique, et permet de réutiliser les propriétés dans le contexte d'autres preuves.

Cependant un certain nombre de point ont pu être remarqués lors du développement et de l'évaluation de l'outil : l'approche de Self-Composition nécessite un ensemble de renommage fastidieux à réaliser et impose des restrictions sur les pointeurs. Ces limitations nous ont poussés à élaborer une seconde approche pour la vérification de propriété relationnelle utilisant les caractéristiques d'un générateur d'obligation de preuves. Cette nouvelle méthode permet de s'affranchir des contraintes imposées par les méthodes basées sur une approche de Self-Composition, et tolère des axiomatisations de propriétés relationnelles alternatives. Une implémentation partielle de cette nouvelle méthode est proposée dans l'outil RPP.

Contents

Chapter 1

Introduction

Today's software is characterized by increasing complexity and ever greater expansion over all areas of society. Traditionally, software is known to be found in desktops, laptops, smartphones. With the development of the Internet of Things, it also appears in everyday objects, like door locks, smart speakers, ... In addition, software, which is increasingly complex, tends to contain errors. Thus, everyone has experienced situations when software crashes or shows unwanted behaviour. The presence of those bugs can have several explanations.

The same program is often run on different architectures with different specifications, and in collaboration with other programs. For example, the Linux kernel runs on most architectures available on the market, without losing the support of previous platforms (except for some really obsolete architectures). It is possible to run the latest version of the Kernel on a twenty-year-old computer [K+14]. Since it is usually not possible in practice to test all architectures, it is difficult for developers to guarantee that their programs will run flawlessly on any possible machine. At best, the program is available with the guarantee that it was thoroughly tested and should work on a sensible set of architectures.

Moreover, depending on the context, bugs may be acceptable. The developers are mainly focused on the functional part. The task to find and report non-critical issues might be left to users. In the case of Open-Source software, bugs might also be fixed by contributors. However, this may result in inconsistent code quality or new bugs. Even worse, programs are sometimes written as quickly as possible to lower development costs or to save time. This implies that programs are released in a poor state where bugs are inevitable.

On the other hand, the use of software in critical domains such as energy, transportation, health, defense, etc., requires different development strategies. Indeed, in such systems, bugs can have extremely severe consequences for costly equipments or human lives. One of the most well-known examples is the first flight of Ariane 5 which ended in the loss of the launcher [Lio96]. Thus, critical embedded software often needs to be assessed against a certain number of criteria, depending on criticality level and application domain. Those criteria include safety and security, and result in a strong need for analysis and verification, in particular, for powerful and expressive theories, capable to express and treat ever more complex properties of software.

1.1 Formal Verification

Formal verification is devoted to provide strong mathematical grounds for reasoning on programs. Since programs are written in a programming language with a well defined semantics (at least theoretically), it is possible to consider a mathematical model of the program. Thus, mathematical analysis can contribute to reliability and robustness. For example, properties such as the absence of runtime errors or absence of dead-code can be verified. One of the specificities of formal verification is that the program is not executed with a given input, but a static analysis is performed on the program code. Different theoretical foundations exist to provide a formal analysis. The following list mentions only the most notable ones:

- Model checking [CES86]: Verification of temporal properties by an exhaustive space exploration on an abstract model of the program's semantics.

- Symbolic execution [Kin76]: Verification of properties on symbolic execution of a program *i.e.* the execution proceeds as in a normal execution except that variables are mapped to symbolic expressions based on fresh symbols representing arbitrary input values.

- Abstract interpretation [CC77]: Verification of properties on an abstract execution of a program *i.e.* variables are mapped to an abstraction (domain) that over-approximates the set of concrete values that they can take during any possible concrete execution (in case of integers, a variable can for instance be abstracted by a sign or an interval).

- Deductive verification [Hoa69]: Verification of the adequacy between a specification and an implementation by transforming the program and specification into formulas that need to be verified.

Deductive verification, model-checking and symbolic execution are methods where the expected properties can be precisely specified and proven. Abstract interpretation supports a limited set of properties: the properties that can be defined in the context of the chosen abstract domain(s).

All those methods require some manual work. Abstract interpretation requires the choice of the right domain and the interpretation of the results. Model checking and symbolic execution require the definition of bounds (how many times loops are unrolled, maximum size of arrays, ...). Deductive verification requires the addition of specifications inside the source code (notably to model the behaviour of loops and functions). Using those additional pieces of information makes it possible to abstract the complex parts (functions and loops) which are often a pitfall for techniques like model checking and symbolic execution due to state space explosion. Thus, deductive verification is a modular and scalable verification approach and has similarities with verification of properties using proof assistants [BC04] like Coq [Tea17], where intermediate properties can be defined and proven separately.

Thus, deductive verification, as introduced above, will be the basis of the work described in this thesis.

1.2 Deductive Verification

Deductive verification techniques provide powerful methods for formal verification of properties expressed in Hoare Logic [Flo67, Hoa69]. In this formalization, also known as axiomatic semantics, a program is seen as a predicate transformer, where a program c executed on a state verifying a property P leads to a state verifying another property Q. This is summarized in the form of a *Hoare triple*:

$$\{P\}c\{Q\}$$

In this setting, P and Q refer to states before and after a single execution of a program c. Properties P and Q are commonly called *precondition* and *postcondition* respectively. For example, we can consider the following triple:

$$\{x_1 = 10\}x_1 := x_1 + 1\{x_1 = 11\}$$

This triple states that for an initial state where location x_1 contains value 10, executing a program that increases the value of location x_1 by one ends in a state where location x_1 contains value 11. This triple is clearly valid and can be proven using the Hoare proof system [Hoa69] the weakest precondition calculus [Dij68] or verification condition generation [Gor88].

Deductive verification can of course handle more complex cases, and it has been extensively studied. Different tools exist for performing verification on different programming languages. This includes for instance Spec# [BLS05] for C#, Dafny [LW14] for Dafny, OpenJML [Cok14], Verifast [JSP10], KeY [ABB+16], and Krakatoa [FM07] for Java, Why3 [FP13] for WhyML, WP plugin of Frama-C [KKP+15] and Verifast [JSP10] for C, Spark2014 [KCC+14] for Ada.

On Figure 1.1, we show an example of a function computing the factorial of an integer n, written in C. The function is equipped with annotations written in the ACSL specification language [BCF+13]. We recognize the pre- and post-condition on lines 7–8. The function requires that integer n is non-negative in the state before the execution (line 7). If the precondition is satisfied, we specify that the execution ends in a state where location `\result` (the return of the function) contains the factorial of n by stating what n and location `\result` satisfy the predicate `isFact` (post-condition line 8). This predicate is defined in an axiomatic definition lines 1–5 and is a copy of the usual mathematical definition of factorial. Moreover, loop invariants, *i.e.* properties that are true after each loop iteration and at loop entry, are used to summarize the behaviour of the loop, lines 12–13. Here, we state that variable x is between 0 and n and that factorial of x times y is equal to factorial of n. In addition, a frame rule is specified line 14, defining the loop side effects. Finally, for the proof of termination, a loop variant is defined line 15. Note that for now we ignore issues arising from potential arithmetic overflows, as they can be dealt with separately.

As shown on Figure 1.1, functions are specified and verified separately, following the concept of design-by-contract [Mey97] and generally using verification condition generation [Gor88]. For a given function f, any individual call to f can be proven to respect the *contract* of f, that is, basically an implication: if the given *precondition* is true before the call, the given *postcondition* is true after its execution. As mentioned earlier, the success of deductive verification is due to the fact that those contracts are used to summarize the behavior of some parts of the programs. This approach is commonly called modular deductive verification.

```
1 /*@ axiomatic Fact {
2      predicate isFact(integer n, integer fact);
3      axiom Fact_1: isFact(0,1);
4      axiom Fact_2: ∀ integer n,r1; n > 0 ==> isFact(n-1,r1) ==> isFact(n,n*r1);
5 }*/
6
7 /*@ requires n >= 0;
8   @ ensures isFact(n,\result);*/
9 int fact (int n) {
10   int y = 1;
11   int x = n;
12   /*@ loop invariant 0 <= x <= n;
13     @ loop invariant ∀ integer r1; isFact(x,r1) ==> isFact(n,y*r1);
14     @ loop assigns x,y;
15     @ loop variant x;*/
16   while (x > 1) {
17     y = y * x;
18     x = x - 1;
19   };
20   return y;
21 }
```

Figure 1.1 – C function computing factorial and equipped with ACSL annotation

Figure 1.2a and Figure 1.2b show two C functions, `abs` computing the absolute value of `x` and `max` computing the maximum between two values `x` and `y`. Both functions have their own specification, written in ACSL, which refers to the lexical scope of the function. Notice that the specification of `abs` uses a more advanced contract style based on the notion of behavior to distinguish the contexts of use of the function (a deeper explanation is provided in Chapter 2).

However, we may want to be able to say more about those functions. For example, we might want to state a property like:

```
∀ integer x,y; max(x,y) == (x+y+abs(x - y))/2.
```

But in the context of classical deductive verification we cannot express such properties as one cannot express properties that refer to two distinct executions of a program c, or properties relating executions of different programs c_1 and c_2 (in the present case max and abs).

```
1 /*@ requires x > INT_MIN;
2     assigns  \nothing;
3     behavior pos:
4       assumes x >= 0;
5       ensures \result  == x;
6     behavior neg:
7       assumes x < 0;
8       ensures \result  == -x;*/
9 int abs (int x){
10   return (x >= 0) ? x : (-x);
11 }
```

(a) C function computing the absolute value

```
1 /*@ assigns \nothing;
2     ensures \result >= y && \result >= x;
3     ensures \result == y || \result == x;*/
4 int max(int x,int y){
5   return (x >= y) ? x : y;
6 }
```

(b) C function computing the maximum between two values

Figure 1.2 – Two annotated functions

As we will see in the next sections, such properties, that are generically called *relational properties*, occur quite regularly in practice. Hence, it is desirable to provide an easy way to specify them and to verify that implementations are conforming to such properties.

1.3 Relational Properties

Relational properties can be seen as an extension of axiomatic semantics. But, instead of linking one program to two properties, relational properties link n programs to two properties. More precisely, a relational property is a property about n programs $c_1, ..., c_n$, stating that if each program c_i starts in a state s_i and ends in a state s'_i such that $P(s_1, ..., s_n)$ holds, then $Q(s'_1, ..., s'_n)$ holds. Thus, relational properties invoke any finite number of executions of possibly dissimilar programs.

1.3.1 Notations

Different notations exist for relational properties. The most common, proposed by Benton in [Ben04], describes relational properties linking two programs by $\{P\}c_1 \sim c_2\{Q\}$. As Benton's work focuses on comparing equivalent programs, using symbol $\sim$ to denote a relation of similarity between two programs is quite natural. As multiple states are combined, tags are used to make distinctions. For example, let us consider the following quadruple:

$$\left\{ \begin{array}{c} x_2\langle 1\rangle = x_2\langle 2\rangle \\ \wedge \\ x_3\langle 1\rangle = x_3\langle 2\rangle \end{array} \right\} \begin{array}{l} x_1 := -x_2; \\ x_3 := x_3 - x_1; \\ x_1 := -x_1 \end{array} \langle 1\rangle \sim \begin{array}{l} x_1 := x_2; \\ x_3 := x_3 + x_1 \end{array} \langle 2\rangle \left\{ \begin{array}{c} x_1\langle 1\rangle = x_1\langle 2\rangle \\ \wedge \\ x_2\langle 1\rangle = x_2\langle 2\rangle \\ \wedge \\ x_3\langle 1\rangle = x_3\langle 2\rangle \end{array} \right\} \quad (1.1)$$

The quadruple links an optimized version of a program (right of $\sim$) to its original version (left of $\sim$). It states that both programs (with tag $\langle 1\rangle$ on the left and tag $\langle 2\rangle$ on the right) executed from two states named $\langle 1\rangle$ and $\langle 2\rangle$ verifying $x_2\langle 1\rangle = x_2\langle 2\rangle \wedge x_3\langle 1\rangle = x_3\langle 2\rangle$ (the value of x_2 is the same in both states and the value of x_3 is the same in both states), lead to two states verifying $x_1\langle 1\rangle = x_1\langle 2\rangle \wedge x_2\langle 1\rangle = x_2\langle 2\rangle \wedge x_3\langle 1\rangle = x_3\langle 2\rangle$ (the value of x_1 is the same in both states, the value of x_2 is the same in both states and the value x_3 is the same in both states).

An alternative, but equivalent, notation has been proposed in [Yan07]:

$$\{P\} \left(\begin{array}{c} c_1 \\ c_2 \end{array} \right) \{Q\}$$

Although the benefit of this notation is the absence of symbol $\sim$ that can be confusing for properties that do not express program similarity and link more than two programs, we prefer the first notation. Most relational properties in this thesis do not exceed two programs and ambiguity about the meaning of a property is resolved by an appropriate explanation.

1.3.2 Examples

Relational properties are not uncommon in practice. Using Benton's notation, we can present a few more examples taken from case studies:

1. Verification of monotonic functions [BBC13] in an industrial case study on smart sensor software.

   Suppose we have a program c implementing a monotonic functions f ($\forall\, x, y.\ x < y \Rightarrow f(x) < f(y)$). We call x_{param} the program entry and x_{res} the result.

   $$\left\{\ x_{param}\langle 1\rangle < x_{param}\langle 2\rangle\ \right\}\, c\langle 1\rangle \sim c\langle 2\rangle \left\{\ x_{res}\langle 1\rangle < x_{res}\langle 2\rangle\ \right\}$$

   Executing two instances of c on two states $\langle 1\rangle$ and $\langle 2\rangle$ satisfying $x_{param}\langle 1\rangle < x_{param}\langle 2\rangle$, ends in two states satisfying $x_{res}\langle 1\rangle < x_{res}\langle 2\rangle$.

2. Verification of properties on voting rules [BBK$^+$16].

   Suppose we have a program $voting$ implementing a voting rule f. We call x_{param} the program entry, corresponding to a sequence of ballots, and x_{res} the program result, corresponding to the result of applying the voting rule f to the sequences of ballots in x_{param} (the winner according to rule f). We can define the following relational property, called *anonymity*, assuming the existence of a predicate $permut(a, b)$ being true if a and b are sequences of ballots and b is a permutation of a:

   $$\left\{\ permut(x_{param}\langle 1\rangle, x_{param}\langle 2\rangle)\right\}\, voting\langle 1\rangle \sim voting\langle 2\rangle \left\{\ x_{res}\langle 1\rangle = x_{res}\langle 2\rangle\ \right\}$$

   Applying the voting rule to a sequence of ballots and a permutation of the same sequence of ballots ends in the same result, *i.e* regardless of the order in which the ballots are passed to the voting function, the result is the same.

3. Verification of properties on comparator functions [SD16].

   Suppose we have a comparator function f, comparing x and y and returning -1 if $x < y$, 1 if $x > y$ and 0 if $x = y$. Three typical properties can be defined on f:

   - anti-symmetry ($\forall\, x, y.\ f(x, y) = -f(y, x)$),
   - transitivity ($\forall\, x, y, z.\ f(x, y) > 0 \wedge f(y, z) > 0 \Rightarrow f(x, z) > 0$),
   - extensionality $\forall\, x, y, z.\ f(x, y) = 0 \Rightarrow f(x, z) = f(y, z)$.

   Suppose we have a comparator program $compare$, implementing a compare function f. We call x_{param_1} and x_{param_2} the program entries, and x_{res} the programs result. Anti-symmetry can for instance be defined as follows:

   $$\left\{\begin{array}{c} x_{param_1}\langle 1\rangle = x_{param_2}\langle 2\rangle \\ \wedge \\ x_{param_2}\langle 1\rangle = x_{param_1}\langle 2\rangle \end{array}\right\}\, compare\langle 1\rangle \sim compare\langle 2\rangle \left\{\ x_{res}\langle 1\rangle = -x_{res}\langle 2\rangle\ \right\}$$

   More details about those three properties can be found in Chapter 7.

4. Verification of secure information flow [BDR11] properties.

   In case of secure information flow, the main relational property of interest is known as *non-interference*. As program variables can be composed of high security variables $H = \{x_{h_1}, ..., x_{h_n}\}$ and low security variables $L = \{x_{l_1}, ..., x_{l_m}\}$, a program can be said non-interferent if and only if any execution in which the low security variables have the same initial values will result in the same values for the low security variables at the end of the execution, regardless of what the high level inputs are. This definition of non-interference can be expressed by a relational property (for a given program c):

$$\left\{ \begin{array}{c} x_{l_1}\langle 1\rangle = x_{l_1}\langle 2\rangle \\ \wedge \\ ... \\ x_{l_n}\langle 1\rangle = x_{l_n}\langle 2\rangle \end{array} \right\} c\langle 1\rangle \sim c\langle 2\rangle \left\{ \begin{array}{c} x_{l_1}\langle 1\rangle = x_{l_1}\langle 2\rangle \\ \wedge \\ ... \\ x_{l_n}\langle 1\rangle = x_{l_n}\langle 2\rangle \end{array} \right\}$$

Although these properties are relational, some of them are called k-safety properties in the literature [SD16]. A k-safety property is a relational property linking k instances of the same program and mostly states a safety property. Example 4 above is a 2-safety property.

1.3.3 Verification of Relational Properties

Different deductive verification methods exist for proving valid relational properties. Most notable, based on proof systems, are Relational Hoare Logic [Ben04] (used for Example 1.1), Relational Separation Logic [Yan07], and Cartesian Hoare Logic [SD16] supporting specifically k-safety properties (used for Example 3). As Cartesian Hoare Logic and Relational Separation Logic are similar to Relational Hoare Logic, we will focus in the sequel on Relational Hoare Logic.

Alternative approaches relying on the existing Hoare Logic are Self-Composition [BDR11] (used for Examples 1, 2 and 4) and Product Programs [BCK16, BCK11]. Those methods propose an approach to prove relational properties by reducing the verification of relational properties to a standard deductive verification problem. The benefit of such an approach is that existing tools can be used for the verification. We propose in Chapter 4 a more thorough presentation of Relational Hoare Logic, Self-Composition and Product Program.

Beyond deductive verification, relational properties and more precisely k-safety properties are used as an oracle [ZS13] in testing. However, in this context, k-safety properties are called metamorphic relations. A method using abstract interpretation is presented in [AGH$^+$17] for verification of k-safety properties and a Relational Symbolic Execution is proposed in [FCG17]. Finally, a method using model checking, proposed in [YVS$^+$18], is one example showing that Self-composition is not limited to deductive verification.

1.4 Motivations

Although all methods presented in Section 1.3 propose a verification approach for relational properties, none has an efficient support for function or procedure calls or provides a way to use

proven properties. Works described in [EMH18] (for k-safety properties) and [KKU18] propose some solutions for supporting function calls, but with limited modularity. In other words, there is no support for a *modular* verification of relational properties.

Lack of support for relational properties in verification tools is discussed in [BBC13]. It is often required to perform Self-Composition techniques manually, which is relatively tedious, error-prone, and does not provide a completely automated link between three key components:

(i) the specification of the property,

(ii) the proof that the implementation satisfies the property,

(iii) the ability to use the property as hypothesis in other proofs of relational as well as non-relational properties.

1.5 Contribution

This thesis brings a solution to the modular verification of relational properties. Like most papers mentioned in Section 1.3, we focus on imperative languages. Thus, we propose to model our approach using a simple imperative *while* language borrowed from [Win93]. However, this simple language from [Win93] does not have program calls. Thus, we propose our own extension, called R-WHILE language (Recursive While language), where function calls are without explicit parameters and return value, similar to what is proposed in [AdO09]. Moreover, our contribution requires some expressiveness concerning the specification language. The R-WHILE language is therefore equipped with labels and predicates, as we will see in Chapter 5.

In the context of the R-WHILE language, we design two techniques for proving and using relational properties in deductive verification. One approach is based on Self-Composition, which provides a powerful theoretical approach to prove relational properties, while remaining extremely simple and relying on existing verification approaches (Chapter 4). The other approach uses the properties of verification condition generators for proving relational properties, and require no code transformation, in oposition to Self-Composition.

To address the absence of tools supporting relational properties, we propose an implementation of our approaches. Our implementation is performed in the context of the C programming language, the FRAMA-C [KKP$^+$15] platform, the ACSL specification language [BCF$^+$13] and the deductive verification plugin WP. The new tool takes the form of a Frama-C plug-in called RPP (Relational Property Prover) and allows the user to specify a relational property, to prove it using classic deductive verification, and to use it as hypothesis in the proof of other properties that may rely on it.

More specifically, RPP provides an extension to ACSL for expressing relational properties. In the case of the Self-Composition approach, the extended annotations are translated into standard ACSL annotations and C code such that the WP plugin can be used. This is a typical approach in the FRAMA-C collaborative framework.

In the case of the second verification approach, RPP communicates directly with the WP plugin without going through a code transformation.

RPP is evaluated over a set of illustrative examples. We have performed experiments with runtime checking of relational properties and counterexample generation when a property cannot be proved.

1.6 Outline

This thesis is structured as follows.

Chapter 2 presents in more detail FRAMA-C, ACSL and WP. We also provide a more detailed example of how to prove a program using the WP proof system. This chapter sets all required components for the next chapters.

Chapter 3 introduces notations that are used in this thesis and the R-WHILE language. We present the language syntax and semantics using denotational semantics. We also provide a basic axiomatic semantic for the language.

In Chapter 4, we present in more detail the concepts of Relational Hoare Logic, Self-Composition and Product Program and discuss their benefits and limitations. As in the literature those methods are typically presented using a basic *while* language, we use the R-WHILE language introduced in Chapter 3.

Chapter 5 introduces extensions added to the R-WHILE, language introduced in Chapter 3. These extensions consist in adding labels and predicates in order to improve expressiveness and write more interesting properties. We also present a verification condition generator for proving validity of Hoare Triples based on the extended syntax.

In Chapter 6, we present our method for proving relational properties in a modular way, using basic self-composition, as presented in Chapter 4. We explain the method by using the language shown in Chapters 3 and 5 and examples of proofs using verification approaches presented in Chapters 3 and 5.

Chapter 7 describes the implementation of the approach described in Chapter 6 in the context of the C language and the FRAMA-C platform. First we present the extension we added to the original ACSL language presented in Chapter 2. We then review the code transformation of Chapter 6 in the context of FRAMA-C (the C programming language and ACSL). Finally, we present different case studies of relational property verification using the method of Chapter 6. Part of this work has been published in [BKGP17, BKG+18]. The tool [1] and case studies [2] are available online.

Chapter 8 presents an alternative to self-composition for the deductive verification of relational properties. We also present a refinement of the method for using relational properties shown in Chapter 6.

Chapter 9 concludes and presents some perspectives.

[1] https://github.com/lyonel2017/Frama-C-RPP

[2] https://github.com/lyonel2017/RPP-Examples-TAP-2018

Chapter 2

Frama-C

In this chapter, we present FRAMA-C, a tool that allows users to machine-check formally expressed properties of programs. As we mention in Section 1.5, we chose to implement our approach in the context of the C programming language and the FRAMA-C [KKP+15] platform. Thus, the chapter is devoted to this language and tool.

The C programming language is a well-known, old language, still widely used, powerful, and well-adapted for a significant number of applications (notably operating systems, embedded systems). The fact that it is difficult to write code without bugs in C and that the language is widely used in critical areas makes it an interesting target language for verification tools.

FRAMA-C [1] is one of these tools. It is an open-source platform dedicated to the analysis of source code written in the C programming language. The FRAMA-C platform combines several analysis techniques in the form of interconnected plug-ins. A list of available techniques can be found on the web page. We focus in this thesis on the deductive part of the tool, and more precisely on the WP plugin that verifies that an implementation complies with a set of formal specifications written in a dedicated language, ACSL, described in Section 2.1. A detailed example of use of the WP plugin is shown in Section 2.2.

As the C language exposes many notoriously awkward constructs, we consider in this thesis only a small part of the C language syntax, mostly equivalent to the R-WHILE language presented in Chapter 3. We assume in the following that the reader has a minimal knowledge of the C programming language.

2.1 The ACSL specification language

The ANSI/ISO C Specification Language (ACSL) [BCF+13] is a formal specification language for the C programming language. It aims at specifying behavioral properties of C source code (a Behavioral Interface Specification Language) and is agnostic towards the underlying verification techniques, *i.e.* tries to remain purely at the specification level.

[1] `https://frama-c.com/`

ACSL is similar to the general design-by-contract [Mey97] principle implemented in the Eiffel language. Other specification languages were also used as inspiration. For example, the specification language of the Caduceus tool [FM07] for C programs, or the Java Modeling Language (JML) [LRL+00] for Java source code.

In the following section we introduce all the concepts of ACSL we need in the manuscript. For a complete presentation of ACSL, the reader is invited to consult the manual [BCF+13].

```
1 #include <limits.h>
2 /*@ requires x > INT_MIN;
3     assigns  \nothing;
4     ensures (\old(x) >= 0 ==> \result == \old(x));
5     ensures (\old(x) < 0 ==> \result == -\old(x));
6 */
7 int abs ( int x ) {
8   if ( x  >=0 ){
9     return x;
10  }
11  return  -x ;
12 }
```

Figure 2.1 – Annotated C function

ACSL specifications are written inside comments to guarantee no interference with the original C code and begin with the symbol `@`. A classical function contract is composed of three parts:

- A list of preconditions stating that the caller must call the function in a state where the preconditions hold. Preconditions are written using the clause **`requires`** as shown in Figure 2.1 line 2. Each precondition can be specified in a separate clause or grouped with the other preconditions in a single clause using conjunction.
- A frame clause, stating that the function does not modify any non-local memory location except those specified in the clause **`assigns`** as shown in Figure 2.1 line 3. We show later a more formal definition of the frame clause.
- A list of postconditions stating that the function returns a state where the postconditions hold. Postconditions are written using the clause **`ensures`** as shown in Figure 2.1 line 4-5. As for the case of precondition, each postcondition can be specified in a separate clause or grouped with the other postconditions in a single clause using conjunction.

Figure 2.1 shows function `abs`, computing the absolute value of `x`, and the attached ACSL contract on lines 2-5. Line 2 shows the precondition stating that variable `x` must be greater than `INT_MIN` to avoid overflow. Line 3 shows the frame clause stating that the function leaves the global memory entirely unchanged. Finally, line 4-5 shows the postconditions stating that the result of function `abs` is the absolute value of parameter `x`.

We use the construct **`\old`** in the postcondition to refer to the value of `x` in the state before the execution of the program. As described in the manual [BCF+13], formal parameters in function

contracts are defined such that they always refer implicitly to their values interpreted in the pre-state. Thus, the postcondition can also be written without construct `\old`. Construct `\result` is used to refer to the result of the function.

Named Behaviors can be used to make function annotations more structured: pre- and post-conditions associated to a named behavior must only be ensured if the associated assumption is verified at the beginning of the function.

```
1  #include <limits.h>
2  /*@ behavior pos:
3          assumes x >= 0;
4          requires \true;
5          assigns \nothing;
6          ensures \result  == x;
7      behavior neg:
8          assumes x < 0;
9          requires x > INT_MIN;
10         assigns \nothing;
11         ensures \result  == -x;
12     complete behaviors;
13     disjoint behaviors;
14 */
15 int abs ( int x ) {
16   if ( x  >=0 ){
17     return x;
18   }
19   return  -x ;
20 }
```

Figure 2.2 – Annotated C function with behaviors in ACSL

Figure 2.2 shows function `abs` and the contracts written using named behaviors. We recognize two named behaviors: `pos` for `x` $\geq$ `0` and `neg` for `x < 0`. Each behavior is composed of four clauses; **`requires`**, **`ensures`** and **`assigns`** building the function contract for the behavior, and a new clause **`assumes`**. This new clause sets the condition for which the behavior must be ensured. Default behavior (without assumes clause, *i.e.* that must always hold) can also be defined in parallel to named behaviors. In the case of the example on Figure 2.2, no default behavior is specified.

The semantics of named behavior is as follows:

- The caller must ensure that if the assumption of the behavior holds, the precondition of the behavior holds. Moreover the default precondition must hold. In case of the example of Figure 2.2, the caller must ensure that the call is performed in a state where the property `(x >= 0 ==> \true) && (x < 0 ==> x > INT_MIN)` holds.

- The called function returns a state where the postcondition of each behavior holds, assuming the associated assumption holds. Moreover the function returns a state where the default postcondition holds. In case of the example of Figure 2.2, we have the postconditions `(x >= 0 ==> \result == x) && (x < 0 ==> \result == -x)`.

- If the assumption of the behavior holds, the function does not modify any non-local memory location except those specified in the clause **assigns** of the behavior or of the default behavior.

ACSL accepts contracts that are not complete or disjoint; partial specifications, or behaviors that are partially or completely overlapping are authorized. To ensure that we have complete and disjoint behaviors, we can use clause **complete behaviors** and **disjoint behaviors**. These clauses actually verify properties on the function contract itself, and not on the implementation.

For a default precondition `R` and a list of behavior conditions `A_1, A_2, ... ,A_n`, the semantics of clause **complete behaviors** is:

```
R ==> (A_1 || A_2 || ... || A_n).
```

That is, we never have the case where the default precondition holds, but no behavior condition holds

```
!(R && !A_1 && !A_2 && ... & !A_n).
```

The semantics of clause **disjoint behaviors** is:

```
R ==> !(A_1 && A_2) && ... && !(A_1 && A_n) && !(A_2 && A_3) && ... && !(A_n-1 && A_n).
```

That is, we never have the case where the default precondition holds and two (or more) behavior conditions holds.

```
!(R && ((A_1 && A_2) || ... || (A_1 && A_n) || (A_2 && A_3) || ... || (A_n-1 && A_n))).
```

Statement annotations allow writing annotations directly on a statement. The **assert** `P` clause is an example of a statement annotation that ensures that a condition `P` holds at a given program point.

Loop annotations are statement annotations used specifically for loops. They are written before the loop and are divided into three clauses:

- Loop invariants are written using the clause **loop invariant** `I` and have the same semantic as loop invariants in *Hoare logic* (Section 3.3): `I` holds in state before the loop, and for any iteration, if `I` holds at the beginning of the iteration, then it also holds after executing another loop step. The proof of invariances is done by induction; we assume that the invariant holds at the beginning of an arbitrary iteration and prove that the invariant holds after the execution of the loop body.
- Loop variants are written using the clause **loop variant** `V` and have the same semantic as loop variants in *Hoare logic* [CPR11]: for any iteration, after the execution of the loop body, the value of `V` must be smaller than at the beginning of the iteration, and the value of `V` must be positive at the beginning of any iteration.

- Frame clauses for loops are written using the clause **loop assigns** M and are similar to **assigns** M (the loop does not modify any memory location except those specified in the clause **loop assigns**).

Figure 2.3 shows a function **loop** always returning 10. The case where $x \geq 10$ is easy as we directly return 10. However, the case where $x < 10$ is more complicated since we have a loop. We have to prove that at loop exit, the value of n is 10. To do so, we use the invariant $10 \geq n$. We also specify that the loop only changes the memory location n and that the expression $10 - n$ is a variant of the loop to guarantee termination.

```
1  /*@ requires x >= 0;
2      assigns \nothing;
3      ensures \result == 10;
4  */
5  int loop(int x){
6    int  n=x;
7    if (n>=10)
8      return 10;
9    else {
10     /*@ loop invariant 10 >= n ;
11         loop assigns n ;
12         loop variant 10 - n;
13     */
14     while (n<10) n++;
15     return n;
16   }
17 }
```

Figure 2.3 – Loop annotations

Memory locations can be constrained through specific built-in predicates. The predicate

\valid{L}(s)

applies to a set of terms of some pointer type and holds if and only if dereferencing any pointer $p \in s$ is safe at memory state L, both for reading and writing. The predicate

\separated(s_1,s_2)

applies to two sets of terms of some pointer type and holds if and only if $\forall p \in s_1$ and $\forall q \in s_2$, p and q are segregated:

$$\forall i, j \in \textbf{integer}, 0 <= i < \textbf{sizeof}(*p), 0 <= j < \textbf{sizeof}(*q) ==> (\textbf{char}*)p + i \neq (\textbf{char}*)q + j$$

Figure 2.4 shows function swap that swaps the values in locations indexed by x and y. To ensure that both read and write accesses to *x and *y are safe, we add the precondition at line 1. If we want to ensure that swap is not called for the same location (swap(x,x)) or for overlapping locations, we can add the precondition at line 2. This precondition ensures that the function is called with two pointers indexing two different memory locations. Finally on line 3, the

```
1 /*@ requires \valid(x) && \valid(y);
2     requires \separated(x,y);
3     ensures *x == \old(*y) && *y == \old(*x);
4     assigns *x,*y;
5 */
6 void swap(int *x, int *y){
7   int t = *x;
8   *x=*y;
9   *y=t;
10 }
```

Figure 2.4 – Annotated C function with pointers

postcondition ensures that the value stored in memory indexed by x is the initial value stored in memory indexed by y and vice versa.

Frame rule, introduced earlier in case of ACSL specifications, indicates the set of non-local memory locations that may be modified by a function.

For a clause `/*@ assigns` l`;*/`, we have location l that may be modified. That is, using the semantics of `\separated` and assuming `locations` represents the set of allocated locations of the memory, we have:

`\forall` $loc \in$ `locations, \separated(`loc`, &`l`) ==>` $*loc$ `==` `\old(`$*loc$`)`

Functional dependencies is an extended syntax of clauses `assigns` adding a `\from` part. It indicates that the assigned value of a potentially modified location, can only depend upon the mentioned locations.

For a clause `/*@ assigns` l `\from` $l_1, \ldots l_k$`;*/`, the assigned values, to location l, does not depend on any locations which is separated from l_1,...l_k. That is, the assigned value to location l can be expressed in function of location l_1,...l_k;

$$l = \texttt{f}(l_1, \ldots, l_k)$$

If the `\from` part is absent, all the locations are supposed to be used.

Memory state referring can be done using the built-in construct `\at(e,id)` to refer to the value of an expression `e` at specific program point `id`. The `\old(e)` construct, shown previously, is just syntatic sugar for `\at(e,Pre)`. There exist different predefined program points we can refer to. The most common are:

- `Pre` for the state before the function execution,
- `Post` for the state after the function execution,
- `Here` for the current state.

It is also possible to use C labels in construct `\at`.

Global annotations are used to add expressiveness to the language used in annotations by declarations of new logic types, logic constants, logic functions and predicates.

For example, consider function `find_min` on Figure 2.5 for finding the index corresponding to the smallest value in an array between indices `a` and `b`. We use in the contract of this function predicate `is_min` to denote the fact that an integer `min` is the index of an element that is smaller or equal to all elements in an array `tab` between indices `a` and `b`.

```
1 /*@ predicate is_min(int* tab, integer a, integer b, integer min) =
2       \forall integer k; a <= k <= b ==> tab[min] <= tab[k];*/
3
4 /*@ requires 0 <= a <= b;
5     requires \valid(tab+(a..b));
6     assigns \nothing;
7     ensures is_min(tab,a,b,\result);
8     ensures a <= \result <= b;
9 */
10 int find_min(int tab[], int a, int b){
11   int min = a, i;
12   /*@ loop invariant a <= i <= b+1;
13       loop invariant a <= min <= b;
14       loop invariant is_min(tab,a,i-1,min);
15       loop assigns min, i;
16       loop variant b - i;
17    */
18   for(i = a; i <= b; i++){
19     if(tab[i] < tab[min]) min = i;
20   }
21   return min;
22 }
```

Figure 2.5 – Usage of global annotation in ACSL: predicates

On Figure 1.1, we have shown the use of axiomatic definitions to axiomatize the behaviour of factorial.

2.2 The WP plugin

The FRAMA-C/WP plugin is a verification condition generator, which for a C program annotated with ACSL specifications, returns a set of proof obligations that can be discharged either automatically by automated theorem provers (e.g. Alt-Ergo, CVC4, Z3 [2]) or with some help from the user *via* a proof assistant. If those proof obligations are valid, the program satisfies the annotations.

[2]See, resp., `https://alt-ergo.ocamlpro.com`, `http://cvc4.cs.nyu.edu`, `https://z3.codeplex.com/`

2.2.1 Simple Example

If we consider function `loop` shown on Figure 2.3, applying WP for proving the preservation of loop invariant on line 10 by the loop body, gives us the following proof obligation;

Assume {
(* Pre-condition *)
Have: $0 <= x$. 1
(* Initializer *)
Init: $x = n$. 2
(* Else *)
Have: $n <= 9$. 3
(* Invariant *)
Have: $n_2 <= 10$. 4
(* Then *)
Have: $n_2 <= 9$. 5
Have: $(1 + n_2) = n_3$. 6
}
Prove: $n_3 <= 10$.

The proof obligation is composed of a set of assumptions, and a formula that has to be proven.

The assumptions are a translation into first order logic of the precondition (1), the assignment of parameter `x` to local variable `n` (2), the fact that the boolean condition of the **if** is false (3), the fact that the loop invariant holds at the begining of the loop body (4), the fact that the loop condition holds at the begining of the loop body (5) and the loop body, corresponding to an assignment (6). Note that there is no connection between the logical variable modeling the local variable `n` before the loop and at the beginning of the loop body, since the proof of preservation of the loop invariant is done for any loop iteration.

The formula that has to be proven corresponds to the invariant after the loop body. As briefly explained before, the proof of the preservation of the invariant consists in verifying that the invariant holds after the body of the loop, assuming the invariant holds before the loop iteration. In the case of the example, it is not very hard to prove that the invariant holds from the assumptions in (5) and (6).

In Chapter 5 we present, in a simplified way, how such a formula is generated.

2.2.2 Advanced Example

The WP plugin supports a large subset of the C syntax and allows proving advanced examples, as shown in *ACSL by Example* [3] for various examples taking from the C++ library *Standard Template Library* (STL).

We propose in the following an example of an annotated selection sort algorithm on arrays of integers, to present an advanced example.

[3] `https://github.com/fraunhoferfokus/acsl-by-example.git`

The proof of a sorting algorithm is frequently divided in two parts: proving that the resulting array is sorted, and that the algorithm preserves the elements of the initial array. Therefore, we first define what is a sorted array using a logical predicate:

```
/*@ predicate sorted(int* tab, integer idx) =
      \forall integer x,y; 0 <= x < y < idx ==> tab[x] <= tab[y]; */
```

The predicate states that an array `tab` is sorted between indices `0` included and `idx` excluded, by comparing the elements at different indices pairwise.

Then, we define an inductive predicate stating that two arrays have the same elements (possibly in a different order) between two states:

```
/*@ inductive same_elements{L1, L2}(int *a, int *b, integer begin, integer end) {
      case refl{L1, L2}:
        \forall int *a, int *b, integer begin, end;
        same_array{L1,L2}(a, b, begin, end) ==>
        same_elements{L1, L2}(a, b, begin, end);
      case swap{L1, L2}: \forall int *a, int *b, integer begin, i, j, end;
        swap{L1, L2}(a, b, begin, i, j, end) ==>
        same_elements{L1, L2}(a, b, begin, end);
      case trans{L1, L2, L3}: \forall int* a, int *b, int *c, integer begin, end;
        same_elements{L1, L2}(a, b, begin, end) ==>
        same_elements{L2, L3}(b, c, begin, end) ==>
        same_elements{L1, L3}(a, c, begin, end);
}*/
```

The case `refl` states that if both arrays are equal, using predicate `same_array`, then the arrays have the same elements.

```
/*@ predicate same_array{L1,L2}(int *a, int *b, integer begin, integer end) =
      \forall integer k; begin <= k < end ==> \at(a[k],L1) == \at(b[k],L2);
*/
```

The case `swap` states that if an array has only two indices that are swapped against another array, then the arrays have the same elements.

```
/*@ predicate swap{L1, L2}(int *a, int *b, integer begin,
                           integer i, integer j, integer end) =
      begin <= i < end && begin <= j < end &&
      \at(a[i], L1) == \at(b[j], L2) &&
      \at(a[j], L1) == \at(b[i], L2) &&
      \forall integer k; begin <= k < end && k != i && k != j
                         ==> \at(a[k], L1) == \at(b[k], L2);
*/
```

Finally, the case `trans` states that the predicate `same_elements` is transitive. Notice that the inductive definition of predicate `same_elements` can also have an equivalent definition using simple axioms (like for predicate `isFact` in Figure 1.1).

Using those predicates, we can annotate the implementation of a selection sort algorithm written in C shown on Figure 2.6. The implemented sorting algorithm divides the input array into two parts: the sub-array already sorted, which is built up from `0` to `i` (excluded), and the sub-array remaining unsorted from `i` to `n`. Initially, the sorted sub-array is empty and the unsorted sub-array is the entire input array. The algorithm proceeds by finding the smallest element in

```
1  /*@ requires 0 <= n;
2      requires \valid(tab+(0..n));
3      ensures sorted(tab,n+1);
4      ensures same_elements{Pre,Post}(tab,tab,0,n+1);
5      assigns tab[0..n];
6  */
7  void select_sort(int tab[], int n){
8    int i, min;
9    /*@ loop invariant 0 <= i <= n+1;
10       loop invariant \forall int j,k; 0 <= j < i <= k < n+1 ==> tab[j] <= tab[k];
11       loop invariant sorted(tab,i);
12       loop invariant same_elements{Pre,Here}(tab,tab,0,n+1);
13       loop assigns i, tab[0..n], min;
14       loop variant (n+1) -i;
15   */
16   for (i = 0; i <= n; i++){
17     min = find_min(tab,i,n);
18     if(min != i) {
19     l1:swap(tab+i, tab+min);
20       /*@ assert swap{l1,Here}(tab,tab,0,i,min,n+1);*/
21     }
22   }
23   return;
24 }
```

Figure 2.6 – Selection sort algorithm written in C with ACSL annotations

the unsorted sub-array, using function `find_min`, defined previously on Figure 2.5 and swapping it with the leftmost unsorted element (index `i`) using the `swap` function defined previously on Figure 2.4.

The contract of function `select_sort` has a postcondition which states that after the call the array `tab` is sorted between indices `0` an `n+1` (excluded) (line 3 on Figure 2.6). A second postcondition states that the resulting array `tab` in state **`Post`** has the same elements as array `tab` in state **`Pre`** (line 4 on Figure 2.6).

The contract is also composed of two preconditions which state that the integer `n` is not negative and `tab` has valid locations up to `n` (lines 1–2 on Figure 2.6).

Finally, an assigns clause states that only locations of `tab` between indices `0` to `n` may be modified (line 5 on Figure 2.6).

To prove the postconditions, four loop invariants are used to summarize the behavior of the loop. The invariant at line 8 states that the value of `i` is between `0` and `n+1`. Thus, we know at loop exit that the value of `i` is `n+1`, by combining the invariant with the negation of the loop condition (`i <= n`).

The invariant at line 9 states that the values in the left sub-array are smaller than the values of the right sub-array.

The invariant at line 10 states that the left sub-array (from `0` to `i` excluded) is sorted. At loop exit we know that the whole array is sorted (post condition at line 3), since the value of `i` is `n+1`.

The invariant at line 11 states that the current array has always the same elements as the initial array. At loop exit, we have the final array that has the same elements as the initial array

(postcondition at line 4). Notice that we use an assertion at line 20 to guide the prover on how the case `trans` must be instantiated.

Chapter 3

Context

This chapter is devoted to the presentation of the language used in our formalizations. Since we address the verification of properties on imperative programs such as C, we choose as a basis for our modelling language a simple imperative *while* language with procedure calls. *while* language has the benefit of providing basis for tractable formalizations while remaining close to real imperative languages. The R-WHILE language we propose in this chapter is similar to the one proposed in [Win93], augmented with procedure calls [AdO09]. Indeed, as stated in Chapter 1, we focus on properties connecting functions. Knowing that modelling function calls is complicated and requires a significant amount of work, we propose in the following chapter a formalization of function calls that is as simple as possible: our functions have neither parameters nor a return value. Everything is done by reading and updating global variables shared by all functions. Examples of formalization with function calls with parameters can be found in [Car94, AdO09].

In addition to the language presentation, we recall the notion of Hoare Triples for reasoning about program properties, and Hoare Logic [Hoa69] for the verification of Hoare Triples.

The chapter is organized as follows. Section 3.1 will first introduce the notations we use in the rest of this thesis. Section 3.2 presents the syntax and semantics of the *while* language, and Section 3.3 presents the related Hoare Logic.

3.1 Notations

Most notations and terms presented in the following are taken from [Win93]. We recall them in Sections 3.1.1 and 3.1.2 with slight variations that will make it easier to introduce our work.

Section 3.1.3 introduces some notations about Fist-Order Logic used in the following formalization.

3.1.1 Set notations

Sets are denoted with upper-case letters A and elements of sets with lower-case letters a:

$$a \in A.$$

We use $\mathcal{P}$ to denote the set consisting of all subsets of a set (powerset):

$$\mathcal{P}(A) = \{B | B \subseteq A\}.$$

We use u with index A as a notation for an element of the powerset of a set $A : u_A \in \mathcal{P}(A)$.

We denote by $X \times Y$, the set of pairs (x, y), with the first element from X and the second from Y. We denote by $A^n = A \times .... \times A$, the set of tuples of size n over A . We denote by $S \backslash X$ the subtraction of subset X from S.

The set of partial functions from A to B is written $A \rightharpoonup B$ and the set of total functions $A \to B$. The fact that a function f is in the set of total functions taking a parameter from A and returning a value in B is written $f : A \to B$. The application of a (partial) function f to x (called a parameter) is written $f(x)$ or $f\ x$ and can be defined or undefined (in the case of partial functions), the latter being noted $\perp$. We assume that the application of $\perp$ is equal to $\perp$: for all set X, we have $\forall x \in X. \perp x = \perp$. Thus, for a function $f : X \rightharpoonup (Y \rightharpoonup Z)$, we have following behavior:

$$\forall x \in X, y \in Y.(f(x))y = \begin{cases} \perp & \text{if } f(x) = \perp \\ (f(x))y \end{cases} .$$

We call dom the set for which a function $f : A \rightharpoonup B$ is defined:

$$dom(f) = \{a | a \in A. f(a) \neq \perp\}.$$

A function can be defined using a relation between the parameters.

Example 3.1. We define a function $fst : X \times Y \to X$ for extracting the first element of a pair:

$$fst((x, y)) = x,$$

and its counterpart $snd : X \times Y \to Y$ for extracting the second element of a pair:

$$snd((x, y)) = y.$$

Alternatively, a function can be defined over a finite domain by writing the set of its bindings.

Example 3.2. Let us define a function f as follows:

$$f = \{a_1 \mapsto b_1, ..., a_n \mapsto b_n\}.$$

Using the defined function f, we can define a new function in two ways:

- By adding new bindings

$$f[a_{n+1} \leftarrow b_{n+1}] \equiv \{a_1 \mapsto b_1, ..., a_n \mapsto b_n, a_{n+1} \mapsto b_{n+1}\}.$$

- By modifying existing bindings

$$f[a_1 \leftarrow b_m] \equiv \{a_1 \mapsto b_m, ..., a_n \mapsto b_n\}.$$

3.1.2 Syntax and Semantics notations

In this manuscript, all grammar rules are given using a notation close to BNF [Pie02]. All semantics are given using denotational semantics [Win93, Min15, Sch86]. Thus, we use the traditional notation $[\![\cdot]\!]$ around an argument of a semantic function to show that the argument is a piece of syntax.

Example 3.3. Let $f : A \to (B \to C)$ be a semantic function from A to a function from B to C. When we use the notation $f[\![a]\!]$, we denote that the parameter of function f matches the syntactic object a. Moreover, since $f[\![a]\!]$ is a function of $B \to C$, $f[\![a]\!]b$ is the application of the function $f[\![a]\!]$ to b and returns an object of C.

3.1.3 Monomorphic First-Order Logic

In Section 2.2 we presented the plugin WP for the generation of verification conditions, defined in a First-Order Logic. As in the following we intend to formalize a verification condition generator such as WP, we introduce in this section the syntax of Monomorphic First-Order Logic [SSCB12] used in the following chapters. Monomorphic First-Order Logic, called below MFOL, is an extension of the well-known First-Order Logic that consists in adding types to the logic.

Example 3.4. Assume we have two types, nat and array, for the sets of natural numbers and arrays of natural numbers, and following associated functions:

- $m[e]$ (of type : array $\times$ nat $\to$ nat) for accessing an array m at index e.
- $m[e_1 \leftarrow e_2]$ (of type : array $\times$ nat $\times$ nat $\to$ array) for updating an array m at index e_1 with e_2.

Axiomatisation of arrays can be done by the following two formulas:

$$\forall v_t : \text{array}, v_1, v_2, v_3 : \text{nat}, \qquad v_1 \neq v_3 \Rightarrow (v_t[v_1 \leftarrow v_2])[v_3] = v_t[v_3], \qquad \text{(Q-NO-UPDATE)}$$

$$\forall v_t : \text{array}, v_1, v_2 : \text{nat}, \qquad (v_t[v_1 \leftarrow v_2])[v_1] = v_2. \qquad \text{(Q-UPDATE)}$$

Formula Q-NO-UPDATE tells that updating a location in an array does not change other locations. Formula Q-UPDATE tells that getting the value of a location updated with value v results in value v.

We quickly recall the grammar rules defining MFOL, but typing rules and semantics of

MFOL are not presented, as they are standard [BBPS16]:

$$
\begin{aligned}
t &::= \text{nat} \\
&\mid \text{array}
\end{aligned}
$$

$$
\begin{aligned}
e &::= f(e_1, ..., e_n) \\
&\mid v^t
\end{aligned}
$$

$$
\begin{aligned}
q &::= T \mid F \\
&\mid \text{p}(e_1, ..., e_n) \mid q_1 \wedge q_2 \mid q_1 \vee q_2 \mid q_1 \Rightarrow q_2 \mid \neg q \\
&\mid e_1 =_t e_2 \\
&\mid \forall v : t.q \mid \exists v : t.q
\end{aligned}
$$

The syntax of MFOL is made of types, terms and formulae. Types t are composed of naturals and arrays of naturals. Note that only simple types can be defined in MFOL. Type constructors can be found in Polymorphic First-Order Logic [BP13] and we will not use them in our formalization.

A term e is either a typed variable v^t, composed of a variable and a type, or the application of a function symbol f to a list of terms (we call $\mathbb{V}$ the set of typed variables and use variables $v, v_0, v_1, ...$ to range over $\mathbb{V}$).

Function symbols f have a signature composed of a list of types $t_1, ..., t_n$ and a type return t such that $f : t_1 \times ... \times t_n \to t$. Terms of a given type are noted e^t in cases where the type is not clear. In the following we use $\mathbb{E}_q$ for the set of terms and use variables $e, e_0, e_1, ...$ to range over $\mathbb{E}_q$.

A formula q is either T (true), F (false), conjunction, disjonction, negation, equality between terms or application of a predicate symbol. We also add implication for convenience only, since, in classical logic, $q_1 \Rightarrow q_2$ is equivalent to $\neg q_1 \vee q_2$. Predicate symbols p have a signature composed of a list of types $t_1, ..., t_n$ such that $\text{p} : t_1 \times ... \times t_n \to o$ (o is the pseudotype for set $\{T, F\}$). In the following we use $\mathbb{Q}$ for the set of formulae and use variables $q, q_0, q_1, ...$ to range over $\mathbb{Q}$.

To avoid having different signatures for the same symbol we assume the sets of function and predicate symbols are disjoint. As the signature is given by a function that maps function and predicate symbols to a signature, nothing more is required. In order to make the variables easier to distinguish, we sometimes use m for variables of type array (the associated set is called $\mathbb{M}$) and i for natural variables (the associated set is called $\mathbb{I}$).

For telling if a formula q is valid for a given set of closed formulas (no free variable) or axioms $u_{\mathbb{Q}}$, we define function smt with the following signature:

$$\text{smt} : \mathcal{P}(\mathbb{Q}) \times \mathbb{Q} \to Verdict,$$

with $Verdict = \{V, U\}$ where V states for valid and U for unknown.

We consider function smt as an oracle or a black box for the theories used by Satisfiability Modulo Theories (SMT) solver. As the theories used by SMT solvers are often not decidable, it may be not possible to prove that a valid formula is valid. We assume in the following that if function smt declares a formula valid (V), the formula is valid, and otherwise we do not know if the formula is false or if it cannot be proven valid. In the latter case we consider that the validity of the formula is unknown (U).

In the following we assume the function smt has a native theory for naturals, which depends in practice on the SMT solvers. Thus, we consider in the following that the theory (axioms and functions) for naturals is suitable for our needs and is left implicit.

Example 3.5. If we name formula Q-NO-UPDATE $q_{no-update}$ and Q-UPDATE q_{update}, and consider formula q_1 defined by:

$$(m_1 = m_0[4 \leftarrow i_1] \wedge m_2 = m_1[i_2 \leftarrow i_3] \wedge \neg(4 = i_2)) \Rightarrow m_2[4] = i_1.$$

We expect that calling function smt, with formula $q_{no-update}$ and q_{update} as axioms and asking if q_1 is valid, $\mathrm{smt}(\{q_{no-update}, q_{update}\}, q_1)$ returns V. In practice any SMT solver classically used as program verification will be able to answer V.

3.2 *While* Language with Procedure calls

In this section, we present the syntax of our R-WHILE language and the corresponding denotational semantics.

3.2.1 Program Syntax

The following categories of sets are associated to the language:

- Constants:
  - $\mathbb{N}$, the set of natural numbers,
  - $\mathbb{B} = \{true, false\}$, the set of boolean values,
- Memory locations:
  - $\mathbb{X}$, the set of locations for natural numbers,
  - $\mathbb{Y}$, the set of program names,
- Syntactic expressions:
  - $\mathbb{E}_a$, the set of arithmetic expressions,
  - $\mathbb{E}_b$, the set of boolean expressions,
- Commands:
  - $\mathbb{C}$, the set of commands,

- Operators:
  - $\mathbb{O}_a$ the set of arithmetic operators,
  - $\mathbb{O}_b$ the set of boolean operators,
  - $\mathbb{O}_l$ the set of logical operators.

We define the following metavariables to range over the categories:

- $n, n_0, n_1, ... \in \mathbb{N}$,
- $x, x_0, x_1, ... \in \mathbb{X}$,
- $a, a_0, a_1, ... \in \mathbb{E}_a$,
- $b, b_0, b_1, ... \in \mathbb{E}_b$,
- $c, c_0, c_1, ... \in \mathbb{C}$,
- $y, y_0, y_1, ... \in \mathbb{Y}$,

We can now define the grammar rules for arithmetic expressions, boolean expressions and commands:

$$
\begin{aligned}
opa &::= + \mid \times \mid - \\
opb &::= < \mid = \mid > \\
opl &::= \vee \mid \wedge
\end{aligned}
$$

$$
\begin{aligned}
a ::&= n \\
&\mid x \\
&\mid a_1\ opa\ a_2
\end{aligned}
$$

$$
\begin{aligned}
b ::&= true \mid false \\
&\mid a_1\ opb\ a_2 \\
&\mid b_1\ opl\ b_2 \mid \neg b_1
\end{aligned}
$$

$$
\begin{aligned}
c ::&= \textbf{skip} && \text{do nothing} \\
&\mid x := a && \text{assignment} \\
&\mid c_1; c_2 && \text{sequence} \\
&\mid \textbf{assert}(b) && \text{assertion} \\
&\mid \textbf{if}\ b\ \textbf{then}\ \{c_1\}\ \textbf{else}\ \{c_2\} && \text{condition} \\
&\mid \textbf{while}\ b\ \textbf{do}\ \{c_1\} && \text{loop} \\
&\mid \textbf{call}(y) && \text{procedure call}
\end{aligned}
$$

Command **assert**(b) is called an assertion and indicates that boolean expression b must be valid at the point where the command occurs. Command **call**(y) is a function call without explicit parameter and return value (thus we use in the future the expression procedure call rather than function call). As in assembly code, with a given calling convention [Kip14], parameters and return value are shared implicitly between the caller and the callee through memory locations *i.e.* it is left to the caller to put the right values to the right locations before the call. The called program is the program located at y. For simplicity reasons, we do not provide commands allowing to update command bindings: our set of routines will be fixed for each program. Finally, to resolve ambiguity between sequences of commands, we use $\{\}$.

As an example, we consider the following two programs, with one using a procedure call.

$$\begin{array}{ll} x_1 := x_0 + 5; & x_1 := x_0 + 5; \\ \textbf{call}(y); & x_2 := x_1 + 4; \\ x_3 := x_2 + x_1 & x_3 := x_2 + x_1 \end{array}$$

The semantics we give in the following section implies that if program name y maps to commands $x_2 := x_1 + 4$, the two programs are equivalent, since by replacing the call command by the called program, we get identical programs.

3.2.2 Program Evaluation

In addition to the categories of sets introduced in the previous section, we add $\Sigma = \mathbb{X} \rightharpoonup \mathbb{N}$, the set of memory states for natural numbers, mapping locations to naturals. We also define $\Psi = \mathbb{Y} \rightharpoonup \mathbb{C}$, the set of memory states for commands, mapping program names to command. We use metavariables $\sigma, \sigma_0, \sigma_1, ...$ to range over Σ, and $\psi, \psi_0, \psi_1, ...$ to range over Ψ.

In order to avoid complexity, we assume that the set of locations for natural numbers $\mathbb{X}$ and the set of program names $\mathbb{Y}$ are disjoint ($\mathbb{X} \cap \mathbb{Y} = \emptyset$), *i.e.* we can only access natural numbers through Σ using $\mathbb{X}$ and commands through Ψ using $\mathbb{Y}$.

ξ_a will denote the function that evaluates arithmetic expressions in $\mathbb{N}$, according to a memory state for natural numbers. Thus, for an arithmetic expression a in $\mathbb{E}_a$, we have $\xi_a[\![a]\!] : \Sigma \rightharpoonup \mathbb{N}$. ξ_b will denote the function that evaluates boolean expressions in $\mathbb{B}$, according to a memory state for naturals numbers. Thus, for a boolean expression b in $\mathbb{E}_b$, we have $\xi_b[\![b]\!] : \Sigma \rightharpoonup \mathbb{B}$. ξ_c will denote the function that evaluates commands in Σ, according to a pair, composed of a memory state for natural numbers and a memory state for commands. Thus, for a command c in $\mathbb{C}$, we have $\xi_c[\![c]\!] : \Sigma \times \Psi \rightharpoonup \Sigma$

Notice that all those functions are partial functions, since memory states are partial functions. In the case of $\xi_c[\![c]\!]$ there are also other reasons that will be discussed later. As a result, some evaluation cases may be undefined. For convenience, we lift the sets of natural numbers and booleans with $\perp$:

$$\mathbb{N}_\perp = \mathbb{N} \cup \{\perp\},$$
$$\mathbb{B}_\perp = \mathbb{B} \cup \{\perp\},$$

to get total functions for arithmetic expressions and boolean expressions. We also extend any binary arithmetic operator opa, boolean operator opb and logical operator opl with the following

rules:

$$\forall a_1, a_2 \in \mathbb{N}_\perp . a_1 \ opa_\perp \ a_2 = \begin{cases} \perp & \text{if } a_1 = \perp \text{ or } a_2 = \perp \\ a_1 \ opa_{\mathbb{N}} \ a_2 & \text{otherwise} \end{cases}$$

$$\forall a_1, a_2 \in \mathbb{N}_\perp . a_1 \ opb_\perp \ a_2 = \begin{cases} \perp & \text{if } a_1 = \perp \text{ or } a_2 = \perp \\ a_1 \ opb_{\mathbb{N}} \ a_2 & \text{otherwise} \end{cases}$$

$$\forall b_1, b_2 \in \mathbb{B}_\perp . b_1 \ opl_\perp \ b_2 = \begin{cases} \perp & \text{if } b_1 = \perp \text{ or } b_2 = \perp \\ b_1 \ opl_{\mathbb{B}} \ b_2 & \text{otherwise} \end{cases}$$

Unary operator $\neg$ is extended with the following rule:

$$\forall b, \in \mathbb{B}_\perp . \neg_\perp b = \begin{cases} \perp & \text{if } b = \perp \\ \neg b & \text{otherwise} \end{cases}$$

We can now provide the semantics for arithmetic expression evaluation, and the semantics for boolean expression evaluation.

Definition 3.1. Evaluation function $\xi_a : \mathbb{E}_a \to (\Sigma \to \mathbb{N}_\perp)$, for arithmetic expressions $\mathbb{E}_a$, is defined by structural induction on arithmetic expressions:

$$\begin{aligned} \xi_a[\![n]\!]\sigma &= n \\ \xi_a[\![x]\!]\sigma &= \sigma(x) \\ \xi_a[\![a_0 \ opa \ a_1]\!]\sigma &= \xi_a[\![a_0]\!]\sigma \ opa_\perp \ \xi_a[\![a_1]\!]\sigma. \end{aligned}$$

Definition 3.2. Evaluation function $\xi_b : \mathbb{E}_b \to (\Sigma \to \mathbb{B}_\perp)$, for boolean expressions $\mathbb{E}_b$, is defined by structural induction on boolean expressions

$$\begin{aligned} \xi_b[\![true]\!]\sigma &= true \\ \xi_b[\![false]\!]\sigma &= false \\ \xi_b[\![a_0 \ opb \ a_1]\!]\sigma &= \xi_a[\![a_0]\!]\sigma \ opb_\perp \ \xi_a[\![a_1]\!]\sigma \\ \xi_b[\![b_0 \ opl \ b_1]\!]\sigma &= \xi_b[\![b_0]\!]\sigma \ opl_\perp \ \xi_b[\![b_1]\!]\sigma \\ \xi_b[\![\neg b]\!]\sigma &= \neg_\perp \xi_b[\![b]\!]\sigma. \end{aligned}$$

For command evaluation, in addition to the fact that memory states are partial functions, we have to take into account that commands **while** and **call** may lead to a non-terminating evaluation.

Example 3.6. For memory state $\psi = \{y \mapsto \textbf{call}(y)\}$ and the following program c defined by:

$$\textbf{call}(y)$$

the evaluation does not terminate, since **call**(y) calls **call**(y).

To obtain a finite mathematical object, the denotational semantics for commands commonly uses least fixed point [Win93, AGM94]. However, for commands **while** and **call**, we prefer a fixpoint free semantics to get a total function. To ensure that evaluation terminates, we pass an additional parameter to the evaluation function that tells it how long it can run. The evaluation terminates either normally or stops if the parameter reaches 0. In our case, we take a natural number n which decreases for each call to the evaluation function. In case n reaches 0 the evaluation returns the state Ω_t. This approach is commonly used in Coq formalization of programming languages when using functions as model (see e.g. [PAC+18], where this approach is used to prove termination of the evaluation function for a small *while* program). In this context, the parameter is called *fuel*.

In addition, we introduce the state Ω_a for the case where an assertion is false, and $\Omega_\perp$ for the case where an expression evaluates to $\perp$. Thus, we end up with a set Ω of three particular states:

$$\Omega = \{\Omega_\perp, \Omega_t, \Omega_a\},$$

and, for σ in Ω, by convention, we define $\xi_a[\![a]\!]\sigma = \perp$ and $\xi_b[\![b]\!]\sigma = \perp$.

We now lift the set Σ with Ω

$$\Sigma_\Omega = \Sigma \cup \Omega,$$

to get a total function for command evaluation.

Definition 3.3. Evaluation function $\xi_c : \mathbb{C} \to (\mathbb{N} \times \Sigma_\Omega \times \Psi \to \Sigma_\Omega)$, for commands $\mathbb{C}$, is defined, if we run out of fuel or the memory state for natural numbers is in Ω, by

$$\begin{aligned} \xi_c[\![c]\!](n, \sigma, \psi) &= \Omega_t && \text{if } n = 0 \\ \xi_c[\![c]\!](n, \sigma, \psi) &= \sigma && \text{if } \sigma \in \Omega, \end{aligned}$$

and otherwise, by structural induction on commands

$$\xi_c[\![\textbf{skip}]\!](n, \sigma, \psi) = \sigma$$

$$\xi_c[\![x := a]\!](n, \sigma, \psi) = \begin{cases} \Omega_\perp & \text{if } \xi_a[\![a]\!]\sigma = \perp \\ \Omega_\perp & \text{if } \sigma(x) = \perp \\ \sigma[x \leftarrow \xi_a[\![a]\!]\sigma] & \text{otherwise} \end{cases}$$

$$\xi_c[\![c_0; c_1]\!](n, \sigma, \psi) = \xi_c[\![c_1]\!](n-1, \xi_c[\![c_0]\!](n-1, \sigma, \psi), \psi)$$

$$\xi_c[\![\textbf{assert}(b)]\!](n, \sigma, \psi) = \begin{cases} \Omega_\perp & \text{if } \xi_b[\![b]\!]\sigma = \perp \\ \sigma & \text{if } \xi_b[\![b]\!]\sigma = true \\ \Omega_a & \text{if } \xi_b[\![b]\!]\sigma = false \end{cases}$$

$$\xi_c[\![\textbf{if } b \textbf{ then } \{c_0\} \textbf{ else } \{c_1\}]\!](n, \sigma, \psi) = \begin{cases} \Omega_\perp & \text{if } \xi_b[\![b]\!]\sigma = \perp \\ \xi_c[\![c_0]\!](n-1, \sigma, \psi) & \text{if } \xi_b[\![b]\!]\sigma = true \\ \xi_c[\![c_1]\!](n-1, \sigma, \psi) & \text{if } \xi_b[\![b]\!]\sigma = false \end{cases}$$

$$\xi_c[\![\textbf{call}(y)]\!](n, \sigma, \psi) = \begin{cases} \Omega_\perp & \text{if } \psi(y) = \perp \\ \xi_c[\![\psi(y)]\!](n-1, \sigma, \psi) & \text{otherwise} \end{cases}$$

$$\xi_c[\![\textbf{while } b \textbf{ do } \{c\}]\!](n, \sigma, \psi) = \begin{cases} \Omega_\perp & \text{if } \xi_b[\![b]\!]\sigma = \perp \\ \xi_c[\![c; \textbf{while } b \textbf{ do } \{c\}]\!](n-1, \sigma, \psi) & \text{if } \xi_b[\![b]\!]\sigma = true \\ \sigma & \text{if } \xi_b[\![b]\!]\sigma = false \end{cases}$$

Notice that $\Omega_\perp$ is returned if evaluation function $\xi_a[\![a]\!]\sigma$ or $\xi_b[\![b]\!]\sigma$ returns $\perp$, or we try to update an undefined memory location ($\sigma(x) = \perp$ in the case of assignment). This last point implies that no additional binding from memory location to value is added to the memory state *i.e.* we have no dynamic memory allocation. State Ω_a appears only if an assertion does not hold. As previously discussed, for each recursive call to $\xi_c[\![c]\!]$, the parameter n decreases.

3.3 Hoare Triple

Since a program is a function from state to state, one may want to check if executing a program c on a state verifying a boolean expression b_1 leads to a state verifying another property b_2. In the spirit of the approach initiated by Hoare in [Hoa69], such a connection between c, b_1 and b_2 can be defined using the following notation:

$$[b_1]c[b_2].$$

The triple states that, for any state σ, if σ satisfies b_1 then the execution of c on σ terminates in a state that satisfies b_2. Commonly, b_1 is called the precondition and b_2 the postcondition.

We define expression

$$\sigma \models b,$$

indicating that $\xi_b[\![b]\!]\sigma$ evaluates to $true$, and we say that σ satisfies b. We can now give a formal definition of the triple (for a fixed ψ):

$$\forall \sigma \in \Sigma . \sigma \models b_1 \Rightarrow (\exists n \in \mathbb{N} . \xi_c[\![c]\!](n, \sigma, \psi) \models b_2).$$

Notice that, since for all σ in Ω we have $\xi_b[\![b]\!]\sigma = \bot$, $\sigma \models b$ implies that $\sigma \notin \Omega$ *i.e.* the program terminates, all assertions are valid and all locations are bound.

In the future we assume that the following statements are always satisfied for a Hoare Triple:

- The set of variables used in boolean expressions b_1 and b_2 are subsets of the domain of the memory state σ:

  $$\mathcal{C}_{v_b}[\![b_1]\!] \subseteq dom(\sigma),$$

  $$\mathcal{C}_{v_b}[\![b_2]\!] \subseteq dom(\sigma),$$

  where function $\mathcal{C}_{v_b}$ (defined in Appendix A.1.1) returns the set of variables used in a boolean expression of $\mathbb{E}_b$.

- The domain of the memory state σ is equal to the set of locations used in command c and the commands associated to all defined program names in ψ:

  $$\mathcal{C}_{v_c}[\![c]\!] \cup \bigcup_{y \in dom(\psi)} \mathcal{C}_{v_c}[\![\psi(y)]\!] = dom(\sigma), \qquad (\mathcal{W}_v(c, \psi, \sigma))$$

  where function $\mathcal{C}_{v_c}$ (defined in Appendix A.1.1) returns the set of variables used in a command of $\mathbb{C}$.

- The set of program names used in command c is a subset of the domain of ψ. Moreover, for all defined program names in ψ, the associated commands use sets of program names that are subsets of the domain of the memory state ψ.

  $$\mathcal{C}_f[\![c]\!] \cup \bigcup_{y \in dom(\psi)} \mathcal{C}_f[\![\psi(y)]\!] \subseteq dom(\psi), \qquad (\mathcal{W}_f(c, \psi))$$

  where function $\mathcal{C}_f$ (defined in Appendix A.1.2) returns the set of program names used in a command c.

Statements $\mathcal{W}_v$ and $\mathcal{W}_f$ ensure that no access to undefined locations or program names are performed. Thus, the evaluation of command c is different from $\Omega_\bot$ ($\xi_c[\![c]\!](n, \sigma, \psi) \neq \Omega_\bot$). The evaluation of boolean expressions b_1 and b_2 is different from $\bot$ ($\xi_b[\![b_1]\!]\sigma \neq \bot$, $\xi_b[\![b_2]\!](\xi_c[\![c]\!](n, \sigma, \psi)) \neq \bot$), since variables used inside boolean expressions are defined in the memory state.

Lemma 3.1. *For a given c, ψ and n*

$$\forall \sigma \in \Sigma. \mathcal{W}_v(c, \psi, \sigma) \wedge \mathcal{W}_f(c, \psi) \Rightarrow \xi_c[\![c]\!](n, \sigma, \psi) \neq \Omega_\perp$$

Proof. By structural induction on $\mathbb{C}$, $\mathbb{E}_b$ and $\mathbb{E}_a$ □

An environment satisfies $\mathcal{W}_v$ and $\mathcal{W}_f$ is called well defined and denoted $\mathcal{WD}(c, \psi, \sigma)$.

We can now refine the Hoare triple in two cases, for a given ψ:

- Proof of termination:

  $$\forall \sigma \in \Sigma. \sigma \models b_1 \Rightarrow (\exists n \in \mathbb{N}. \xi_c[\![c]\!](n, \sigma, \psi) \neq \Omega_t).$$

  As we are working on finite programs, the proof of existence of an n such that the evaluation for c terminates can be refined to the proof of existence of an n such that the evaluation terminates for each loop and recursive call that occurs in c. Proving that such a natural number exists can be done by exhibiting an integral expression that stays positive but decreases strictly at each step of a loop or a recursive call. Generally, program termination has been extensively studied (see e.g. [CPR11]) and will not be discussed any further in this manuscript, as well-known techniques are readily available in our context.

- Functional correctness:

  $$\forall n \in \mathbb{N}. \forall \sigma \in \Sigma. \sigma \models b_1 \wedge \xi_c[\![c]\!](n, \sigma, \psi) \neq \Omega_t \Rightarrow \xi_c[\![c]\!](n, \sigma, \psi) \models b_2.$$

  Functional correctness is also known as partial correctness and states that, if a state σ satisfing b_1, and if the execution of c on σ terminates, the state resulting from the execution of c on σ satisfies b_2. It was originally proposed by Hoare in [Hoa69] and noted $b_1\{c\}b_2$. We use in the following the more modern notation:

  $$\{b_1\}c\{b_2\}.$$

  By taking the convention that for any boolean expression b, we have $\Omega_t \models b$, *i.e.* an evaluation that has not finished satifies any assertion, we can refine the definition of functional correctness by

  $$\forall \sigma \in \Sigma. \sigma \models b_1 \Rightarrow \dot{\xi_c}[\![c]\!](\sigma, \psi) \models b_2,$$

  where function $\dot{\xi_c}$ calls function ξ_c with an arbitrary amount of fuel n.

In [AdO09] a proof system is provided to decide if a statememt $\{b_1\}c\{b_2\}$ is valid, written $\models \{b_1\}c\{b_2\}$. The system is an extension of the original one without procedure calls that can be found in [Hoa69, Win93], known as Hoare Logic.

For a given ψ, if there is a derivation such that $\vdash \{b_1\}c\{b_2\}$ is a conclusion of rule RECURSION , then $\models \{b_1\}c\{b_2\}$ is valid.

$$\frac{\forall y \in dom(\xi).\xi \vdash \{fst(\xi(y))\}\psi(y)\{snd(\xi(y))\} \qquad \xi \vdash \{b_1\}c\{b_2\}}{\vdash \{b_1\}c\{b_2\}} \quad \text{(RECURSION)}$$

where environment ξ is defined by

$$\xi \in \Xi = \mathbb{Y} \rightharpoonup \mathbb{E}_b \times \mathbb{E}_b,$$

that is, a mapping between program names to the associated (procedure) *contract*, composed of a pre- and a post-conditions. Procedure contracts allow specifying the behavior of a single procedure call, that is, for a given program name y, if $\sigma \models fst(\xi(y))$ is verified when calling y in state σ, $\sigma' \models snd(\xi(y))$ will be verified when the call returns with state σ'. We use metavariables $\xi, \xi_0, \xi_1, ...$ to range over Ξ.

As for the initial triple, we assume that the set of variables used in the contracts are subsets of the domain of the memory state σ :

$$\begin{gathered}\forall y \in dom(\xi).\mathcal{C}_{v_b}[\![fst(\xi(y))]\!] \subseteq dom(\sigma), \\ \forall y \in dom(\xi).\mathcal{C}_{v_b}[\![snd(\xi(y))]\!] \subseteq dom(\sigma).\end{gathered} \quad (\mathcal{W}_a(\xi, \sigma))$$

The meaning of rule RECURSION, is that $\vdash \{b_1\}c\{b_2\}$ is valid, if for a set of assumptions (procedure contracts) ξ, there is a derivation such that $\xi \vdash \{b_1\}c\{b_2\}$ and each procedure contract in ξ is a conclusion of the following axioms and rules.

$$\frac{}{\xi \vdash \{b\}\textbf{skip}\{b\}} \quad \text{(SKIP)}$$

$$\frac{\models b \Rightarrow b_a}{\xi \vdash \{b\}\textbf{assert}(b_a)\{b \wedge b_a\}} \quad \text{(ASSERT)}$$

$$\frac{}{\xi \vdash \{b[a/x]\}x := a\{b\}} \quad \text{(ASSIGN)}$$

$$\frac{\xi \vdash \{b_1\}c_1\{b_2\} \qquad \xi \vdash \{b_2\}c_2\{b_3\}}{\xi \vdash \{b_1\}c_1; c_2\{b_3\}} \quad \text{(SEQUENCE)}$$

$$\frac{\xi \vdash \{b_1 \wedge b_{if}\}c_1\{b_2\} \qquad \xi \vdash \{b_1 \wedge \neg b_{if}\}c_2\{b_2\}}{\xi \vdash \{b_1\}\textbf{if } b_{if} \textbf{ then } \{c_1\} \textbf{ else } \{c_2\}\{b_2\}} \quad \text{(CONDITION)}$$

$$\frac{\xi \vdash \{b_I \wedge b_{while}\} c \{b_I\}}{\xi \vdash \{b_I\} \textbf{while } b_{while} \textbf{ do } \{c\} \{b_I \wedge \neg b_{while}\}} \quad \text{(WHILE)}$$

$$\frac{\xi(y) = (b_1, b_2)}{\xi \vdash \{b_1\} \textbf{call}(y) \{b_2\}} \quad \text{(CALL)}$$

$$\frac{\models (b_1 \Rightarrow b_1') \qquad \xi \vdash \{b_1'\} c \{b_2'\} \qquad \models (b_2' \Rightarrow b_2)}{\xi \vdash \{b_1\} c \{b_2\}} \quad \text{(CONSEQUENCE)}$$

We use notation $b[a/x]$ in rule ASSIGN for the substitution of all occurences of location x by the arithmetic expression a. Notice that rule ASSERT for assertion guarantees that Ω_a does not occur by providing the fact that the assertion b_a is a logical consequence of precondition b. For the rule WHILE for loop, we name b_I a *loop invariant*, since it is preserved by the loop body c and it holds at loop entry and exit.

The set of axioms and rules shares the environment ξ of assumptions used in case of rule CALL for procedure calls. Note that contracts must be well chosen in order to be used in accordance with the rule CALL. A proof of soundness of the proof system can be found in [AdO09].

Example 3.7. If we consider the following environment for command ψ:

$$\psi = \left\{ \begin{array}{ll} & \textbf{if } x_1 > 0 \textbf{ then } \{ \\ & \quad x_2 := x_2 + x_3; \\ & \quad x_1 := x_1 - 1; \\ y \rightarrow & \quad \textbf{call}(y) \\ & \} \textbf{ else } \{ \\ & \quad \textbf{skip} \\ & \} \end{array} \right\}$$

and the following Hoare Triple:

$$\{true\} x_1 := x_4; x_2 := 0; \textbf{call}(y) \{x_2 = x_4 \times x_3\},$$

we can use the previous proof system to show that the triple is valid. We search a derivation with the following as conclusion:

$$\vdash \{true\} x_1 := x_4; x_2 := 0; \textbf{call}(y) \{x_2 = x_4 \times x_3\}.$$

Using rule RECURSION we get the following two sub-proofs:

$$\xi \vdash \{true\} x_1 := x_4; x_2 := 0; \textbf{call}(y) \{x_2 = x_4 \times x_3\}, \quad \text{(sub-proof 1)}$$

$$\xi \vdash \{x_2 = x_3 \times (x_4 - x_1) \wedge 0 \leq x_1 \wedge x_1 \leq x_4\} \psi(y) \{x_2 = x_4 \times x_3\}, \quad \text{(sub-proof 2)}$$

where ξ is wisely chosen as follows:

$$\left\{ y \to \left(\begin{array}{c} x_2 = x_3 \times (x_4 - x_1) \wedge 0 \leq x_1 \wedge x_1 \leq x_4, \\ x_2 = x_4 \times x_3 \end{array} \right) \right\}.$$

Proof of sub-proof 1: Using rule for sequence SEQUENCE and $x_2 = x_3 \times (x_4 - x_1) \wedge 0 \leq x_1 \wedge x_1 \leq x_4$ as the intermediate boolean expression, we get the following two sub-proofs:

$$\xi \vdash \{true\} x_1 := x_4; x_2 := 0 \left\{ \begin{array}{c} x_2 = x_3 \times (x_4 - x_1) \wedge \\ 0 \leq x_1 \wedge x_1 \leq x_4 \end{array} \right\} \quad \text{(sub-proof 1.1)}$$

$$\xi \vdash \left\{ \begin{array}{c} x_2 = x_3 \times (x_4 - x_1) \wedge \\ 0 \leq x_1 \wedge x_1 \leq x_4 \end{array} \right\} \textbf{call}(y)\{x_2 = x_4 \times x_3\} \quad \text{(sub-proof 1.2)}$$

Proof of sub-proof 1.1: using rule CONSEQUENCE, and the fact that

$$true \Rightarrow 0 = x_3 \times (x_4 - x_4) \wedge 0 \leq x_4 \wedge x_4 \leq x_4,$$

(as x_4 is a location for a natural, statememt $0 \leq x_4$ is consequence of $true$) we get :

$$\xi \vdash \left\{ \begin{array}{c} 0 = x_3 \times (x_4 - x_4) \wedge \\ 0 \leq x_4 \wedge x_4 \leq x_4 \end{array} \right\} x_1 := x_4; x_2 := 0 \left\{ \begin{array}{c} x_2 = x_3 \times (x_4 - x_1) \wedge \\ 0 \leq x_1 \wedge x_1 \leq x_4 \end{array} \right\}$$

Using the rule for sequence SEQUENCE and $0 = x_3 \times (x_4 - x_1) \wedge 0 \leq x_1 \wedge x_1 \leq x_4$ as the intermediate boolean expression, we get:

$$\xi \vdash \left\{ \begin{array}{c} 0 = x_3 \times (x_4 - x_4) \wedge \\ 0 \leq x_4 \wedge x_4 \leq x_4 \end{array} \right\} x_1 := x_4 \left\{ \begin{array}{c} 0 = x_3 \times (x_4 - x_1) \wedge \\ 0 \leq x_1 \wedge x_1 \leq x_4 \end{array} \right\} \quad \text{(sub-proof 1.1.1)}$$

$$\xi \vdash \left\{ \begin{array}{c} 0 = x_3 \times (x_4 - x_1) \wedge \\ 0 \leq x_1 \wedge x_1 \leq x_4 \end{array} \right\} x_2 := 0 \left\{ \begin{array}{c} x_2 = x_3 \times (x_4 - x_1) \wedge \\ 0 \leq x_1 \wedge x_1 \leq x_4 \end{array} \right\} \quad \text{(sub-proof 1.1.2)}$$

Using axiom ASSIGN and the fact that

$$\begin{array}{c} 0 = x_3 \times (x_4 - x_1) \wedge \\ 0 \leq x_1 \wedge x_1 \leq x_4 \end{array} \equiv \left(\begin{array}{c} x_2 = x_3 \times (x_4 - x_1) \wedge \\ 0 \leq x_1 \wedge x_1 \leq x_4 \end{array} \right) [0/x_2]$$

and

$$\begin{array}{c} 0 = x_3 \times (x_4 - x_4) \wedge \\ 0 \leq x_4 \wedge x_4 \leq x_4 \end{array} \equiv \left(\begin{array}{c} 0 = x_3 \times (x_4 - x_1) \wedge \\ 0 \leq x_1 \wedge x_1 \leq x_4 \end{array} \right) [x_4/x_1]$$

we can prove sub-proof 1.1.1 and sub-proof 1.1.2.

Proof of sub-proof 1.2: Since for procedure y the associated contract corresponds to the triple of Proof of sub-proof 1.2

$$\xi(y) = (x_2 = x_3 \times (x_4 - x_1) \wedge 0 \leq x_1 \wedge x_1 \leq x_4, x_2 = x_4 \times x_3),$$

we can use directly the rule CALL.

Proof of sub-proof 2:

$$\xi \vdash \left\{ \begin{array}{c} x_2 = x_3 \times (x_4 - x_1) \wedge \\ 0 \leq x_1 \wedge x_1 \leq x_4 \end{array} \right\} \psi(y) \{x_2 = x_4 \times x_3\}$$

Using the condition rule CONDITION, we get:

$$\xi \vdash \left\{ \begin{array}{c} x_2 = x_3 \times (x_4 - x_1) \wedge \\ 0 \leq x_1 \wedge x_1 \leq x_4 \wedge \\ \neg x_1 > 0 \end{array} \right\} \textbf{skip} \{x_2 = x_4 \times x_3\} \qquad \text{(sub-proof 2.1)}$$

$$\xi \vdash \left\{ \begin{array}{c} x_2 = x_3 \times (x_4 - x_1) \wedge \\ 0 \leq x_1 \wedge x_1 \leq x_4 \wedge \\ x_1 > 0 \end{array} \right\} \begin{array}{l} x_2 := x_2 + x_3 \\ x_1 := x_1 - 1; \ \{x_2 = x_4 \times x_3\} \\ \textbf{call}(y) \end{array} \qquad \text{(sub-proof 2.2)}$$

Proof of sub-proof 2.1: using rule CONSEQUENCE and the fact that :

$$x_2 = x_3 \times (x_4 - x_1) \wedge 0 \leq x_1 \wedge x_1 \leq x_4 \wedge \neg(x_1 > 0) \Rightarrow x_2 = x_3 \times x_4$$

we get

$$\xi \vdash \{x_2 = x_4 \times x_3\} \textbf{skip} \{x_2 = x_4 \times x_3\}$$

that can be proven using axiom SKIP.

Proof of sub-proof 2.2: using the rule for sequence SEQUENCE and choosing $x_2 = x_3 \times (x_4 - x_1) \wedge 0 \leq x_1 \wedge x_1 \leq x_4 \wedge x_1 > 0$ as the intermediate boolean expression, we get:

$$\xi \vdash \left\{ \begin{array}{c} x_2 = x_3 \times (x_4 - x_1) \wedge \\ 0 \leq x_1 \wedge x_1 \leq x_4 \wedge \\ x_1 > 0 \end{array} \right\} \begin{array}{l} x_2 := x_2 + x_3 \\ x_1 := x_1 - 1; \end{array} \left\{ \begin{array}{c} x_2 = x_3 \times (x_4 - x_1) \wedge \\ 0 \leq x_1 \wedge x_1 \leq x_4 \wedge \\ x_1 >= 0 \end{array} \right\}$$

(sub-proof 2.2.1)

$$\xi \vdash \left\{ \begin{array}{c} x_2 = x_3 \times (x_4 - x_1) \wedge \\ 0 \leq x_1 \wedge x_1 \leq x_4 \wedge \\ x_1 >= 0 \end{array} \right\} \textbf{call}(y) \{x_2 = x_4 \times x_3\} \qquad \text{(sub-proof 2.2.2)}$$

Proof of sub-proof 2.2.1: Using the rule for sequence SEQUENCE and choosing $x_2 = x_3 \times (x_4 - (x_1 - 1)) \wedge 0 \leq x_1 - 1 \wedge x_1 - 1 \leq x_4 \wedge x_1 - 1 >= 0$ as the intermediate boolean expression, we get:

$$\xi \vdash \left\{ \begin{array}{c} x_2 = x_3 \times (x_4 - x_1) \wedge \\ 0 \leq x_1 \wedge x_1 \leq x_4 \wedge \\ x_1 > 0 \end{array} \right\} x_2 := x_2 + x_3 \left\{ \begin{array}{c} x_2 = x_3 \times (x_4 - (x_1 - 1)) \wedge \\ 0 \leq x_1 - 1 \wedge x_1 - 1 \leq x_4 \wedge \\ x_1 - 1 >= 0 \end{array} \right\}$$

(sub-proof 2.2.1.1)

$$\xi \vdash \left\{ \begin{array}{c} x_2 = x_3 \times (x_4 - (x_1 - 1)) \wedge \\ 0 \leq x_1 - 1 \wedge x_1 - 1 \leq x_4 \wedge \\ x_1 - 1 >= 0 \end{array} \right\} x_1 := x_1 - 1 \left\{ \begin{array}{c} x_2 = x_3 \times (x_4 - x_1) \wedge \\ 0 \leq x_1 \wedge x_1 \leq x_4 \wedge \\ x_1 - 1 >= 0 \end{array} \right\}$$

(sub-proof 2.2.1.2)

By noticing that

$$\begin{array}{c} x_2 = x_3 \times (x_4 - x_1) \wedge \\ 0 \leq x_1 \wedge x_1 \leq x_4 \wedge \\ x_1 > 0 \end{array} \quad \Rightarrow \quad \begin{array}{c} x_2 + x_3 = x_3 \times (x_4 - (x_1 - 1)) \wedge \\ 0 \leq x_1 - 1 \wedge x_1 - 1 \leq x_4 \wedge \\ x_1 - 1 >= 0 \end{array}$$

and

$$\begin{array}{c} x_2 + x_3 = x_3 \times (x_4 - (x_1 - 1)) \wedge \\ 0 \leq x_1 - 1 \wedge x_1 - 1 \leq x_4 \wedge \\ x_1 - 1 >= 0 \end{array} \equiv \left(\begin{array}{c} x_2 = x_3 \times (x_4 - (x_1 - 1)) \wedge \\ 0 \leq x_1 - 1 \wedge x_1 - 1 \leq x_4 \wedge \\ x_1 - 1 >= 0 \end{array} \right) [x_2 + x_3 / x_2]$$

and

$$\begin{array}{c} x_2 = x_3 \times (x_4 - (x_1 - 1)) \wedge \\ 0 \leq x_1 - 1 \wedge x_1 - 1 \leq x_4 \wedge \\ x_1 - 1 >= 0 \end{array} \equiv \left(\begin{array}{c} x_2 = x_3 \times (x_4 - x_1) \wedge \\ 0 \leq x_1 \wedge x_1 \leq x_4 \wedge \\ x_1 >= 0 \end{array} \right) [x_1 - 1 / x_1]$$

and using rule CONSEQUENCE and ASSIGN we can prove sub-proof 2.2.1.1 and sub-proof 2.2.1.2.

Proof of sub-proof 2.2.2: Using rule CONSEQUENCE, CALL and noticing that

$$\begin{array}{c} x_2 = x_3 \times (x_4 - x_1) \wedge \\ 0 \leq x_1 \wedge x_1 \leq x_4 \wedge \\ x_1 >= 0 \end{array} \quad \Rightarrow \quad \begin{array}{c} x_2 = x_3 \times (x_4 - x_1) \wedge \\ 0 \leq x_1 \wedge x_1 \leq x_4 \end{array}$$

and

$$\xi(y) = (x_2 = x_3 \times (x_4 - x_1) \wedge 0 \leq x_1 \wedge x_1 \leq x_4, x_2 = x_4 \times x_3)$$

The previous Example 3.7 shows that Hoare logic is a powerful method for properties on programs. However, the example also shows that it is not always easy to guess the right boolean expression in case of rules CONSEQUENCE and SEQUENCE. Moreover, the system quickly becomes cumbersome. Chapter 5 shows how to solve this difficulty.

Chapter 4

Background on Relational Property Verification

This chapter focuses on the presentation of relational properties. It follows the same outline as [BCK16]. First, we propose a formal definition of relational properties in Section 4.1. Based on this definition, we present three existing deductive methods for proving validity of relational properties: Relational Hoare Logic (Section 4.2), Self-Composition (Section 4.3) and Product Program (Section 4.4). We discuss their benefits and limitations and the support for the construct **call**() in the context of the R-WHILE language. In order to distinguish the syntactic elements and functions defined in this chapter from those in the previous chapter, we add the symbol ˜ on each new syntactic elements and function.

4.1 Relational Properties

We mentioned in Chapter 1 that standard axiomatic semantics relates one command c to two boolean expressions b_1 and b_2, respectively the pre- and post-condition. We recall the definition of triple $\{b_1\}c\{b_2\}$ given in Chapter 3 for a fixed environment ψ (mapping program names to commands):

$$\forall \sigma \in \Sigma. \sigma \models b_1 \Rightarrow \dot{\xi}_c[\![c]\!](\sigma, \psi) \models b_2$$

As already noted in Chapter 1, it can happen that we want to reason about more than one program. More precisely, we want to express the following statement:

$$\forall (\sigma_1, ..., \sigma_n) \in \Sigma^n. (\sigma_1, ..., \sigma_n) \models \tilde{b}_1 \Rightarrow (\dot{\xi}_c[\![c_1]\!](\sigma_1, \psi_1), ..., \dot{\xi}_c[\![c_n]\!](\sigma_n, \psi_n)) \models \tilde{b}_2$$

That is, n programs executed from n states verifying a property $\tilde{b}_1$ lead to n states verifying another property $\tilde{b}_2$. Function $\dot{\xi}_c$ (introduced in Section 3.3) calls evaluation function for commands ξ_c with an arbitrary amount of fuel n.

In the following, we give a formal syntax and semantics for the evaluation of relational properties. To avoid cumbersome notations due to the manipulation of tuples, we propose the use of functions for handling the set of memory locations and programs.

We define two new categories of sets $\tilde{\mathbb{E}}_a$ for relational arithmetic expressions and $\tilde{\mathbb{E}}_b$ for relational boolean expressions. We use metavariables $\tilde{a}, \tilde{a}_0, \tilde{a}_1, ...$ to range over $\tilde{\mathbb{E}}_a$ and $\tilde{b}, \tilde{b}_0, \tilde{b}_1, ...$ to range over $\tilde{\mathbb{E}}_b$. We also define the set $\mathbb{T}$ for tags, used to distinguish between memory states, and use metavariables $t, t_0, t_1, ...$ to range over $\mathbb{T}$.

Using these new sets, we define the following grammar rules for relational arithmetic expressions and relational boolean expressions:

$$\begin{aligned}
\tilde{a} &::= n \\
&\mid x\langle t\rangle \\
&\mid \tilde{a}\ opa\ \tilde{a} \\
\tilde{b} &::= true \mid false \\
&\mid \tilde{a}_1\ opb\ \tilde{a}_2 \\
&\mid \tilde{b}_1\ opl\ \tilde{b}_2 \mid \neg\tilde{b}'
\end{aligned}$$

The only difference with the grammar rules proposed in Section 3.2, is the use of notation $\langle t\rangle$ in the case of arithmetic expressions. This notation has been proposed by Benton [Ben04] to distinguish between memory locations from different memory states.

We define Φ, the relational state environment that maps tags to memory states,

$$\Phi = \mathbb{T} \rightharpoonup \Sigma$$

and use metavariables $\phi, \phi_0, \phi_1, ...$ to range over Φ.

Using environment Φ, we define the evaluation function for $\tilde{\mathbb{E}}_a$ and $\tilde{\mathbb{E}}_b$ as follows.

Definition 4.1. Evaluation function $\tilde{\xi}_a : \tilde{\mathbb{E}}_a \to (\Phi \to \mathbb{N}_\perp)$, for relational arithmetic expressions $\tilde{\mathbb{E}}_a$, is defined by structural induction on relational arithmetic expressions:

$$\begin{aligned}
\tilde{\xi}_a[\![n]\!]\phi &= n \\
\tilde{\xi}_a[\![x\langle t\rangle]\!]\phi &= (\phi(t))(x) \\
\tilde{\xi}_a[\![\tilde{a}_0\ opa\ \tilde{a}_1]\!]\phi &= \tilde{\xi}_a[\![\tilde{a}_0]\!]\phi\ opa_\perp\ \tilde{\xi}_a[\![\tilde{a}_1]\!]\phi.
\end{aligned}$$

Definition 4.2. Evaluation function $\tilde{\xi}_b : \tilde{\mathbb{E}}_b \to (\Phi \to \mathbb{B}_\perp)$, for relational boolean expressions $\tilde{\mathbb{E}}_b$, is defined by structural induction on relational boolean expressions:

$$\begin{aligned}
\tilde{\xi}_b[\![true]\!]\phi &= true \\
\tilde{\xi}_b[\![false]\!]\phi &= false \\
\tilde{\xi}_b[\![\tilde{a}_0 \ opb \ \tilde{a}_1]\!]\phi &= \tilde{\xi}_a[\![\tilde{a}_0]\!]\phi \ opb_\perp \ \tilde{\xi}_a[\![\tilde{a}_1]\!]\phi \\
\tilde{\xi}_b[\![\tilde{b}_0 \ opl \ \tilde{b}_1]\!]\phi &= \tilde{\xi}_b[\![\tilde{b}_0]\!]\phi \ opl_\perp \ \tilde{\xi}_b[\![\tilde{b}_1]\!]\phi \\
\tilde{\xi}_b[\![\neg\tilde{b}]\!]\phi &= \neg_\perp \tilde{\xi}_b[\![\tilde{b}]\!]\phi.
\end{aligned}$$

We define Φ_c, the relational execution environment that maps tags to the pair composed of a command and a memory state for commands:

$$\Phi_c = \mathbb{T} \rightharpoonup \mathbb{C} \times \Psi.$$

We use metavariables $\phi_c, \phi_{c0}, \phi_{c1}, ...$ to range over Φ_c. To simplify reading, we define the following projections to access the command and the memory state for commmand in a pair bound to a tag in an environment ϕ_c:

- $body : \mathbb{C} \times \Psi \to \mathbb{C}$,
- $state : \mathbb{C} \times \Psi \to \Psi$.

We now lift the set Φ with Ω

$$\Phi_\Omega = \mathbb{T} \rightharpoonup \Sigma_\Omega,$$

and define evaluation function $\tilde{\xi}_c$ for the evaluation of relational execution environment ϕ_c.

Definition 4.3. Evaluation function $\tilde{\xi}_c : \Phi_c \times \Phi_\Omega \to \Phi_\Omega$, for relational execution environment Φ_c, is defined by:

$$\tilde{\xi}_c(\phi_c, \phi) = \phi[t_1 \leftarrow \sigma_1]...[t_n \leftarrow \sigma_n]$$

where

(i) $\quad \{t_1, ..., t_n\} = dom(\phi_c)$,

(ii) $\quad \sigma_i = \dot{\xi}_c[\![body(\phi_c(t_i))]\!](\phi(t), state(\phi_c(t_i)))$.

Function $\tilde{\xi}_c$ evaluates all commands defined in ϕ_c with the associated memory state defined in environment ϕ using evaluation function $\dot{\xi}_c$ for simple commands $\mathbb{C}$. The environment ϕ is updated with the resulting state.

In the future we assume that the following statements are always satisfied:

- There is no command evaluation on undefined memory states:

$$dom(\phi_c) \subseteq dom(\phi).$$

- The sets of tags used in the relational boolean expressions $\tilde{b}_1$ and $\tilde{b}_2$ are defined in ϕ_c:

$$\tilde{\mathcal{C}}_{t_b}[\![\tilde{b}_1]\!] \subseteq dom(\phi_c),$$
$$\tilde{\mathcal{C}}_{t_b}[\![\tilde{b}_2]\!] \subseteq dom(\phi_c),$$

  where function $\tilde{\mathcal{C}}_{t_b}$ (defined in Appendix A.1.4) returns the set of tags used in a relational boolean expression.

- The sets of locations used in the context of a tag in $\tilde{b}_1$ and $\tilde{b}_2$ are defined in the memory state associated to that tag:

$$\forall t \in \tilde{\mathcal{C}}_{t_b}[\![\tilde{b}_1]\!], \tilde{\mathcal{C}}_{v_b}[\![\tilde{b}_1]\!]t \subseteq dom(\phi(t)),$$

$$\forall t \in \tilde{\mathcal{C}}_{t_b}[\![\tilde{b}_2]\!], \tilde{\mathcal{C}}_{v_b}[\![\tilde{b}_2]\!]t \subseteq dom(\phi(t)),$$

  where function $\tilde{\mathcal{C}}_{v_b}$ (defined in Appendix A.1.4) returns the set of locations associated to a given tag in a relational boolean expression.

- The set of triples formed by a memory state ψ, σ and a command c for a given tag are well defined:

$$\forall t \in dom(\phi_c).\mathcal{WD}(body(\phi_c(t)), state(\phi_c(t)), \phi(t)),$$

  where statement $\mathcal{WD}$ is defined in Section 3.3.

As for the Hoare triple, those hypotheses ensure that state $\Omega_\perp$ is not occurring.

Assuming the previouse hpotheses and using function $\tilde{\xi}_c$, we can give our definition of relational properties for a given relational execution environment ϕ_c:

Definition 4.4. We call relational property the statement

$$\forall \phi \in \Phi.\phi \models \tilde{b}_1 \Rightarrow \tilde{\xi}_c(\phi_c, \phi) \models \tilde{b}_2$$

stating that, if an environment ϕ satisfies the relational boolean expression $\tilde{b}_1$, and if evaluation of the relational execution environment ϕ_c on ϕ terminates, the resulting environment satisfies the relational boolean expression $\tilde{b}_2$.

We define judgement

$$\phi \models \tilde{b},$$

indicating that $\tilde{\xi}_b[\![\tilde{b}]\!]\phi$ evaluates to $true$ or there is an evaluation that has not finished:

$$\exists t \in dom(\phi).\phi(t) = \Omega_t.$$

In the rest of this chapter, we will present existing methods for the verification of relational properties. For that, we will focus on relations over two executions. In other words, we will consider functions $\phi \in \Phi$ such that $dom(\phi) \subseteq \{t_1, t_2\}$ and $\phi_c \in \Phi_c$ such that $dom(\phi_c) \subseteq \{t_1, t_2\}$. We note this set of environments Φ^2 and Φ_c^2.

In this context, we use the notation proposed in [Ben04] to denote relational properties :

$$\{\tilde{b}_1\}c_1\langle t_1\rangle \sim c_2\langle t_2\rangle\{\tilde{b}_2\}.$$

This notation has the benefit of being simple and short to write (by omitting the memory state for command ψ). We interpret the notation as an application of function $\tilde{\xi}_c$ on an environment ϕ, satisfying relational boolean expression $\tilde{b}_1$, and an environment ϕ_c, with command c_1 defined for tag t_1 and command c_2 defined for tag t_2, and resulting in an environment satisfying relational boolean expression $\tilde{b}_2$:

$$\forall \phi \in \Phi^2.\phi \models \tilde{b}_1 \Rightarrow \phi[t_1 \leftarrow \sigma_1][t_2 \leftarrow \sigma_2] \models \tilde{b}_2$$

where
(i) $\quad body(\phi_c(t_1)) = c_1,$
(ii) $\quad body(\phi_c(t_2)) = c_2,$
(iii) $\quad \sigma_1 = \dot{\xi}_c[\![body(\phi_c(t_1))]\!](\phi(t_1), state(\phi_c(t_1))),$
(iv) $\quad \sigma_2 = \dot{\xi}_c[\![body(\phi_c(t_2))]\!](\phi(t_2), state(\phi_c(t_2))).$

4.2 Relational Hoare Logic

In Section 3.3 we presented Hoare Logic for proving validity of a Hoare triple. As relational properties are an extension of standard properties, proposing a similar verification approach seems natural. Benton introduced in [Ben04] Relational Hoare Logic for the verification of relational properties relating two programs.

4.2.1 Minimal Relational Hoare Logic

Relational Haore Logic consider programs executed in locksteps *i.e.* both commands have the same shape for each rule. As for Hoare Logic, a relational property is valid ($\models \{\tilde{b}_1\}c_1\langle t_1\rangle \sim c_2\langle t_2\rangle\{\tilde{b}_2\}$) if $\vdash \{\tilde{b}_1\}c_1\langle t_1\rangle \sim c_2\langle t_2\rangle\{\tilde{b}_2\}$ is a conclusion of a proof in the following rule system:

$$\frac{}{\vdash \{\tilde{b}\}\textbf{skip}\langle t_1\rangle \sim \textbf{skip}\langle t_2\rangle\{\tilde{b}\}} \quad \text{(R-SKIP)}$$

$$\frac{\vdash\ \tilde{b} \Rightarrow b_1\langle t_1\rangle \wedge b_2\langle t_2\rangle}{\vdash \{\tilde{b}\}\textbf{assert}(b_1)\langle t_1\rangle \sim \textbf{assert}(b_2)\langle t_2\rangle\{\tilde{b} \wedge b_1\langle t_1\rangle \wedge b_2\langle t_2\rangle\}} \quad \text{(R-ASSERT)}$$

$$\frac{}{\vdash \{\tilde{b}[a_1\langle t_1\rangle/x_1\langle t_1\rangle, a_2\langle t_2\rangle/x\langle t_2\rangle]\}x_1 := a_1\langle t_1\rangle \sim x_2 := a_2\langle t_2\rangle\{\tilde{b}\}} \quad \text{(R-ASSIGN)}$$

$$\frac{\vdash \{\tilde{b}_1\}c_1\langle t_1\rangle \sim c_3\langle t_2\rangle\{\tilde{b}_2\} \qquad \vdash \{\tilde{b}_2\}c_2\langle t_1\rangle \sim c_4\langle t_2\rangle\{\tilde{b}_3\}}{\vdash \{\tilde{b}_1\}c_1; c_2\langle t_1\rangle \sim c_3; c_4\langle t_2\rangle\{\tilde{b}_3\}} \quad \text{(R-SEQUENCE)}$$

$$\frac{\vdash \{\tilde{b}_1 \wedge b_1\langle t_1\rangle \wedge b_2\langle t_2\rangle\} c_1\langle t_1\rangle \sim c_3\langle t_2\rangle \{\tilde{b}_2\} \quad \vdash \{\tilde{b}_1 \wedge \neg b_1\langle t_1\rangle \wedge \neg b_2\langle t_2\rangle\} c_2\langle t_1\rangle \sim c_4\langle t_2\rangle \{\tilde{b}_2\}}{\vdash \{\tilde{b}_1 \wedge b_1\langle t_1\rangle \equiv b_2\langle t_2\rangle\} \textbf{if } b_1 \textbf{ then } \{c_1\} \textbf{ else } \{c_2\}\langle t_1\rangle \sim \textbf{if } b_2 \textbf{ then } \{c_3\} \textbf{ else } \{c_4\}\langle t_2\rangle \{\tilde{b}_2\}} \text{ (R-CONDITION)}$$

$$\frac{\vdash \{\tilde{b} \wedge b_1\langle t_1\rangle \wedge b_2\langle t_2\rangle\} c_1 \sim c_2 \{\tilde{b} \wedge b_1\langle t_1\rangle \equiv b_2\langle t_2\rangle\}}{\vdash \{\tilde{b} \wedge b_1\langle t_1\rangle \equiv b_2\langle t_2\rangle\} \textbf{while } b_1 \textbf{ do } \{c_1\}\langle t_1\rangle \sim \textbf{while } b_2 \textbf{ do } \{c_2\}\langle t_2\rangle \{\tilde{b} \wedge \neg(b_1\langle t_1\rangle \vee b_2\langle t_2\rangle)\}} \text{ (R-WHILE)}$$

$$\frac{\models (\tilde{b}_1 \Rightarrow \tilde{b}'_1) \qquad \vdash \{\tilde{b}'_1\} c_1\langle t_1\rangle \sim c_2\langle t_2\rangle \{\tilde{b}'_2\} \qquad \models (\tilde{b}'_2 \Rightarrow \tilde{b}_2)}{\vdash \{\tilde{b}_1\} c_1\langle t_1\rangle \sim c_2\langle t_2\rangle \{\tilde{b}_2\}} \text{ (R-CONSEQUENCE)}$$

$a\langle t\rangle$ is a shorthand for the application of function $\tilde{\mathcal{A}}_{t_a}$ (defined in Appendix A.5.1). The function adds tag t to an arithmetic expression a of $\mathbb{E}_a$. For instance, tagging $x_1 + x_2$ yields the following result:

$$\tilde{\mathcal{A}}_{t_a}[\![x_1 + x_2]\!]t \equiv x_1\langle t\rangle + x_2\langle t\rangle$$

Similarly, $b\langle t\rangle$ is a shorthand for the application of function $\tilde{\mathcal{A}}_{t_b}$ (defined in Appendix A.5.1). The function adds tag t to a boolean expression b of $\mathbb{E}_b$. For instance, tagging $x_1 < x_2$ yields the following result:

$$\tilde{\mathcal{A}}_{t_b}[\![x_1 < x_2]\!]t \equiv x_1\langle t\rangle < x_2\langle t\rangle$$

$\tilde{b}_1 \equiv \tilde{b}_2$ is a shorthand for the encoding of equivalence in relational boolean expression $((\neg\tilde{b}_1 \wedge \neg\tilde{b}_2) \vee (\tilde{b}_1 \wedge \tilde{b}_2))$.

Most rules are a straightforward extension of the original rules from Hoare Logic, adapted to support two commands. Note that the while rule R-WHILE forces the loops to be synchronized, since the two loop conditions must have the same boolean value (a similar requirement exists in the case of rule R-CONDITION for condition).

Example 4.1. We consider the following relational property

$$\{x\langle t_1\rangle = x\langle t_2\rangle\} x := x + 1\langle t_1\rangle \sim x := x + 2\langle t_2\rangle \{x\langle t_1\rangle + 1 = x\langle t_2\rangle\}.$$

We can use the previous proof system to show that this quadruple is valid.

$$\vdash \{x\langle t_1\rangle = x\langle t_2\rangle\} x := x + 1\langle t_1\rangle \sim x := x + 2\langle t_2\rangle \{x\langle t_1\rangle + 1 = x\langle t_2\rangle\}.$$

Using rule R-CONSEQUENCE and the fact that $x\langle t_1\rangle = x\langle t_2\rangle \Rightarrow x\langle t_1\rangle + 2 = x\langle t_2\rangle + 2$, we get:

$$\vdash \{x\langle t_1\rangle + 2 = x\langle t_2\rangle + 2\} x := x + 1\langle t_1\rangle \sim x := x + 2\langle t_2\rangle \{x\langle t_1\rangle + 1 = x\langle t_2\rangle\}.$$

Which is equivalent to

$$\vdash \{(x\langle t_1\rangle + 1 = x\langle t_2\rangle)[r]\} x := x + 1\langle t_1\rangle \sim x := x + 2\langle t_2\rangle \{x\langle t_1\rangle + 1 = x\langle t_2\rangle\},$$

with $r = \{x\langle t_1\rangle + 1/x\langle t_1\rangle, x\langle t_2\rangle + 2/x\langle t_2\rangle\}$. Using axiom R-ASSIGN, we prove the quadruple.

4.2.2 Extended Minimal Relational Hoare Logic

Since program are considering executed in locksteps, it is hard to compare programs with different structures. To solve this limitations, additional rules can be added to the system. For example, in the cases of a relational property between two programs with different number of assignment commands, rule R-SI can be used to introduce a command **skip** and axiom R-DA to prove properties between a **skip** command and an assign command (the symmetrical versions of these rules are also true).

$$\frac{\vdash \{\tilde{b}_1\}c_1\langle t_1\rangle \sim (\textbf{skip}; c_2)\langle t_2\rangle\{\tilde{b}_2\}}{\vdash \{\tilde{b}_1\}c_1\langle t_2\rangle \sim c_2\langle t_2\rangle\{\tilde{b}_2\}} \quad \text{(R-SI)}$$

$$\frac{}{\vdash \{\tilde{b}[a\langle t_1\rangle/x\langle t_1\rangle]\}x := a\langle t_1\rangle \sim \textbf{skip}\langle t_2\rangle\{\tilde{b}\}} \quad \text{(R-DA)}$$

Example 4.2. We consider a modified version of the relational property shown in Example 4.1,

$$\{x\langle t_1\rangle = x\langle t_2\rangle\} \begin{array}{l} x := x+1; \\ x := x+1 \end{array} \langle t_1\rangle \sim x := x+2\langle t_2\rangle\{x\langle t_1\rangle = x\langle t_2\rangle\},$$

and want to prove validity of the quadruple.

$$\vdash \{x\langle t_1\rangle = x\langle t_2\rangle\} \begin{array}{l} x := x+1; \\ x := x+1 \end{array} \langle t_1\rangle \sim x := x+2\langle t_2\rangle\{x\langle t_1\rangle = x\langle t_2\rangle\}.$$

We can first use rule R-SI to add a skip command.

$$\vdash \{x\langle t_1\rangle = x\langle t_2\rangle\} \begin{array}{l} x := x+1; \\ x := x+1 \end{array} \langle t_1\rangle \sim \begin{array}{l} \textbf{skip}; \\ x := x+2 \end{array} \langle t_2\rangle\{x\langle t_1\rangle = x\langle t_2\rangle\}.$$

We can then use rule R-ASSIGN to split the proof in two sub proofs.

$$\begin{cases} \vdash \{x\langle t_1\rangle = x\langle t_2\rangle\}x := x+1\langle t_1\rangle \sim \textbf{skip}\langle t_2\rangle\{x\langle t_1\rangle + 1 = x\langle t_2\rangle + 2\} \\ \\ \vdash \{x\langle t_1\rangle + 1 = x\langle t_2\rangle + 2\}x := x+1\langle t_1\rangle \sim x := x+2\langle t_2\rangle\{x\langle t_1\rangle = x\langle t_2\rangle\} \end{cases}$$

Using the fact that $x\langle t_1\rangle = x\langle t_2\rangle \Rightarrow x\langle t_1\rangle + 2 = x\langle t_2\rangle + 2$ and $r = \{x\langle t_1\rangle + 1/x\langle t_1\rangle, x\langle t_2\rangle + 2/x\langle t_2\rangle\}$, we get the following rules:

$$\begin{cases} \vdash \{x\langle t_1\rangle + 2 = x\langle t_2\rangle + 2\}x := x+1\langle t_1\rangle \sim \textbf{skip}\langle t_2\rangle\{x\langle t_1\rangle + 1 = x\langle t_2\rangle + 2\} \\ \\ \vdash \{x\langle t_1\rangle = x\langle t_2\rangle[r]\}x := x+1\langle t_1\rangle \sim x := x+2\langle t_2\rangle\{x\langle t_1\rangle = x\langle t_2\rangle\} \end{cases}$$

We can rewrite the first rule to get the following rules:

$$\begin{cases} \vdash \{x\langle t_1\rangle + 1 = x\langle t_2\rangle + 2[x\langle t_1\rangle + 1/x\langle t_1\rangle]\}x := x+1\langle t_1\rangle \sim \textbf{skip}\langle t_2\rangle\{x\langle t_1\rangle + 1 = x\langle t_2\rangle + 2\} \\ \\ \vdash \{x\langle t_1\rangle = x\langle t_2\rangle[r]\}x := x+1\langle t_1\rangle \sim x := x+2\langle t_2\rangle\{x\langle t_1\rangle = x\langle t_2\rangle\} \end{cases}$$

Using axioms R-ASSIGN and R-DA respectively, we prove both quadruples.

Since many rules exist for handling dissimilar programs, we refer interested readers to [Ben04, BCK16] to get more rules. Those rules are often symmetric and/or inverted and trivial except for the case of the loop which is as usual more difficult.

In general, Relational Hoare Logic requires many additional rules to get a system powerful enough to support differently structured programs. As a result, it is not always easy to find which rules must be applied and which property must be inferred. Finally, Relational Hoare Logic requires the support of relational boolean expression $\tilde{b}$ and handles two programs at the same time. This makes it difficult to use in existing verification tools.

4.2.3 Relational Hoare Logic and Procedures

The literature offers no rules for handling procedure calls **call**() in relational Hoare Logic, although it is possible to solve this problem by connecting Relational Hoare Logic to the proof system presented in Section 3.3 for Hoare Triples, as we show below.

First, we define environment Φ_a associating a tag to an environment of procedure contracts

$$\Phi_a = \mathbb{T} \rightharpoonup \Xi.$$

As for the simple Hoare Triple, we assume that the sets of variables used in the contracts for a given tag are subsets of the domain of the memory state for the given tag.

$$\forall t \in dom(\phi_a).\mathcal{W}_a(\phi_a(t), \phi(t))$$

Moreover, we assume that there is at most one set of contracts for each tag belonging to the domaine of ϕ_c:

$$dom(\phi_a) = dom(\phi_c)$$

Then, we can notice that a relational property that links only one tag is actually a Hoare triple. This is formally described by rule R-CALL

$$\frac{\phi_a(t_1) \vdash \{\tilde{\mathcal{D}}_{t_b}[\![\tilde{b}_1]\!]\}c_1\{\tilde{\mathcal{D}}_{t_b}[\![b_2]\!]\} \qquad \tilde{\mathcal{C}}_{t_b}[\![\tilde{b}_1]\!] = \{t_1\} \qquad \tilde{\mathcal{C}}_{t_b}[\![\tilde{b}_2]\!] = \{t_1\}}{\phi_a \vdash \{\tilde{b}_1\}c_1\langle t_1\rangle\{\tilde{b}_2\}} \text{(R-CALL)}$$

where function $\tilde{\mathcal{D}}_{t_b}$ (defined in Appendix A.4.1) removes all tags from relational boolean expression and function $\tilde{\mathcal{C}}_{t_b}$ (defined in Appendix A.1.4) returns the set of tags used in a relational boolean expression. The fact that $\tilde{\mathcal{C}}_{t_b}[\![\tilde{b}_1]\!] = \{t_1\}$ and $\tilde{\mathcal{C}}_{t_b}[\![\tilde{b}_2]\!] = \{t_1\}$ is ensured by the hypotheses on relational properties.

Then, we can define rule R-COMBINE that splits a relational property.

$$\frac{\begin{array}{c}\phi_a \vdash \{\tilde{b}_{1\_t_1}\}c_1\langle t_1\rangle\{\tilde{b}_{2\_t_1}\} \qquad \phi_a \vdash \{\tilde{b}_{1\_t_2}\}c_2\langle t_2\rangle\{\tilde{b}_{2\_t_2}\} \\ \tilde{\mathcal{C}}_{t_b}[\![\tilde{b}_{1\_t_1}]\!] = \{t_1\} \qquad \tilde{\mathcal{C}}_{t_b}[\![\tilde{b}_{2\_t_1}]\!] = \{t_1\} \qquad \tilde{\mathcal{C}}_{t_b}[\![\tilde{b}_{1\_t_2}]\!] = \{t_2\} \qquad \tilde{\mathcal{C}}_{t_b}[\![\tilde{b}_{2\_t_2}]\!] = \{t_2\}\end{array}}{\phi_a \vdash \{\tilde{b}_{1\_t_1} \wedge \tilde{b}_{1\_t_2}\}c_1\langle t_1\rangle \sim c_2\langle t_2\rangle\{\tilde{b}_{2\_t_1} \wedge \tilde{b}_{2\_t_2}\}} \text{(R-COMBINE)}$$

Notice that we can split a relational property only if the relational boolean expression does not link locations with different tags. This is an important restriction. In particular, the rule cannot be applied on the previous examples of the form $x\langle t_1\rangle = x\langle t_2\rangle$.

Finally, we define rule R-RECURSION, an extension of rule RECURSION.

$$\frac{\forall t \in dom(\phi_a).\forall y \in dom(\phi_a(t)).\phi_a(t) \vdash \{fst(\phi_a(t)(y))\}state(\phi_c(t))(y)\{snd(\phi_a(t)(y))\} \quad \phi_a \vdash \{\tilde{b}_1\}c_1\langle t_1\rangle \sim c_2\langle t_2\rangle\{\tilde{b}_2\}}{\vdash \{\tilde{b}_1\}c_1\langle t_1\rangle \sim c_2\langle t_2\rangle\{\tilde{b}_2\}} \text{(R-RECURSION)}$$

The combination of rules R-COMBINE and R-RECURSION allow the use of standard Hoare Triples in the context of the proof of relational properties. Unfortunately, Hoare Triples are not always practical, as relational boolean expressions linking locations with different tags are not supported. Thus, it is required to find an equivalent form that do not link locations with different tags, which is not always trivial.

4.3 Self-Composition

An alternative to Relational Hoare Logic is Self-Composition. Self-composition is a theoretical approach to prove relational properties by reducing the verification of a relational property to the verification of a Hoare Triple.

In this section, we give an overview of self-composition in the context of our definition of relational properties. A proof of this reasoning can be found in [BDR11].

First, the definition 4.4 of relational properties can be refined as follows (for a given $\phi_c \in \Phi_c^2$):

$$\forall \phi \in \Phi^2.\phi \models \tilde{b}_1 \Rightarrow \tilde{\xi}_c(\{t_2 \to \phi_c(t_2)\}, \tilde{\xi}_c(\{t_1 \to \phi_c(t_1)\}, \phi)) \models \tilde{b}_2 \tag{4.1}$$

This definition corresponds to composing in sequence function $\tilde{\xi}_c$ over the domain of ϕ_c.

Now, if for both tags in ϕ the associated memory states do not share locations, *i.e.*

$$dom(\phi(t_1)) \cap dom(\phi(t_2)) = \emptyset,$$

we can merge the two memory states into one memory state using a merging function:

$$\tilde{\mathcal{M}}(\phi) = \{t_s \to \phi(t_1) \cup \phi(t_2)\}.$$

Afterwards, we change all tags in the boolean expressions $\tilde{b}_1$ and $\tilde{b}_2$ into one single tag using function $\tilde{\mathcal{R}}_{tt_b} : \tilde{\mathbb{E}}_b \to (\mathbb{T} \to \tilde{\mathbb{E}}_b)$ (defined in Appendix A.3.2) with the property that for a given relational boolean expression and a given tag t, we have $\tilde{\mathcal{C}}_{t_b}[\![\tilde{\mathcal{R}}_{tt_b}[\![\tilde{b}]\!]t]\!] = \{t\}$. Similarly, we change the environments ϕ_c such that the evaluation of commands $\phi_c(t_1)$ and $\phi_c(t_2)$ is performed on the same state in environment ϕ. We get the following statement from 4.1:

$$\forall \phi \in \Phi^2.\tilde{\mathcal{M}}(\phi) \models \tilde{\mathcal{R}}_{tt_b}[\![\tilde{b}_1]\!]t_s \Rightarrow \tilde{\xi}_c(\{t_s \to \phi_c(t_2)\}, \tilde{\xi}_c(\{t_s \to \phi_c(t_1)\}, \tilde{\mathcal{M}}(\phi))) \models \tilde{\mathcal{R}}_{tt_b}[\![\tilde{b}_2]\!]t_s \tag{4.2}$$

By noticing that composing in sequence function $\tilde{\xi}_c$ on environments ϕ_c having as domain a single tag is equivalent to the sequence composition of both commands, we can define an environment ϕ_{cs} by:

$$\{t_s \to \{body(\phi_c(t_1)); body(\phi_c(t_2)), state(\phi_c(t_1)) \cup state(\phi_c(t_2))\},$$

where we assume that $dom(state(\phi_c(t_1))) \cap dom(state(\phi_c(t_2))) = \emptyset$. We get the following statement from 4.2:

$$\forall \phi \in \Phi^2.\tilde{\mathcal{M}}(\phi) \models \tilde{\mathcal{R}}_{tt_b}[\![\tilde{b}_1]\!]t_s \Rightarrow \tilde{\xi}_c(\phi_{cs}, \tilde{\mathcal{M}}(\phi)) \models \tilde{\mathcal{R}}_{tt_b}[\![\tilde{b}_2]\!]t_s. \tag{4.3}$$

Since property 4.3 use only one tag t_s, we can use rule R-CALL, saying that a relational property using only one tag is a Hoare Triple, to give the following rule for the verification of relational properties:

$$\frac{\vdash \{\tilde{\mathcal{D}}_{t_b}[\![\tilde{b}_1]\!]\}c_1; c_2\{\tilde{\mathcal{D}}_{t_b}[\![b_2]\!]\}}{\vdash \{\tilde{b}_1\}c_1\langle t_1\rangle \sim c_2\langle t_2\rangle\{\tilde{b}_2\}} \text{ (SELF-COMP)}$$

Rule SELF-COMP says that a relational property is valid if the Hoare Triple corresponding to the self-composed programs is valid. We recall that rule SELF-COMP is only valid if the memory states for natural numbers and the memory states for commands are disjoint for each tag:

$$dom(\phi(t_1)) \cap dom(\phi(t_2)) = \emptyset$$
$$dom(state(\phi_c(t_1))) \cap dom(state(\phi_c(t_2))) = \emptyset.$$

By performing a straightforward combination of rule RECURSION and rule SELF-COMP, we get support of procedure call in Self-Composition.

$$\frac{\begin{array}{c}\forall t \in dom(\phi_a).\forall y \in dom(\phi_a(t)).\phi_a(t) \vdash \{fst(\phi_a(t)(y))\}state(\phi_c(t))(y)\{snd(\phi_a(t)(y))\} \\ \phi_a(t_1) \cup \phi_a(t_2) \vdash \{\tilde{\mathcal{D}}_{t_b}[\![\tilde{b}_1]\!]\}c_1; c_2\{\tilde{\mathcal{D}}_{t_b}[\![b_2]\!]\}\end{array}}{\vdash \{\tilde{b}_1\}c_1\langle t_1\rangle \sim c_2\langle t_2\rangle\{\tilde{b}_2\}}$$

(RECURSION-SELF-COMP)

Notice that the rule requires that for both tags in ϕ_a the domain of the contracts are disjoint:

$$dom(\phi_a(t_1)) \cap dom(\phi_a(t_2)) = \emptyset,$$

which is ensured from the moment we have:

$$dom(state(\phi_c(t_1))) \cap dom(state(\phi_c(t_2))) = \emptyset.$$

Example 4.3. We consider the same example as in Section 4.1.

$$\{x\langle t_1\rangle = x\langle t_2\rangle\}x := x + 1\langle t_1\rangle \sim x := x + 2\langle t_2\rangle\{x\langle t_1\rangle + 1 = x\langle t_2\rangle\}.$$

We can prove this example using Self-Composition. However, the memory states are sharing location since both programs and relational boolean expressions $\tilde{b}_1$ and $\tilde{b}_2$ share locations. To fulfill the requirement of Self-Composition, we can simply rename the locations to get disjoint

memory states, since the set of variables used is equal to the domain of the memory state. Similarly, in case the memory state for commands and for contracts are not disjoint for all tags, we can rename the procedures. We present in the following only rule R-RENAME-L, for the renaming of locations:

$$\frac{\vdash \{\tilde{\mathcal{R}}_{v_b}[\![\tilde{b}_1]\!](t_2, x, x')\} c_1\langle t_1\rangle \sim c_2[x/x']\langle t_2\rangle \{\tilde{\mathcal{R}}_{v_b}[\![\tilde{b}_2]\!](t_2, x, x')\}}{\vdash \{\tilde{b}_1\} c_1\langle t_1\rangle \sim c_2\langle t_2\rangle \{\tilde{b}_2\}} \text{(R-RENAME-L)}$$

where function $\tilde{\mathcal{R}}_{v_b}$ (defined in Appendix A.3.1) rename location x into x' for a tag t in a relational boolean expression. We use notation $c[x/x']$ for the substitution of all occurrences of location x by location x' in a command. Notice that the rule is not complete for simplicity; the renaming of locations in the procedures and procedure contracts are absent. The symmetric version (the command c_2 is renamed) and the other way around of rule R-RENAME-L are also true. A proof that renaming is sound can be found in [BDR11].

Using the renaming we get:

$$\{x_{t_1}\langle t_1\rangle = x_{t_2}\langle t_2\rangle\} x_{t_1} := x_{t_1} + 1\langle t_1\rangle \sim x_{t_2} := x_{t_2} + 2\langle t_2\rangle \{x_{t_1}\langle t_1\rangle + 1 = x_{t_2}\langle t_2\rangle\}.$$

Now, since no locations are shared between commands and relational boolean expressions, we can replace all tags by the same tag

$$\{x_{t_1}\langle t_s\rangle = x_{t_2}\langle t_s\rangle\} x_{t_1} := x_{t_1} + 1\langle t_s\rangle \sim x_{t_2} := x_{t_2} + 2\langle t_s\rangle \{x_{t_1}\langle t_s\rangle + 1 = x_{t_2}\langle t_s\rangle\},$$

and compose sequentially both programs.

$$\{x_{t_1}\langle t_s\rangle = x_{t_2}\langle t_s\rangle\} x_{t_1} := x_{t_1} + 1; x_{t_2} := x_{t_2} + 2\langle t_s\rangle \{x_{t_1}\langle t_s\rangle + 1 = x_{t_2}\langle t_s\rangle\}.$$

Since the same tag is used, we can remove the tag to get a Hoare Triple.

$$\{x_{t_1} = x_{t_2}\} x_{t_1} := x_{t_1} + 1; x_{t_2} := x_{t_2} + 2\{x_{t_1} + 1 = x_{t_2}\}.$$

Using rules SEQUENCE,CONSEQUENCE and ASSIGN, the triple can be proven valid.

Example 4.3 shows that Self-Composition is a simple method to prove validity of relational properties. However, the fact that Self-Composition relies on methods for the proof of Hoare Triple has also its drawbacks. No relational property can be expressed between intermediate commands. For example, in case of loops, it may be necessary to have a relational invariant (example 4.4) *i.e.* an invariant linking locations from different loops. This is possible in Relational Hoare Logic, since the invariant in the rule for loops R-WHILE is a relational boolean expression $\tilde{b}$, and the body of the loops are handled simultaneously.

Example 4.4. The following example, taken from [BCK11], shows a relational property between a program (tagged $\langle t_1\rangle$) adding the sum of the first ten naturals to location x_1 and an optimized version (tagged $\langle t_2\rangle$). The optimisation consists in beginning the loop with x_3 set to 1

instead of 0. Thus, the number of iterations is reduced by one. The property consists in proving that both programs are equivalent by checking that the value of location x_1 is the same after evaluation of each program, by assuming that before the evaluation, the value of location x_2 is the same for each program.

$$\{x_2\langle t_1\rangle = x_2\langle t_2\rangle\}\ \begin{array}{l} x_1 := x_2; \\ x_3 := 0; \\ \textbf{while } x_3 < 10 \textbf{ do } \{ \\ \quad x_1 = x_1 + x_3; \\ \quad x_3 = x_3 + 1 \\ \} \end{array}\ \langle t_1\rangle \sim \begin{array}{l} x_1 := x_2; \\ x_3 := 1; \\ \textbf{while } x_3 < 10 \textbf{ do } \{ \\ \quad x_1 = x_1 + x_3; \\ \quad x_3 = x_3 + 1 \\ \} \end{array}\ \langle t_2\rangle\{x_1\langle t_1\rangle = x_1\langle t_2\rangle\}$$

Using Self-Composition, we get a new program where the loops composing the initial programs are in sequence.

$$\{x_{2\_t_1} = x_{2\_t_2}\}\ \begin{array}{l} x_{1\_t_1} := x_{2\_t_1}; \\ x_{3\_t_1} := 0; \\ \textbf{while } x_{3\_t_1} < 10 \textbf{ do } \{ \\ \quad x_{1\_t_1} = x_{1\_t_1} + x_{3\_t_1}; \\ \quad x_{3\_t_1} = x_{3\_t_1} + 1 \\ \}; \\ x_{1\_t_2} := x_{2\_t_2}; \\ x_{3\_t_2} := 1; \\ \textbf{while } x_{3\_t_2} < 10 \textbf{ do } \{ \\ \quad x_{1\_t_2} = x_{1\_t_2} + x_{3\_t_2}; \\ \quad x_{3\_t_2} = x_{3\_t_2} + 1 \\ \} \end{array}\ \{x_{1\_t_1} = x_{1\_t_2}\}$$

No property relating the values of locations involved in both loops can be defined. Only standard loop invariant can be used. In this case, the following non linear invariant is required twice to prove that the value in location $x_{1\_t_1}$ and $x_{1\_t_2}$ are equal after the loops :

$$x_{1\_t_i} = x_{2\_t_i} + \frac{x_{3\_t_i} \times (x_{3\_t_i} - 1)}{2}.$$

4.4 Product Program

Example 4.4 shows that proving relational properties using Self-Composition can be tedious. Barthe et al. proposed in [BCK16, BCK11] Product Program, an extension of Self-Composition performing more advanced code transformations. The objective is to solve the problems mentioned in example 4.4.

4.4.1 Minimal Product Program

As for the Self-Composition, Product Program requires that for both tags in ϕ the associated memory states do not share locations and for both tags in ϕ_c the associated states do not share

commands. Verification of relational properties using Product Program is based on the following rules:

$$\frac{c_1 \times c_2 \to c \qquad \vdash \{\tilde{\mathcal{D}}_{t_b}[\![\tilde{b}_1]\!]\}c\{\tilde{\mathcal{D}}_{t_b}[\![\tilde{b}_2]\!]\}}{\vdash \{\tilde{b}_1\}c_1\langle t_1\rangle \sim c_2\langle t_2\rangle\{\tilde{b}_2\}} \quad \text{(PRODUCT)}$$

In addition to Self-Composition, we have $c_1 \times c_2 \to c$ performing specific transformations depending on the related command c_1 and c_2 and using following system:

$$\frac{}{\textbf{skip} \times \textbf{skip} \to \textbf{skip}; \textbf{skip}} \quad \text{(P-SKIP)}$$

$$\frac{}{x_1 := a_1 \times x_2 := a_2 \to x_1 := a_1; x_2 := a_2} \quad \text{(P-ASSIGN)}$$

$$\frac{}{\textbf{assert}(b_1) \times \textbf{assert}(b_2) \to \textbf{assert}(b_1); \textbf{assert}(b_2)} \quad \text{(P-ASSERT)}$$

$$\frac{c_1 \times c_2 \to c \qquad c_1' \times c_2' \to c'}{c_1; c_1' \times c_2; c_2' \to c; c'} \quad \text{(P-SEQUENCE)}$$

$$\frac{c_1 \times c_3 \to c \qquad c_2 \times c_4 \to c'}{\begin{array}{c}\textbf{if } b_1 \textbf{ then } \{c_1\} \textbf{ else } \{c_2\} \times \textbf{if } b_2 \textbf{ then } \{c_3\} \textbf{ else } \{c_4\} \to \\ \textbf{assert}(b_1 \equiv b_2); \textbf{if } b_1 \textbf{ then } \{c\} \textbf{ else } \{c'\}\end{array}} \quad \text{(P-CONDITION)}$$

$$\frac{c_1 \times c_2 \to c}{\begin{array}{c}\textbf{while } b_1 \textbf{ do } \{c_1\} \times \textbf{while } b_2 \textbf{ do } \{c_2\} \to \\ \textbf{assert}(b_1 \equiv b_2); \textbf{while } b_1 \textbf{ do } \{c; \textbf{assert}(b_1 \equiv b_2)\}\end{array}} \quad \text{(P-WHILE)}$$

Notice that in case of command **skip**, assignment, assertion and sequence, Product Program is almost equivalent to Self-Composition. Only the order of the commands are changed due to the rule for sequence.

Where Product Program differs from Self-Composition is in the case of condition and loops. Instead of composing in sequence the command, and thus duplicating commands, a single command is maintained. For example, the product of two loops is one loop, with as body the product of the bodies of the related loops. However, this is true only if the loops are synchronized. Thus, the loop conditions must evaluate to the same boolean value at entry and during iteration. This explains the presence of the assertions (the case of command **if** is similar).

4.4.2 Extended Minimal Product Program

Notice that Relational Hoare Logic and Product Program are close. This is used in [BCK16] to combine both methods in order to avoid introducing assertions in the program by rule P-CONDITION and P-WHILE; equivalence between conditions are proved in the rules. Moreover, if no rule for handling dissimilar program can be applied, Self-Composition is used to get a complete proof system.

The similarities with Relational Hoare Logic also implies that additional rules must be used in Product Program to overcome the requirement of program similarity. We can present following rules taken from [BCK16]:

$$\frac{}{x_1 := a_1 \times \textbf{skip} \to x_1 := a_1} \quad \text{(P-DAssign)}$$

$$\frac{}{\textbf{assert}(b) \times \textbf{skip} \to \textbf{assert}(b)} \quad \text{(P-DAssert)}$$

$$\frac{(\textbf{skip}; c_1) \times c_2 \to c}{c_1 \times c_2 \to c} \quad \text{(P-SI)}$$

$$\frac{\textbf{assert}(b); c; \textbf{while } b \textbf{ do } \{c\} \times c' \to c''}{\textbf{while } b \textbf{ do } \{c\} \times c' \to c''} \quad \text{(P-UL)}$$

Similarities between those rules (P-DAssign, P-SI) and the additional rules for Relational Hoare Logic (R-SI,R-DA) can be identified, since they attempt to solve the same problem.

Example 4.5. We reconsider example 4.4 from the previous section, already with renamed locations:

$$\{x_{2\_t_1}\langle t_1\rangle = x_{2\_t_2}\langle t_2\rangle\}$$

$$\begin{array}{l} x_{1\_t_1} := x_{2\_t_1}; \\ x_{3\_t_1} := 0; \\ \textbf{while } x_{3\_t_1} < 10 \textbf{ do } \{ \\ \quad x_{1\_t_1} = x_{1\_t_1} + x_{3\_t_1}; \\ \quad x_{3\_t_1} = x_{3\_t_1} + 1 \\ \} \end{array} \langle t_1\rangle \sim \begin{array}{l} x_{1\_t_2} := x_{2\_t_2}; \\ x_{3\_t_2} := 1; \\ \textbf{while } x_{3\_t_2} < 10 \textbf{ do } \{ \\ \quad x_{1\_t_2} = x_{1\_t_2} + x_{3\_t_2}; \\ \quad x_{3\_t_2} = x_{3\_t_2} + 1 \\ \} \end{array} \langle t_2\rangle$$

$$\{x_{1\_t_1}\langle t_1\rangle = x_{1\_t_2}\langle t_2\rangle\}$$

We notice that the loops are not synchronized, the program on the left performs one additional iteration. Thus, we apply rule P-UL to get synchronized loops. As the loop unfolding results in two programs with dissimilar shapes, we use P-SI to get two programs with similar shapes. We can then apply rule P-DAssert and P-DAssign for the dissimilar part and rule P-Assign and P-While for the similar part to get the product program:

$$\begin{array}{l} x_{1\_t_1} := x_{2\_t_1}; \\ x_{3\_t_1} := 0; \\ \textbf{assert}(x_{3\_t_1} < 10); \\ x_{1\_t_1} = x_{1\_t_1} + x_{3\_t_1}; \\ x_{3\_t_1} = x_{3\_t_1} + 1 \\ \textbf{while } x_{3\_t_1} < 10 \textbf{ do } \{ \\ \quad x_{1\_t_1} = x_{1\_t_1} + x_{3\_t_1}; \\ \quad x_{3\_t_1} = x_{3\_t_1} + 1 \\ \} \end{array} \times \begin{array}{l} x_{1\_t_2} := x_{2\_t_2}; \\ x_{3\_t_2} := 1; \\ \textbf{skip}; \\ \textbf{skip}; \\ \textbf{skip}; \\ \textbf{while } x_{3\_t_2} < 10 \textbf{ do } \{ \\ \quad x_{1\_t_2} = x_{1\_t_2} + x_{3\_t_2}; \\ \quad x_{3\_t_2} = x_{3\_t_2} + 1 \\ \} \end{array}$$

The resulting triple is as follows:

$$
\begin{array}{lll}
 & x_{1\_t_1} := x_{2\_t_1}; & \\
 & x_{1\_t_2} := x_{2\_t_2}; & \\
 & x_{3\_t_1} := 0; & \\
 & x_{3\_t_2} := 1; & \\
 & \textbf{assert}(x_{3\_t_1} < 10); & \\
 & x_{1\_t_1} = x_{1\_t_1} + x_{3\_t_1}; & \\
 & x_{3\_t_1} = x_{3\_t_1} + 1; & \\
\{x_{2\_t_1} = x_{2\_t_2}\} & \textbf{assert}(x_{3\_t_1} < 10 \equiv x_{3\_t_2} < 10); & \{x_{1\_t_1} = x_{1\_t_2}\} \\
 & \textbf{while } x_{3\_t_1} < 10 \textbf{ do } \{ & \\
 & \quad x_{1\_t_1} = x_{1\_t_1} + x_{3\_t_1}; & \\
 & \quad x_{1\_t_2} = x_{1\_t_2} + x_{3\_t_2}; & \\
 & \quad x_{3\_t_1} = x_{3\_t_1} + 1; & \\
 & \quad x_{3\_t_2} = x_{3\_t_2} + 1; & \\
 & \quad \textbf{assert}(x_{3\_t_1} < 10 \equiv x_{3\_t_2} < 10) & \\
 & \} &
\end{array}
$$

Using Hoare Logic, we can prove this triple. Notice that the product program only requires one loop invariant $x_{1\_t_1} = x_{1\_t_2}$ (called coupling invariant) for the loop rule WHILE, in opposition to example 4.4, where we needed two invariants. Moreover, the coupling invariant is much easier to use and prove. In general, if two loops are of similar shape, it is easier to define a coupling invariant, in order to connect the desired parts, than to define two invariants resuming the result of the loops and finally connect desired parts using those invariants.

4.4.3 Product Program and Procedures

Support of procedure call in product program is proposed in [EMH18] through the following rule (here in a simpler form than the transformation proposed in [EMH18]):

$$\frac{state(\phi_c(t_1))(y_1) \times state(\phi_c(t_2))(y_2) \to c}{\textbf{call}(y_1) \times \textbf{call}(y_2) \to \textbf{call}(y)} \text{ with } \psi(y) = c \qquad \text{(P-CALL)}$$

We assume that the **call** on the left of $\times$ is originally tagged with t_1 and the **call** on the right of $\times$ is originally tagged with t_2. The product consists in merging two procedure calls to y_1 (in context of t_1) and y_2 (in context of t_2) into a single call to y. The program bound to y is the product of the programs bound to y_1 (in context of t_1) and y_2 (in context of t_2).

Example 4.6. We consider the following relational property:

$$\{x_1\langle t_1\rangle = x_2\langle t_2\rangle\}\textbf{call}(y_1)\langle t_1\rangle \sim \textbf{call}(y_2)\langle t_2\rangle\{x_1\langle t_1\rangle = x_2\langle t_2\rangle\},$$

a relational execution enviroment ϕ_c, where the memory states for command are defined, for each tag, by:

$$state(\phi_c(t_1)) = \{y_1 \to x_1 := x_1 + 1\},$$

$$state(\phi_c(t_2)) = \{y_2 \to x_2 := x_2 + 1\}.$$

We can apply rule P-CALL and PRODUCT to get the following Hoare Triple:

$$\{x_1 = x_2\}\textbf{call}(y)\{x_1 = x_2\},$$

where state ψ is defined by:

$$\{y \to x_1 := x_1 + 1; x_2 := x_2 + 1\}$$

In this chapter we have presented classical verification methods for relational properties in the context of the R-WHILE language. We have seen that those methods have a limited support for procedure call and cannot use relational properties as hypotheses. We will see in Chapter 6 how we can extend Relational Hoare Logic and Self-Composition to get a better support of procedure calls in order to use relational properties, like procedure contracts in axiomatic semantics.

The previous presentation on product program was made for the sake of completeness on the basic techniques for verifying relational properties. We will not go back over this verification technique.

Chapter 5

Extension

The semantics introduced in Section 3.2, for the R-WHILE language, defines exactly what a program does. The Hoare logic, presented in Section 3.3 provides a way to state properties on programs, and to prove them. However, there are properties that we want to specify on programs that cannot be expressed within this frame.

Example 5.1. We consider a program c defined by:

$$\begin{array}{l} x_3 := x_1; \\ x_1 := x_2; \\ x_2 := x_3 \end{array}$$

It is not possible to write a property, using $\mathbb{E}_b$, that specifies that the values stored at locations x_1 and x_2 after evaluation of the c program are the values stored at locations x_2 and x_1 respectively before evaluation.

More generally, it is not possible to write properties that refer to the values associated to locations in distinct memory states. A solution to this problem is the use of labels and a specific construct, that we call at (as in Section 2.1). Labels are mapped to memory state, and the *at* construct allows referring to the value of a location in a memory state linked to a label, so that Example 5.1 can be written as follows:

$$\{true\} \begin{array}{ll} l_1 & : x_3 := x_1; \\ l_2 & : x_1 := x_2; \\ l_3 & : x_2 := x_3; \\ l_4 & : \textbf{skip}; \end{array} \{at(x_1, l_4) = at(x_2, l_1) \wedge at(x_2, l_4) = at(x_1, l_1)\}$$

Note that we use here an additional label l_4, to refer to the state after the evaluation of the command at the label l_3. We show in Section 5.2 how this problem is solved.

Another limitation of the present model is the impossibility to refer to axiomatized predicates inside boolean expressions. As shown in Example 1.1 for the verification of factorial function, using predicates in deductive verification is standard. Moreover, axiomatized predicates are the basis of the solutions proposed in Chapter 6 and Chapter 8 for handling relational properties.

The chapter is therefore organized as follows. Section 5.1 presents the extensions added to the R-WHILE language, of Section 3.2 to solve the previously mentioned limitations. Section 5.2 refines the definition of Hoare Triples by taking into account the extensions. Finally, Section 5.3 presents verification condition generation [Gor88] for the verification of Hoare Triples in the context of the extended R-WHILE language.

5.1 Extended R-WHILE Language

In the following, sections 5.1.1, 5.1.2 and 5.1.3 present the extended arithmetic expressions, boolean expressions and commands of R-WHILE. Section 5.1.4 defines some hypotheses assumed to be valid in the following chapters. In order to distinguish the syntax and functions defined in this chapter from those in the previous chapters, we add the symbol $\hat{}$ on each new syntax and function. The extended R-WHILE language is called R-WHILE*.

5.1.1 Extension of Arithmetic Expressions

For our first extension, which focuses on $\mathbb{E}_a$, we require that commands are named using identifiers. We therefore introduce a new set $\mathbb{L}$ of identifiers, which are called *label*. We use variables $l, l_0, l_1, ...$ to denote labels. The syntax for naming commands using labels is presented in Section 5.1.3.

We now define the grammar rule for extended arithmetic expression $\hat{\mathbb{E}}_a$;

$$\begin{aligned} \alpha \ &::= n \\ &\mid at(x, l) \\ &\mid \alpha_1 \ opa \ \alpha_2 \end{aligned}$$

We use metavariables $\alpha, \alpha_0, \alpha_1, ...$ to range over the set $\hat{\mathbb{E}}_a$. Extended arithmetic expressions $\hat{\mathbb{E}}_a$ are similar to arithmetic expressions $\mathbb{E}_a$, but use the new construct $at(x, l)$ instead of x. $at(x, l)$ denotes the value of memory location x at the memory state bound to label l. We thus define Λ, the environment that maps labels to memory states

$$\Lambda = \mathbb{L} \rightharpoonup \Sigma,$$

and use metavariables $\lambda, \lambda_0, \lambda_1, ...$ to range over the set Λ.

Using environment Λ, we can give the following evaluation function for $\hat{\mathbb{E}}_a$.

Definition 5.1. Evaluation function $\hat{\xi}_a : \hat{\mathbb{E}}_a \to (\Lambda \to \mathbb{N}_\perp)$ for arithmetic expressions $\hat{\mathbb{E}}_a$, is defined by structural induction on extended arithmetic expressions:

$$\hat{\xi}_a[\![n]\!]\lambda = n$$
$$\hat{\xi}_a[\![at(x,l)]\!]\lambda = (\lambda\, l)\; x$$
$$\hat{\xi}_a[\![\alpha_0\ opa\ \alpha_1]\!]\lambda = \hat{\xi}_a[\![\alpha_0]\!]\lambda\ opa_\perp\ \hat{\xi}_a[\![\alpha_1]\!]\lambda.$$

5.1.2 Extension of Boolean Expressions

For our second extension, which focuses on $\mathbb{E}_b$, we want to refer to predicates inside boolean expressions. More precisely, we want predicates parametrized by extended arithmetic expressions. Therefore, we first define $\mathbb{P}$, the set of predicate identifiers, composed of the set of identifiers $\mathbb{P}_n$ for predicates taking n extended arithmetic expressions.

$$\mathbb{P} = \bigcup_{n\in\mathbb{N}} \mathbb{P}_n.$$

To avoid typing rules, we assume that all sets $\mathbb{P}_n$ are disjoint to get well typed predicate by definition:

$$\forall i_1, i_2 \in \mathbb{N}.(i_1 \neq i_2) \Rightarrow \mathbb{P}_{i_1} \cap \mathbb{P}_{i_2} = \emptyset.$$

We use metavariables $p, p_0, p_1, ...$ to range over the set $\mathbb{P}$ and $p^n, p_0^n, p_1^n, ...$ to range over the set $\mathbb{P}_n$.

We define the grammar rules for extended boolean expressions $\hat{\mathbb{E}}_b$

$$\begin{aligned}\beta\ ::=\ & true \mid false \\ & \mid \alpha_1\ opb\ \alpha_2 \\ & \mid \beta_1\ opl\ \beta_2 \mid \neg\beta' \\ & \mid \beta_0 \Rightarrow \beta_1 \\ & \mid p^n(\alpha_1, ..., \alpha_n)\end{aligned}$$

and use metavariables $\beta, \beta_0, \beta_1, ...$ to range over the set $\hat{\mathbb{E}}_b$. We add implication to $\hat{\mathbb{E}}_b$ for convenience.

For the evaluation of extended boolean expressions we note that, since $\hat{\mathbb{E}}_b$ contains predicates without explicit definition, we cannot use the same type of evaluation function as for $\mathbb{E}_b$. To decide whether an extended boolean expression is valid, we have to use a set of axioms related to the predicates. Therefore, we use the function smt, defined in Section 3.1.3, to decide if, for a set of axioms, a translation into first-order logic of an extended boolean expression is valid. The translation of extended boolean expressions $\hat{\mathbb{E}}_b$ is performed by function $\hat{\xi}_b$ defined as follows.

Definition 5.2. Evaluation function $\hat{\xi}_b : \hat{\mathbb{E}}_b \to (\Lambda \to \mathbb{Q}_\perp)$ for boolean expression $\hat{\mathbb{E}}_b$, is defined by structural induction on extended boolean expressions:

$$\hat{\xi}_b[\![true]\!]\lambda = [\![T]\!]$$

$$\hat{\xi}_b[\![false]\!]\lambda = [\![F]\!]$$

$$\hat{\xi}_b[\![\alpha_0\ opb\ \alpha_1]\!]\lambda = \begin{cases} \perp & \text{if } \hat{\xi}_a[\![\alpha_0]\!]\lambda = \perp \text{ or } \hat{\xi}_a[\![\alpha_1]\!]\lambda = \perp \\ [\![\hat{\xi}_a[\![\alpha_0]\!]\lambda\ opb\ \hat{\xi}_a[\![\alpha_1]\!]\lambda]\!] \end{cases}$$

$$\hat{\xi}_b[\![\beta_0\ opl\ \beta_1]\!]\lambda = \begin{cases} \perp & \text{if } \hat{\xi}_b[\![\beta_0]\!]\lambda = \perp \text{ or } \hat{\xi}_b[\![\beta_1]\!]\lambda = \perp \\ [\![\hat{\xi}_b[\![\beta_0]\!]\lambda\ opl\ \hat{\xi}_b[\![\beta_1]\!]\lambda]\!] \end{cases}$$

$$\hat{\xi}_b[\![\neg\beta]\!]\lambda = \begin{cases} \perp & \text{if } \hat{\xi}_b[\![\beta]\!]\lambda = \perp \\ [\![\neg\hat{\xi}_b[\![\beta]\!]\lambda]\!] \end{cases}$$

$$\hat{\xi}_b[\![\beta_0 \Rightarrow \beta_1]\!]\lambda = \begin{cases} \perp & \text{if } \hat{\xi}_b[\![\beta_0]\!]\lambda = \perp \text{ or } \hat{\xi}_b[\![\beta_1]\!]\lambda = \perp \\ [\![\hat{\xi}_b[\![\beta_0]\!]\lambda \Rightarrow \hat{\xi}_b[\![\beta_1]\!]\lambda]\!] \end{cases}$$

$$\hat{\xi}_b[\![p^n(\alpha_1, ..., \alpha_n)]\!]\lambda = \begin{cases} \perp & \text{if } \hat{\xi}_a[\![\alpha_1]\!]\lambda = \perp \text{ or ... or } \hat{\xi}_a[\![\alpha_n]\!]\lambda = \perp \\ [\![\mathrm{p}(\hat{\xi}_a[\![\alpha_1]\!]\lambda, ..., \hat{\xi}_a[\![\alpha_n]\!]\lambda)]\!]. \end{cases}$$

Notice that the translation of natural numbers, returned by function $\hat{\xi}_a$, into constant terms of $\mathbb{E}_q$, of type nat, is implicit. In other words, we suppose that function smt supports integer arithmetic (which is the case in practice for most of the provers used in program verification).

Notice also that arithmetic operators are absent from the generated formulas; function $\hat{\xi}_a$ evaluates extended arithmetic expressions to naturals. We could have defined function $\hat{\xi}_a$ such that extended arithmetic expressions are translated into terms of $\mathbb{E}_q$, resulting in formulas containing arithmetic operators. However, we have chosen otherwise, for convenience reason.

Boolean and logical operators are also translated implicitly into the MFOL language. Predicate identifier p^n are translated implicitly into an associated predicate identifier p of MFOL.

Example 5.2. If we consider the following extended boolean expression

$$p(at(x_1, l_1), at(x_2, l_2)) \wedge at(x_1, l_1) + at(x_2, l_1) = 3,$$

and an environment $\lambda = \{l_1 \to \{x_1 \to 1, x_2 \to 2\}, l_2 \to \{x_1 \to 1, x_2 \to 1\}\}$, we get the following well typed formula by applying function $\hat{\xi}_b$

$$\mathrm{p}(1, 1) \wedge 3 = 3.$$

Now, if we consider the following set of closed formulas

$$u_{\mathbb{Q}} = \{\forall v_1, v_2 : \text{nat}. v_1 = v_2 \Rightarrow \mathrm{p}(v_1, v_2)\},$$

the oracle function smt will return V for the following query:

$$\mathrm{smt}(u_{\mathbb{Q}}, \mathrm{p}(1, 1) \wedge 3 = 3).$$

5.1.3 Extension of Commands

As stated earlier, the $at(x, l)$ construct requires that some commands are named with labels. We therefore define $\hat{\mathbb{C}}$, an improved $\mathbb{C}$.

$$\begin{aligned}\varsigma ::= {} & l : \textbf{skip} \\ | \; & l : x := a \\ | \; & \varsigma_1; \varsigma_2 \\ | \; & l : \textbf{assert}(\beta) \\ | \; & l : \textbf{if } b \textbf{ then } \{\varsigma_1\} \textbf{ else } \{\varsigma_2\} \\ | \; & l : \textbf{while } b \textbf{ do } \{\varsigma\} \\ | \; & l : \textbf{call}(y)\end{aligned}$$

Command **assert**(β) takes now an extended boolean expression $\hat{\mathbb{E}}_b$ as parameter. The condition of commands **if** and **while** remains a simple boolean expression of $\mathbb{E}_b$. In other words, the evaluation of a program still uses only the current state of the memory. Only assertions can refer to other states through their corresponding label. The assignment expression remains an arithmetic expression of $\mathbb{E}_a$. All commands, except sequential composition, are named using labels. Since all atomic commands are named, it is not required to name the sequential composition. We use metavariables $\varsigma, \varsigma_0, \varsigma_1, ...$ to range over the set $\hat{\mathbb{C}}$.

Since we have extended commands, we have to refine the definition of Ψ into $\hat{\Psi}$ as follows:

$$\hat{\Psi} = \mathbb{Y} \rightharpoonup \hat{\mathbb{C}}.$$

A consequence of the fact that a command **assert**(β) takes now an extended boolean expression as a parameter is that it can refer to any label that occurs in a program, through the construct $at(x, l)$, including labels of commands that have not been executed yet, or that will even not be executed at all during the evaluation of the program. Thus, in order to ensure that we will always be able to evaluate our extended boolean expressions, we limit the use of construct $at(x, l)$ with the following three rules (A formal definition of those rules are given in Section 5.1.4):

Rule 1. *A label l must occur before the occurrence of $at(x, l)$ in the program.*

Example 5.3. The following program is not well defined since we refer, inside an assertion, to label l_3 that occurs after the assertion:

$$l_1 : x_2 = 3; l_2 : \textbf{assert}(at(x_1, l_3) = at(x_1, l_1)); l_3 : x_1 = x_2 + 2.$$

Rule 2. *A label l must not be inside an inner block, that is, we cannot refer to the value of a location in a state that occurs in a loop, a call or conditions.*

Example 5.4. The following program is not well defined, since we refer, inside an assertion which is outside of the loop, to label l_2 that occur inside the loop body:

$$l_1 : \textbf{while } x_1 < 4 \textbf{ do } \{l_2 : x_1 = x_1 + 1\}; l_3 : \textbf{assert}(at(x_1, l_2) < 5).$$

Rule 3. *A label l must not be outside a call, that is, we cannot refer to the value of a location in a state that occurs outside a call.*

Example 5.5. For the following command, the memory $\psi = \{y \mapsto l_3 : \textbf{assert}(at(x_1, l_1) < 5)\}$ is not well defined since we refer, inside an assertion, to label l_1 that doesn't occur inside the program at location y:

$$l_1 : x_1 := 10; l_2 : \textbf{call}(y).$$

We can now give the signature of function $\hat{\xi}_c$ for the evaluation of commands $\hat{\mathbb{C}}$:

$$\hat{\xi}_c : \hat{\mathbb{C}} \to ((\mathbb{N} \times \hat{\Psi} \times \mathcal{P}(\mathbb{Q}) \times (\Sigma_\Omega \times \Lambda)) \to (\Sigma_\Omega \times \Lambda)).$$

The possibility of referring to specific states through construction $at(x, l)$ requires storing states in an environment Λ during evaluation. Thus, the evaluation function for commands takes a pair in $\Sigma \times \Lambda$ as a parameter, rather than a single memory state σ in Σ.

Since in the following definition of function $\hat{\xi}_c$, the memory state for command $\hat{\psi}$ and the set of axioms $u_\mathbb{Q}$ are never modified, we can consider them implicitly and give the following shorthand:

$$\hat{\xi}_c[\![\varsigma]\!](n, (\sigma, \lambda)) = \hat{\xi}_c[\![\varsigma]\!](n, \hat{\psi}, u_\mathbb{Q}, (\sigma, \lambda)).$$

We can now give the definition of function $\hat{\xi}_c$.

Definition 5.3. Evaluation function for extended commands

$$\hat{\xi}_c : \hat{\mathbb{C}} \to ((\mathbb{N} \times \hat{\Psi} \times \mathcal{P}(\mathbb{Q}) \times (\Sigma_\Omega \times \Lambda)) \to (\Sigma_\Omega \times \Lambda)),$$

is defined, if we run out of fuel or the memory state for natural numbers is in Ω, by

$$\begin{aligned} \hat{\xi}_c[\![\varsigma]\!](n,(\sigma,\lambda)) &= (\Omega_t,\lambda) && \text{if } n = 0 \\ \hat{\xi}_c[\![\varsigma]\!](n,(\sigma,\lambda)) &= (\sigma,\lambda) && \text{if } \sigma \in \Omega, \end{aligned}$$

otherwise, by structural induction on commands

$$\hat{\xi}_c[\![l : \textbf{skip}]\!](n,(\sigma,\lambda)) = (\sigma, \lambda[l \leftarrow \sigma])$$

$$\hat{\xi}_c[\![l : x := a]\!](n,(\sigma,\lambda)) = \begin{cases} (\Omega_\perp, \emptyset) & \text{if } \xi_a[\![a]\!]\sigma = \perp \\ (\Omega_\perp, \emptyset) & \text{if } \sigma(x) = \perp \\ (\sigma[x \leftarrow \xi_a[\![a]\!]\sigma], \lambda[l \leftarrow \sigma]) & \text{otherwise} \end{cases}$$

$$\hat{\xi}_c[\![\varsigma_0;\varsigma_1]\!](n,(\sigma,\lambda)) = \hat{\xi}_c[\![\varsigma_1]\!](n-1, \hat{\xi}_c[\![\varsigma_0]\!](n-1,(\sigma,\lambda)))$$

$$\hat{\xi}_c[\![l : \textbf{assert}(\beta)]\!](n,(\sigma,\lambda)) = \begin{cases} (\Omega_\perp, \emptyset) & \text{if } \hat{\xi}_b[\![\beta]\!](\lambda[l \leftarrow \sigma]) = \perp \\ (\sigma, \lambda[l \leftarrow \sigma]) & \text{if } \mathrm{smt}(u_{\mathbb{Q}}, \hat{\xi}_b[\![\beta]\!](\lambda[l \leftarrow \sigma])) = V \\ (\Omega_a, \emptyset) & \text{otherwise} \end{cases}$$

$$\hat{\xi}_c[\![l : \textbf{if } b \textbf{ then } \{\varsigma_0\} \textbf{ else } \{\varsigma_1\}]\!](n,(\sigma,\lambda)) =$$

$$\begin{cases} (\Omega_\perp, \emptyset) & \text{if } \xi_b[\![b]\!]\sigma = \perp \\ (fst(\hat{\xi}_c[\![\varsigma_0]\!](n-1,(\sigma,\lambda[l \leftarrow \sigma]))), \lambda[l \leftarrow \sigma]) & \text{if } \xi_b[\![b]\!]\sigma = true \\ (fst(\hat{\xi}_c[\![\varsigma_1]\!](n-1,(\sigma,\lambda[l \leftarrow \sigma]))), \lambda[l \leftarrow \sigma]) & \text{if } \xi_b[\![b]\!]\sigma = false \end{cases}$$

$$\hat{\xi}_c[\![l : \textbf{call}(y)]\!](n,(\sigma,\lambda)) = \begin{cases} (\Omega_\perp, \emptyset) & \text{if } \hat{\psi}(y) = \perp \\ (fst(\hat{\xi}_c[\![\hat{\psi}(y)]\!](n-1,(\sigma,\emptyset))), \lambda[l \leftarrow \sigma]) & \text{otherwise} \end{cases}$$

$$\hat{\xi}_c[\![l : \textbf{while } b \textbf{ do } \{\varsigma\}]\!](n,(\sigma,\lambda)) =$$

$$\begin{cases} (\Omega_\perp, \emptyset) & \text{if } \xi_b[\![b]\!]\sigma = \perp \\ (fst(\hat{\xi}_c[\![\varsigma; l : \textbf{while } b \textbf{ do } \{\varsigma\}]\!](n-1,(\sigma,\lambda[l \leftarrow \sigma]))), \lambda[l \leftarrow \sigma]) & \text{if } \xi_b[\![b]\!]\sigma = true \\ (\sigma, \lambda[l \leftarrow \sigma]) & \text{if } \xi_b[\![b]\!]\sigma = false \end{cases}$$

In the case of named commands, the environment λ is updated gradually with bindings $l \leftarrow \sigma$ (label of the evaluated command to the memory state before the evaluation). Thus, if the evaluated program satisfies rule 1, we are sure to find in λ the labels we require to evaluate assertions.

In the case of the assertion command, we use function smt to ensure that the associated extended boolean expression holds. If functon smt returns V, we know that the assertion holds. If the function smt returns U, we do not know if the formula holds, thus we return Ω_a. In the case function smt is not complete, *i.e.* some valid formulas cannot no be proven valid, program with valid assertions can return Ω_a. Notice that we could add the proven boolean expression β to the set of formulas $u_{\mathbb{Q}}$ as a hypothesis. However, strictly speaking, this is not needed, since β is a logical consequence of the axioms. Since adding a mechanism to update the set of axioms that we use would be pretty heavy, we chose to avoid it. A specific command for the update will be defined in Section 6.3, where we really require it.

For commands **if** b **then** $\{\varsigma_0\}$ **else** $\{\varsigma_1\}$, **call**(y) and **while** β **do** $\{\varsigma\}$, the returned environment λ is only updated with the label of the current command (as for all other named commands). Function *fst* is used to extract the memory state from the returned pair from the recursive call to evaluation function $\hat{\xi}_c$. Thus, if the evaluated program satisfies rule 2, we are sure to find in λ the labels we require to evaluate assertions.

For the case of procedure call, the recursive call to evaluation function $\hat{\xi}_c$ is performed with empty environment λ. Thus, if the callable procedures satisfy rule 3, we are sure to find in λ the labels we require to evaluate assertions.

As for evaluation function ξ_c, we use a natural number n to ensure termination.

Finally, in case a state of Ω is returned, we associate to it an empty environment λ, since the evaluation will stop anyways.

5.1.4 Well Defined Program

We have presented in the previous Section 5.1.3 the evaluation function for command $\hat{\xi}_c$. This function requires that program satisfies rules 1,2 and 3. Thus, we give in this section a formal definition of those rules through a function checking that commands of $\hat{\mathbb{C}}$ are well-defined.

First, we lift the powerset of labels $\mathcal{P}(\mathbb{L})$with $\bot$:

$$\mathcal{P}(\mathbb{L})_\bot = \mathcal{P}(\mathbb{L}) \cup \{\bot\}.$$

Then, we define function $lift_u$ taking two sets of labels as parameter ($u_{\mathbb{L}_\bot 1}$ and $u_{\mathbb{L}_\bot 2}$), and returning $\bot$ if there exists a label that occurs in both sets of labels. The function also propagates $\bot$.

Definition 5.4. Definition of $lift_u : \mathcal{P}(\mathbb{L})_\bot \times \mathcal{P}(\mathbb{L})_\bot \to \mathcal{P}(\mathbb{L})_\bot$ for merging two sets of labels:

$$lift_u(u_{\mathbb{L}_\bot 1}, u_{\mathbb{L}_\bot 2}) = \begin{cases} \bot & \text{if } \bot = u_{\mathbb{L}_\bot 1} \text{ or } \bot = u_{\mathbb{L}_\bot 2} \\ \bot & \text{if } u_{\mathbb{L}_\bot 1} \cap u_{\mathbb{L}_\bot 2} \neq \emptyset \\ u_{\mathbb{L}_\bot 1} \cup u_{\mathbb{L}_\bot 2} & \text{otherwise} \end{cases}$$

Using function $lift_u$, we define function $\hat{\mathcal{W}}$ to guarantee that a command ς obeys rule 1 and rule 2.

Definition 5.5. Function $\hat{\mathcal{W}} : \hat{\mathbb{C}} \to (\mathcal{P}(\mathbb{L})_\perp \to \mathcal{P}(\mathbb{L})_\perp)$, checking that rule 1 and rule 2 are satisfied by a command $\hat{\mathbb{C}}$, is defined by structural induction on commands:

$$\hat{\mathcal{W}}[\![l : \textbf{skip}]\!]u_{\mathbb{L}_\perp} = \mathit{lift}_u(u_{\mathbb{L}_\perp}, \{l\})$$

$$\hat{\mathcal{W}}[\![l : x := a]\!]u_{\mathbb{L}_\perp} = \mathit{lift}_u(u_{\mathbb{L}_\perp}, \{l\})$$

$$\hat{\mathcal{W}}[\![\varsigma_0; \varsigma_1]\!]u_{\mathbb{L}_\perp} = \hat{\mathcal{W}}[\![\varsigma_1]\!](\hat{\mathcal{W}}[\![\varsigma_0]\!]u_{\mathbb{L}_\perp})$$

$$\hat{\mathcal{W}}[\![l : \textbf{assert}(\beta)]\!]u_{\mathbb{L}_\perp} = \begin{cases} \perp & \text{if } \hat{\mathcal{C}}_{l_b}[\![\beta]\!] \not\subseteq u_{\mathbb{L}_\perp} \cup \{l\} \\ \mathit{lift}_u(u_{\mathbb{L}_\perp}, \{l\}) & \text{otherwise} \end{cases}$$

$$\hat{\mathcal{W}}[\![l : \textbf{if } b \textbf{ then } \{\varsigma_0\} \textbf{ else } \{\varsigma_1\}]\!]u_{\mathbb{L}_\perp} = \begin{cases} \perp & \text{if } \hat{\mathcal{W}}[\![\varsigma_0]\!](\mathit{lift}_u(u_{\mathbb{L}_\perp}, \{l\})) = \perp \\ \perp & \text{if } \hat{\mathcal{W}}[\![\varsigma_1]\!](\mathit{lift}_u(u_{\mathbb{L}_\perp}, \{l\})) = \perp \\ \mathit{lift}_u(u_{\mathbb{L}_\perp}, \{l\}) & \text{otherwise} \end{cases}$$

$$\hat{\mathcal{W}}[\![l : \textbf{call}(y)]\!]u_{\mathbb{L}_\perp} = \mathit{lift}_u(u_{\mathbb{L}_\perp}, \{l\})$$

$$\hat{\mathcal{W}}[\![l : \textbf{while } b \textbf{ do } \{\varsigma\}]\!]u_{\mathbb{L}_\perp} = \begin{cases} \perp & \text{if } \hat{\mathcal{W}}[\![\varsigma]\!](\mathit{lift}_u(u_{\mathbb{L}_\perp}, \{l\})) = \perp \\ \mathit{lift}_u(u_{\mathbb{L}_\perp}, \{l\}) & \text{otherwise} \end{cases}$$

Parameter $u_{\mathbb{L}_\perp}$ corresponds to all labels we can refer to (in addition to the label of the current command) in a boolean expression $\hat{\mathbb{E}}_b$, through a construct $at(x, l)$. Thus, to ensure rule 1 we check that, in case of an assertion, the set of labels used inside a boolean expression $\hat{\mathbb{E}}_b$ is a subset of $u_{\mathbb{L}_\perp} \cup \{l\}$ using function $\hat{\mathcal{C}}_{l_b}$ (defined in Appendix A.1.3) returning the set of labels used in a boolean expression $\hat{\mathbb{E}}_b$. Rule 2 is ensured by the fact that the set of labels $u_{\mathbb{L}_\perp}$ is only lifted with the label of the current command, in case of commands **if**,**while**, and **call**.

Notice that duplicate labels in the set of reachable labels are not allowed. Otherwise some labels would be hidden by others when evaluating a command. The absence of duplicated labels is guaranteed by using function lift_u when merging elements of $u_{\mathbb{L}_\perp}$.

We now refine the statement for well defined program $\mathbb{C}$, defined in Section 3.3, for $\hat{\mathbb{C}}$:

- The domain of the memory state σ is equal to the set of locations used in command ς, and commands $\psi(y)$ for all defined program names y:

$$\hat{\mathcal{C}}_{v_c}[\![\varsigma]\!] \cup \bigcup_{y \in dom(\hat{\psi})} \hat{\mathcal{C}}_{v_c}[\![\hat{\psi}(y)]\!] = dom(\sigma), \qquad (\hat{\mathcal{W}}_v(\varsigma, \hat{\psi}, \sigma))$$

  where function $\hat{\mathcal{C}}_{v_c}$ (defined in Appendix A.1.1) returns the set of variables used in a command $\hat{\mathbb{C}}$.

- The set of program names used in command ς is a subset of the domain of $\hat{\psi}$. Moreover, for all defined program names in $\hat{\psi}$, the associated commands uses a set of program names

that is a subset of the domain of the memory state $\hat{\psi}$.

$$\hat{\mathcal{C}}_f[\![\varsigma]\!] \cup \bigcup_{y \in dom(\hat{\psi})} \hat{\mathcal{C}}_f[\![\hat{\psi}(y)]\!] \subseteq dom(\hat{\psi}), \qquad (\hat{\mathcal{W}}_f(\varsigma, \hat{\psi}))$$

where function $\hat{\mathcal{C}}_f$ (defined in Appendix A.1.2) returns the set of program names used in a command $\hat{\mathbb{C}}$.

- For a given environment λ mapping labels to memory state, for all valid labels, the associated memory state for natural numbers has the same domain as σ:

$$\forall l \in dom(\lambda), dom(\lambda(l)) = dom(\sigma) \qquad (\hat{\mathcal{W}}_\lambda(\lambda, \sigma))$$

- For a set of reachable $u_{\mathbb{L}}$, a program ς is well defined in terms of $\hat{\mathcal{W}}$. Moreover, for all defined program names in a given $\hat{\psi}$, the associated command can only refer to labels that are defined in itself (rule 3):

$$\hat{\mathcal{W}}[\![\varsigma]\!]u_{\mathbb{L}} \neq \bot \wedge \forall y \in dom(\hat{\psi}), \hat{\mathcal{W}}[\![\hat{\psi}(y)]\!]\emptyset \neq \bot \qquad (\hat{\mathcal{W}}_l(u_{\mathbb{L}}, \varsigma, \hat{\psi}))$$

Note that function $\hat{\mathcal{W}}$ is only focused on the labels on a given path of the program, thus duplicated labels can be allowed.

Example 5.6. The following program, using a command **if**, is consider as well-defined by function $\hat{\mathcal{W}}$.

l_1 : $x := x + 10$;
l_2 : **if** $x_1 > 1$ **then** {
  $l_3 : x_2 := 2$
} **else** {
  $l_3 : x_2 := 3$
};
l_4 : **assert**$(at(x, l_2) = at(x, l_1) + 1)$

Note that label l_3 is used in the two branches of the condition.

To avoid such cases, we added an additional statement for well defined program. The set of labels used in a program ς are unique. Moreover, for all defined program names in a given memory state $\hat{\psi}$, the associated command uses a set of unique labels.

$$\hat{\mathcal{U}}[\![\varsigma]\!] \neq \bot \wedge \forall y \in dom(\hat{\psi}), \hat{\mathcal{U}}[\![\hat{\psi}(y)]\!] \neq \bot, \qquad (\hat{\mathcal{W}}_u(c, \hat{\psi}))$$

where function $\hat{\mathcal{U}}$ (defined in Appendix A.2) takes a command and returns all labels used in the command, or $\bot$ if there are duplicated labels.

We now use the previously defined properties to show that the evaluation of a program ς does not result in state $\Omega_\perp$ for an appropriate environment. The fact that an environment satisfies $\hat{\mathcal{W}}_u$, $\hat{\mathcal{W}}_\lambda$, $\hat{\mathcal{W}}_v$, $\hat{\mathcal{W}}_f$ and $\hat{\mathcal{W}}_l$ is called $\hat{\mathcal{WD}}(\varsigma, \hat{\psi}, \sigma, \lambda)$. It should be noted that requiring that $\hat{\mathcal{WD}}$ be satisfied by the environment is not the weakest condition such that the evaluation function $\hat{\xi}_c$ does not lead to state $\Omega_\perp$. Indeed, evaluation function $\hat{\xi}_c$ does not require reachable labels to be unique, as $\hat{\mathcal{W}}_l$ states (this condition simply avoids ambiguities). Moreover, unique labels, as $\hat{\mathcal{W}}_u$ states, are also not required by function $\hat{\xi}_c$ to not end in state $\Omega_\perp$. Again, $\hat{\mathcal{W}}_u$ merely ensures that there is no ambiguity when referring to a label.

Lemma 5.1. *For a given ς, $\hat{\psi}$, $u_\mathbb{Q}$ and n*

$$\forall \sigma \in \Sigma. \forall \lambda \in \Lambda.$$
$$\hat{\mathcal{WD}}(\varsigma, \hat{\psi}, \sigma, \lambda) \Rightarrow \mathit{fst}(\hat{\xi}_c[\![\varsigma]\!](n, \hat{\psi}, u_\mathbb{Q}, (\sigma, \lambda))) \neq \Omega_\perp$$

Proof. By structural induction on $\hat{\mathbb{C}}$, $\hat{\mathbb{E}}_b$, $\mathbb{E}_b$, $\hat{\mathbb{E}}_a$ and $\mathbb{E}_a$ □

5.2 Hoare Triple

In this section we refine the definition of functional correctness using boolean expressions $\hat{\mathbb{E}}_b$ and commands $\hat{\mathbb{C}}$.

The notation for functional correctness, using boolean expressions $\hat{\mathbb{E}}_b$ and commands $\hat{\mathbb{C}}$, is noted as before:

$$\{\beta_1\}\varsigma\{\beta_2\}$$

However, there is a problem if we want to write the associated definition. The memory state after the evaluation is not linked with a label, since it is the last memory state. So, we cannot refer to this state inside the expression β_2. Furthermore, for each program, the memory state before the evaluation may have a different label, since each program can have a different label for the first command. It would be convenient to have a specific label to refer to the state given as parameter to the evaluation function for extended commands and returned by the evaluation function for extended commands. Therefore, we define two specific (and reserved) labels Pre and $Post$, for the state before the evaluation and after the evaluation. Finally, as for function ξ_c in Section 3.3, we define a function $\hat{\dot{\xi}}_c$ calling function $\hat{\xi}_c$ with an arbitrary amount of fuel, in order to distinguish from total correction.

We can now give the definition of functional correctness for a given set of axioms $u_\mathbb{Q}$ and a memory state for extended commands $\hat{\psi}$:

$$\forall \sigma \in \Sigma. \forall \lambda \in \Lambda. \lambda[Pre \leftarrow \sigma] \models \beta_1 \Rightarrow \lambda'[Pre \leftarrow \sigma][Post \leftarrow \sigma'] \models \beta_2$$
$$\text{where } (\sigma', \lambda') = \hat{\dot{\xi}}_c[\![\varsigma]\!](\hat{\psi}, u_\mathbb{Q}, (\sigma, \lambda))$$

Statement $\lambda \models \beta$ indicates that environment λ satisfis a boolean expression β *i.e* $\hat{\xi}_b[\![\beta]\!]\lambda$ must be different from $\perp$ and $\mathrm{smt}(u_\mathbb{Q}, \hat{\xi}_b[\![\beta]\!]\lambda)$ evaluates to V, where $u_\mathbb{Q}$ is the set of given axioms. Moreover, $\lambda \models \beta$ holds if the evaluation has not finished:

$$\exists l \in \mathit{dom}(\lambda). \lambda(l) = \Omega_t,$$

i.e., there is a label for which the associated state is Ω_t.

In the following the environment λ is initially always empty, so we can refine the previous definition to obtain the definition of Hoare Triples $\{\beta_1\}\varsigma\{\beta_2\}$ used in the following:

Definition 5.6. We call extended Hoare Triples the statement

$$\forall \sigma \in \Sigma.[Pre \leftarrow \sigma] \models \beta_1 \Rightarrow \lambda[Pre \leftarrow \sigma][Post \leftarrow \sigma'] \models \beta_2$$
$$\text{where } (\sigma', \lambda) = \dot{\hat{\xi}}_c[\![\varsigma]\!](\hat{\psi}, u_{\mathbb{Q}}, (\sigma, \emptyset))$$

stating that, if an environment λ (only defined for label Pre) satisfies β_1, and if the execution of ς on λ terminates, the resulting environment satisfies β_2.

As for the Hoare Triple defined in Section 3.3, we assume that some hypotheses are satisfied by the environment composed of $\hat{\psi}$, σ and ς:

- The set of variables used in boolean expressions β_1 and β_2 are subsets of the domain of the memory state σ:

  $$\hat{\mathcal{C}}_{v_b}[\![\beta_1]\!] \subseteq dom(\sigma)$$
  $$\hat{\mathcal{C}}_{v_b}[\![\beta_2]\!] \subseteq dom(\sigma)$$

  where function $\hat{\mathcal{C}}_{v_b}$ (defined in Appendix A.1.1) returns the set of variables used in a extended boolean expression $\hat{\mathbb{E}}_b$.

- The set of labels used in the precondition β_1 is a subset of the singleton $\{Pre\}$. The set of labels used in the postcondition β_2 is a subset of the set composed of the labels we can refer to through the command ς and labels Pre and $Post$:

  $$\hat{\mathcal{C}}_{l_b}[\![\beta_1]\!] \subseteq \{Pre\}$$
  $$\hat{\mathcal{C}}_{l_b}[\![\beta_2]\!] \subseteq \hat{\mathcal{W}}[\![\varsigma]\!]\emptyset \cup \{Pre, Post\}$$

  where function $\hat{\mathcal{C}}_{l_b}$ (defined in Appendix A.1.3) returns the set of labels used in an extended boolean expression $\hat{\mathbb{E}}_b$ and function $\hat{\mathcal{W}}$ (defined in Section 5.1.4) returns the set of reachable labels from an initial set of reachable labels and a command.

- The command ς, the memory state $\hat{\psi}$ and σ satisfy property $\hat{\mathcal{WD}}$ defined in section 5.1.4 and labels Pre and $Post$ do no occur in command ς:

  $$\hat{\mathcal{WD}}(\varsigma, \hat{\psi}, \sigma, \emptyset) \wedge \{Pre, Post\} \notin \hat{\mathcal{C}}_{l_c}[\![\varsigma]\!]$$

These properties ensure that rule 1,2 and 3 are satified by the triple. Moreover, labels Pre and $Post$ are never used by command ς and the evaluation of boolean expressions β_1, β_2 and command ς are different from $\perp$ and $\Omega_\perp$ respectively.

5.3 Verification Conditions

In Section 3.3, we presented an extended Hoare Logic for proving functional correctness. However, the proof system is not compliant with boolean expression of $\hat{\mathbb{E}}_b$ and command of $\hat{\mathbb{C}}$ and thus cannot be used here. So, we propose in the following section *Verification Condition Generation* for proving functional correctness. The approach is well-known [Gor88, Win93] and is based on the fact that, for a *well annotated program*, we have functional correctness if all generated verification conditions (called VC's) are valid. We call a *well annotated program*, a program equipped with boolean expressions $\hat{\mathbb{E}}_b$ (assertions) that refer to specific program points and an environment $\hat{\Xi}$ mapping program names to two boolean expressions in $\hat{\mathbb{E}}_b$ (pre- and post-condition), similar to the environment Ξ of Section 3.3.

$$\hat{\Xi} = \mathbb{Y} \rightharpoonup \hat{\mathbb{E}}_b \times \hat{\mathbb{E}}_b.$$

We use metavariables $\hat{\xi}, \hat{\xi}_0, \hat{\xi}_1, ...$ to range over $\hat{\Xi}$. Similar to the assumption $\mathcal{W}_a$ (defined in Section 3.3), we assume for a given σ and $\hat{\xi}$ the following properties to be satisfied:

$$\begin{array}{l} \forall y \in dom(\hat{\xi}).\hat{\mathcal{C}}_{l_b}[\![fst(\hat{\xi}(y))]\!] \subseteq \{Pre\} \wedge \hat{\mathcal{C}}_{v_b}[\![fst(\hat{\xi}(y))]\!] \subseteq dom(\sigma), \\ \forall y \in dom(\hat{\xi}).\hat{\mathcal{C}}_{l_b}[\![snd(\hat{\xi}(y))]\!] \subseteq \{Pre, Post\} \wedge \hat{\mathcal{C}}_{v_b}[\![snd(\hat{\xi}(y))]\!] \subseteq dom(\sigma). \end{array} \quad (\hat{\mathcal{W}}_a(\hat{\xi}, \sigma))$$

That is, for each defined program name, the associated *contract* is composed of a pre- and post-condition. The precondition can only use the label Pre, and the postcondition can use labels Pre and $Post$. As we seek to define a modular verification condition generator, we want to replace any individual call to a procedure y by its contract (if the given precondition is satisfied before the call, the given postcondition is satisfied after it). Therefore the pre- and post-condition only rely on the state before and after the call (label Pre and $Post$).

The basic idea of verification condition generation is to translate boolean expressions $\hat{\mathbb{E}}_b$ and programs $\hat{\mathbb{C}}$ into first-order formulas (in our case MFOL) and then prove validity of the generated VC's (in our case using function smt).

The difficulty of proving the resulting VC's depends on the translation and the annotations. As following formalization intend to show in a very simplified way what a verification condition generator like WP would do on a restricted language like R-WHILE*, we chose a very simple translation by considering a memory state as an array. Such a *memory model* is typically used to model the heap in case of pointers. Several alternative models are possible [Bar11], like using logical variable to model the value of the locations. However, we expect that such a model will be less simple to formalize.

The following translation is inspired by the work proposed in [FS01], and the tool WP.

5.3.1 Translation of $\mathbb{E}_a$ and $\mathbb{E}_b$

As mentioned earlier, we choose to model the memory states by arrays. Thus, in the context of arithmetic and boolean expressions in $\mathbb{E}_a$ and $\mathbb{E}_b$, the translation consists in replacing location by access to an array.

Example 5.7. If we consider the following boolean expression

$$x_1 + x_2 = 10,$$

the translation will result in following formula

$$m[1] + m[2] = 10,$$

where m represents the current memory state where the boolean expression is evaluated. Notice that the choice of the natural number representing a location is arbitrary. We choose here 1 for x_1 and 2 for x_2.

We now define the two functions $\mathcal{T}_a$ and $\mathcal{T}_b$ for translating arithmetic and boolean expression into formulas of $\mathbb{Q}$.

First, we define mapping Δ from locations to naturals

$$\Delta = \mathbb{X} \rightharpoonup \mathbb{N},$$

and use metavariables $\delta, \delta_0, \delta_1, ...$ to range over the set Δ.

We assume a function $\mathcal{N}_n$ returning a fresh constant term of type nat. To avoid any complexity in the future, we write $e = \mathcal{N}_n$ when we want a fresh constant term e.

Finally, we define function *lift* returning the constant term of type nat associated to location x for a given environment δ. If no binding exists for the location in δ, the binding is created using a new constant term.

Definition 5.7. Function $\mathit{lift} : \Delta \times \mathbb{X} \to \Delta \times {\mathbb{E}_q}^{\text{nat}}$ returning the natural number associated to location x for a given environment δ:

$$\mathit{lift}(\delta, x) = \begin{cases} (\delta[x \leftarrow e], e) \text{ where } e = \mathcal{N}_n & \text{if } \delta(x) = \bot \\ (\delta, \delta(x)) & \text{otherwise} \end{cases}$$

Example 5.8. If we call function *lift* with an environment δ defined for location x_1 and a location x_1, we get as result the associated natural $\delta(x_1)$ and the environment δ unchanged:

$$\mathit{lift}(\{x_1 \to 1\}, x_1) = (\{x_1 \to 1\}, 1).$$

If we call function *lift* with an environment δ not defined for a location x_2 and a location x_2, we get as result a fresh natural number and the environment δ lifted with the binding x_2 to the fresh natural number:

$$\mathit{lift}(\{x_1 \to 1\}, x_2) = (\{x_1 \to 1, x_2 \to 2\}, 2).$$

Using function *lift*, we can define function $\mathcal{T}_a$ for translating an arithmetic expression $\mathbb{E}_a$ into a term of $\mathbb{E}_q$.

Definition 5.8. Function $\mathcal{T}_a : \mathbb{E}_a \to (\mathbb{M} \times \Delta \to \mathbb{E}_q \times \Delta)$, translating arithmetic expression $\mathbb{E}_a$ into an arithmetic expression $\mathbb{E}_q$, is defined by structural induction on arithmetic expressions:

$$\mathcal{T}_a[\![n]\!](m, \delta) = ([\![n]\!], \delta)$$
$$\mathcal{T}_a[\![x]\!](m, \delta) = ([\![m[e]]\!], \delta') \text{ where } (\delta', e) = \mathit{lift}(\delta, x)$$

$$\mathcal{T}_a[\![a_0 \ opa \ a_1]\!](m, \delta) = ([\![e_0 \ opa \ e_1]\!], \delta'')$$
$$\text{where } (e_0, \delta') = \mathcal{T}_a[\![a_0]\!](m, \delta) \text{ and } (e_1, \delta'') = \mathcal{T}_a[\![a_1]\!](m, \delta')$$

The array variable m models the current memory state where the arithmetic expression is evaluated. Parameter δ resumes the current naming choices for locations and guarantees no duplicated naming using function lift.

Using function $\mathcal{T}_a$, we can define function $\mathcal{T}_b$ for translating a boolean expression $\mathbb{E}_b$ into a formula of $\mathbb{Q}$, for a given array variable m modeling the current memory state and environment δ containing the current naming choices for locations.

Definition 5.9. Function $\mathcal{T}_b : \mathbb{E}_b \to (\mathbb{M} \times \Delta \to \mathbb{Q} \times \Delta)$, translating a boolean expression $\hat{\mathbb{E}}_b$ into a formula $\mathbb{Q}$, is defined by structural induction on boolean expressions:

$$\mathcal{T}_b[\![true]\!](m, \delta) = ([\![T]\!], \delta)$$
$$\mathcal{T}_b[\![false]\!](m, \delta) = ([\![F]\!], \delta)$$

$$\mathcal{T}_b[\![a_0 \ opb \ a_1]\!](m, \delta) = ([\![e_0 \ opb \ e_1]\!], \delta'')$$
$$\text{where } (e_0, \delta') = \mathcal{T}_a[\![a_0]\!](m, \delta) \text{ and } (e_1, \delta'') = \mathcal{T}_a[\![a_1]\!](m, \delta')$$

$$\mathcal{T}_b[\![b_0 \ opl \ b_1]\!](m, \delta) = ([\![q_0 \ opl \ q_1]\!], \delta'')$$
$$\text{where } (q_0, \delta') = \mathcal{T}_b[\![b_0]\!](m, \delta) \text{ and } (q_1, \delta'') = \mathcal{T}_b[\![b_1]\!](m, \delta')$$

$$\mathcal{T}_b[\![\neg b]\!](m, \delta) = ([\![\neg q]\!], \delta')$$
$$\text{where } (q, \delta') = \mathcal{T}_b[\![b]\!](m, \delta)$$

5.3.2 Translation of $\hat{\mathbb{E}}_a$ and $\hat{\mathbb{E}}_b$

In contrast to expressions $\mathbb{E}_a$ and $\mathbb{E}_b$, extended expressions $\hat{\mathbb{E}}_a$ and $\hat{\mathbb{E}}_b$ can refer to different memory states using the construct at. Thus, the translation consists in replacing the at by accesses to the corresponding array.

Example 5.9. If we consider the following boolean expression:

$$at(x_1, l_1) + at(x_2, l_2) = 10,$$

the translation will result in following formula

$$m_1[1] + m_2[2] = 10,$$

where m_1 models the memory state at label l_1 and m_2 models the memory state at label l_2.

We now define function $\hat{\mathcal{T}}_a$ and $\hat{\mathcal{T}}_b$ for translating extended arithmetic and boolean expression into formulas of $\mathbb{Q}$.

First, we define mapping Θ from labels to array variables

$$\Theta = \mathbb{L} \rightharpoonup \mathbb{M},$$

and we use metavariables $\theta, \theta_0, \theta_1, ...$ to range over the set Θ.

We assume a function $\mathcal{N}_{v_m}$ returning a fresh array variable. As for function $\mathcal{N}_n$, we write $m = \mathcal{N}_{v_m}$ when we want a new array variable m.

We define S, the product of environment Θ and Δ.

$$S = \Theta \times \Delta.$$

Finally, we define function $lift_s$ returning the constant term associated to location x for a given environment δ, and the array variable associated to label l for a given environment θ. If no bindings exist for the label, the binding is created using a new array.

Definition 5.10. Function $lift_s : S \times \mathbb{L} \times \mathbb{X} \to S \times \mathbb{M} \times \mathbb{E}_q{}^{\text{nat}}$, returning the constant term associated to location x for a given environment δ, and the array variable associated to label l for a given environment θ:

$$lift_s((\theta, \delta), l, x) = \begin{cases} ((\theta[l \leftarrow m], \delta'), m, e) & \text{if } \theta(l) = \bot \\ \quad \text{where} & \\ \qquad m = \mathcal{N}_{v_m} & \\ \qquad (\delta', e) = lift(\delta, x) & \\ ((\theta, \delta'), \theta(x), e) & \text{otherwise} \\ \quad \text{where} & \\ \qquad (\delta', e) = lift(\delta, x) & \end{cases}$$

Example 5.10. The behavior of function $lift_s$ is similar to the behavior of function $lift$. When the bindings already exist, the different environments are left unchanged:

$$lift_s((\{l_1 \to m_1\}, \{x_1 \to 1\}), l_1, x_1) = ((\{l_1 \to m_1\}, \{x_1 \to 1\}), m_1, 1)$$

Otherwise, they are lifted;

$$lift_s((\{l_1 \to m_1\}, \{x_1 \to 1\}), l_2, x_2) = ((\{l_1 \to m_1, l_2 \to m_2\}, \{x_1 \to 1, x_2 \to 2\}), m_2, 2)$$

Using function $lift_s$, we can define function $\hat{\mathcal{T}}_a$ for translating an extended arithmetic expression $\hat{\mathbb{E}}_a$ into a term $\mathbb{E}_q$.

Definition 5.11. Function $\hat{\mathcal{T}}_a : \hat{\mathbb{E}}_a \to (S \to \mathbb{E}_q \times S)$, translating an arithmetic expression $\hat{\mathbb{E}}_a$ into an arithmetic expression $\mathbb{E}_q$, is defined by structural induction on extended arithmetic expressions:

$$\hat{\mathcal{T}}_a[\![n]\!]s = ([\![n]\!], s)$$
$$\hat{\mathcal{T}}_a[\![at(x,l)]\!]s = ([\![m[n]]\!], s') \text{ where } (s', m, n) = lift_s(s, l, x)$$

$$\hat{\mathcal{T}}_a[\![\alpha_0 \; opa \; \alpha_1]\!]s = ([\![e_1 \; opa \; e_2]\!], s'')$$
$$\text{where } (e_1, s') = \hat{\mathcal{T}}_a[\![\alpha_0]\!]s \text{ and } (e_2, s'') = \hat{\mathcal{T}}_a[\![\alpha_1]\!]s'$$

Parameter s summarizes the current naming choices for locations and memory states and guarantees no duplicated naming using function $lift_s$.

Using function $\hat{\mathcal{T}}_a$, we can define function $\hat{\mathcal{T}}_b$ for transforming an extended boolean expression $\hat{\mathbb{E}}_b$ into a formula $\mathbb{Q}$.

Definition 5.12. Function $\hat{\mathcal{T}}_b : \hat{\mathbb{E}}_b \to (S \to \mathbb{Q} \times S)$, translating a boolean expression $\hat{\mathbb{E}}_b$ into an formula of $\mathbb{Q}$, is defined by structural induction on extended boolean expressions:

$$\hat{\mathcal{T}}_b[\![true]\!]s = ([\![T]\!], s)$$
$$\hat{\mathcal{T}}_b[\![false]\!]s = ([\![F]\!], s)$$

$$\hat{\mathcal{T}}_b[\![\alpha_0 \ opb \ \alpha_1]\!]s = ([\![e_0 \ opb \ e_1]\!], s'')$$
$$\text{where } (e_0, s') = \hat{\mathcal{T}}_a[\![\alpha_0]\!]s \text{ and } (e_1, s'') = \hat{\mathcal{T}}_a[\![\alpha_1]\!]s'$$

$$\hat{\mathcal{T}}_b[\![\beta_0 \ opl \ \beta_1]\!]s = ([\![q_0 \ opl \ q_1]\!], s'')$$
$$\text{where } (q_0, s') = \hat{\mathcal{T}}_b[\![\beta_0]\!]s \text{ and } (q_1, s'') = \hat{\mathcal{T}}_b[\![\beta_1]\!]s'$$

$$\hat{\mathcal{T}}_b[\![\neg\beta]\!]s = ([\![\neg q]\!], s') \text{ where } (q, s') = \hat{\mathcal{T}}_b[\![\beta]\!]s$$

$$\hat{\mathcal{T}}_b[\![p^n(\alpha_1, ..., \alpha_n)]\!]s = [\![(\mathrm{p}(e_1, ..., e_n)]\!], s_n)$$
$$\text{where } (e_1, s_1) = \hat{\mathcal{T}}_a[\![\alpha_1]\!]s \text{ and ... and } (e_n, s_n) = \hat{\mathcal{T}}_a[\![\alpha_n]\!]s_{n-1}$$

As for function $\hat{\xi}_b$, the translation of boolean and logical operators are implicit, and predicate identifier p^n are translated implicitly into an associated predicate identifier p.

5.3.3 Translation of $\hat{\mathbb{C}}$

In the following, we define function $\hat{\mathcal{T}}_c$ for the translation of a command ς into a formula q. Function $\hat{\mathcal{T}}_c$ can be seen as computing the strongest postcondition [DS90] and has some structural similarities with the evaluation function $\hat{\xi}_c$ defined in Section 5.1.3.

Translation function $\hat{\mathcal{T}}_c$ has the following signature;

$$\hat{\mathcal{T}}_c : \hat{\mathbb{C}} \to (\hat{\Xi} \times V \to V),$$

where V is the product $\mathbb{Q} \times \mathcal{P}(\mathbb{Q}) \times \mathbb{M} \times S$. A tuple $(q, u_{\mathbb{Q}}, m, s)$ gathers the following information:

- a formula q corresponding to the strongest postcondition,
- a set of formulas that must be valid such that q can be considered as the strongest postcondition (*i.e.* verification condition for the assertions, invariant,... found in the program),
- an array variable m modeling the current memory state (similar to the memory state σ in case of function $\hat{\xi}_c$ defined in Section 5.1.3),

- a tuple s defining the current naming choices.

Since in the following definition of function $\hat{\mathcal{T}_c}$, the environment for annotation $\hat{\xi}$ is never modified, we can consider it implicitly and give the following shorthand:

$$\hat{\mathcal{T}_c}[\![\varsigma]\!]v = \hat{\mathcal{T}_c}[\![\varsigma]\!](\hat{\xi}, v).$$

As the definition of function $\hat{\mathcal{T}_c}$ is long, we split the definition in different parts for better readability. First, we define function $\hat{\mathcal{T}_c}$ for basic commands (skip, assignment, assertion and sequence). Then, we define function $\hat{\mathcal{T}_c}$ for command **call**. The definition of function $\hat{\mathcal{T}_c}$ for commands **if** and **while** can be found in Appendices B.1 and B.2 as they are not essential in the following.

Definition 5.13. Function $\hat{\mathcal{T}_c} : \hat{\mathbb{C}} \to (\hat{\Xi} \times V \to V)$, translating a command $\hat{\mathbb{C}}$ (case of skip, assignment, sequence and assertion) into an formula of $\mathbb{Q}$, is defined by structural induction on commands:

$$\hat{\mathcal{T}_c}[\![l : \textbf{skip}]\!](q, u_{\mathbb{Q}}, m, (\theta, \delta)) = (q, u_{\mathbb{Q}}, m, (\theta[l \leftarrow m], \delta))$$

$$\hat{\mathcal{T}_c}[\![l : x := a]\!](q, u_{\mathbb{Q}}, m, (\theta, \delta)) = ([\![q \wedge m' = m[e_x \leftarrow e]]\!], u_{\mathbb{Q}}, m', (\theta[l \leftarrow m], \delta_2))$$

where

(i) $(e, \delta_1) = \mathcal{T}_a[\![a]\!](m, \delta)$,

(ii) $(\delta_2, e_x) = \mathit{lift}(\delta_1, x)$,

(iii) $m' = \mathcal{N}_{v_m}$;

$$\hat{\mathcal{T}_c}[\![\varsigma_0; \varsigma_1]\!]v = \hat{\mathcal{T}_c}[\![\varsigma_1]\!](\hat{\mathcal{T}_c}[\![\varsigma_0]\!]v)$$

$$\hat{\mathcal{T}_c}[\![l : \textbf{assert}(\beta)]\!](q, u_{\mathbb{Q}}, m, (\theta, \delta)) = ([\![q \wedge q']\!], u_{\mathbb{Q}} \cup \{q \Rightarrow q'\}, m, s)$$

where $(q', s) = \hat{\mathcal{T}_b}[\![\beta]\!](\theta[l \leftarrow m], \delta)$

For commands skip, assignment and assertion we add to environment θ the fact that the current label refers to the array variable modeling the memory state before the evaluation (equivalent to the binding added to λ for function $\hat{\xi_c}$).

In case of command skip, as nothing changes on the memory state, the array variable modeling the memory state before and after the evaluation of the command is the same.

In case of assignment,

(i) we translate the arithmetic expression a into a term e,

(ii) we choose a natural for location x,

(iii) we choose a new array variable for modeling the memory state after the evaluation of the command.

Finally, we add to formula q the fact that the array variable modeling the memory state after the evaluation (m') is equal to the array variable modeling the memory state before the evaluation (m) with the position corresponding to location x updated with the term corresponding to the arithmetic expression of the assignment.

In the case of assertion, we translate the extended boolean expression β into a formula and add to the set of verification conditions the fact that the assertion must hold assuming q.

Example 5.11. We consider the following well defined program composed of assignment, skip, sequence and assertions.

$$\begin{array}{l} l_1 : x_1 := x_2 + 2; \\ l_2 : \textbf{skip}; \\ l_3 : x_1 := x_1 + 3; \\ l_4 : \textbf{assert}(at(x_1, l_4) = at(x_2, l_1) + 5); \\ l_5 : x_1 := x_1 + 1 \end{array}$$

We can transform this program in a first order formula, using function $\hat{\mathcal{T}}_c$, starting from assumption T. We propose to present the transformation progressively.

$$\begin{array}{l} l_1 : x_1 := x_2 + 2; \\ l_2 : \textbf{skip}; \\ l_3 : x_1 := x_1 + 3; \\ l_4 : \textbf{assert}(at(x_1, l_4) = at(x_2, l_1) + 5); \\ l_5 : x_1 := x_1 + 1 \end{array} \qquad \begin{array}{l} T \wedge \\ m_{l_2} = m_{l_1}[1 \leftarrow m_{l_1}[2] + 2] \end{array}$$

In the case of the first assignment, the memory state at label l_1 is modeled by array variable m_{l_1}. The next memory state (at label l_2) is modeled by array variable m_{l_2}. Location x_1 is modeled by position 1 in the arrays and location x_2 by position 2.

$$\begin{array}{l} l_1 : x_1 := x_2 + 2; \\ l_2 : \textbf{skip}; \\ l_3 : x_1 := x_1 + 3; \\ l_4 : \textbf{assert}(at(x_1, l_4) = at(x_2, l_1) + 5); \\ l_5 : x_1 := x_1 + 1 \end{array} \qquad \begin{array}{l} T \wedge \\ m_{l_2} = m_{l_1}[1 \leftarrow m_{l_1}[2] + 2] \end{array}$$

The skip command adds nothing in the formula, the array modeling the memory state at label l_3 is the same as the one at label l_2.

$$\begin{array}{l} l_1 : x_1 := x_2 + 2; \\ l_2 : \textbf{skip}; \\ l_3 : x_1 := x_1 + 3; \\ l_4 : \textbf{assert}(at(x_1, l_4) = at(x_2, l_1) + 5); \\ l_5 : x_1 := x_1 + 1 \end{array} \qquad \begin{array}{l} T \wedge \\ m_{l_2} = m_{l_1}[1 \leftarrow m_{l_1}[2] + 2] \wedge \\ m_{l_4} = m_{l_2}[1 \leftarrow m_{l_2}[1] + 3] \end{array}$$

The second assignment has similar effects as the first one.

$$
\begin{array}{ll}
 & T\wedge \\
 & m_{l_2} = m_{l_1}[1 \leftarrow m_{l_1}[2] + 2]\wedge \\
l_1 : x_1 := x_2 + 2; & m_{l_4} = m_{l_2}[1 \leftarrow m_{l_2}[1] + 3]\wedge \\
l_2 : \textbf{skip}; & m_{l_4}[1] = m_{l_1}[2] + 5 \\
l_3 : x_1 := x_1 + 3; & \\
l_4 : \textbf{assert}(at(x_1, l_4) = at(x_2, l_1) + 5); & T\wedge \\
l_5 : x_1 := x_1 + 1 & m_{l_2} = m_{l_1}[1 \leftarrow m_{l_1}[2] + 2]\wedge \\
 & m_{l_4} = m_{l_2}[1 \leftarrow m_{l_2}[1] + 3] \\
 & \Rightarrow m_{l_4}[1] = m_{l_1}[2] + 5
\end{array}
$$

An assertion results in a verification condition for the assertion itself, stating that the strongest postcondition generated up to this point implies the assertion itself. In addition, the assertion is assumed in the main formula and the array modeling the memory states at label l_5 is the same as the one at label l_4.

$$
\begin{array}{ll}
 & T\wedge \\
 & m_{l_2} = m_{l_1}[1 \leftarrow m_{l_1}[2] + 2]\wedge \\
 & m_{l_4} = m_{l_2}[1 \leftarrow m_{l_2}[1] + 3]\wedge \\
l_1 : x_1 := x_2 + 2; & m_{l_4}[1] = m_{l_1}[2] + 5\wedge \\
l_2 : \textbf{skip}; & m_{next} = m_{l_4}[1 \leftarrow m_{l_4}[1] + 1] \\
l_3 : x_1 := x_1 + 3; & \\
l_4 : \textbf{assert}(at(x_1, l_4) = at(x_2, l_1) + 5); & T\wedge \\
l_5 : x_1 := x_1 + 1 & m_{l_2} = m_{l_1}[1 \leftarrow m_{l_1}[2] + 2]\wedge \\
 & m_{l_4} = m_{l_2}[1 \leftarrow m_{l_2}[1] + 3] \\
 & \Rightarrow m_{l_4}[1] = m_{l_1}[2] + 5
\end{array}
$$

The last assignment has similar effects as the previous ones on the main formula.

We now define function $\hat{\mathcal{T}}_c$ for the command **call**. In this case, we want to use the contract in $\hat{\xi}$ to link the state before and after the procedure call.

Example 5.12. We consider the following program composed of assignments and a procedure call to y.

$$
\begin{array}{l}
l_1 : x := x + 1; \\
l_2 : \textbf{call}(y); \\
l_3 : x := x + 1
\end{array}
$$

- If we have no contract for procedure call y we expect following result;

$$
\begin{array}{ll}
l_1 : x := x + 1; & m_{l_2} = m_{l_1}[1 \leftarrow m_{l_1}[1] + 1]\wedge \\
l_2 : \textbf{call}(y); & m_{l_{next}} = m_{l_3}[1 \leftarrow m_{l_3}[1] + 1] \\
l_3 : x := x + 1 &
\end{array}
$$

Here, we have no relation between array variable m_{l_2}, modeling the state at label l_2, and array variable m_{l_3}, modeling the state at label l_3, since we have no information on the effects of the function on memory.

- If we have a contract for y in an environment $\hat{\xi}$ defined by;

$$\{y \to (true, at(x, Post) = at(x, Pre) + 1)\},$$

we expect following result:

$$\begin{array}{ll} l_1 : x := x + 1; & m_{l_2} = m_{l_1}[1 \leftarrow m_{l_1}[1] + 1] \wedge \\ l_2 : \textbf{call}(y); & m_{l_3}[1] = m_{l_2}[1] + 1 \wedge \\ l_3 : x := x + 1 & m_{l_{next}} = m_{l_3}[1 \leftarrow m_{l_3}[1] + 1] \end{array}$$

Since the precondition always holds, we can use the postcondition to get a relation between array variable m_{l_2} and array variable m_{l_3}. However, the relation only concerns location x, which can be limiting.

A solution to the previous limitation is to assume we have a set of locations $u_{\mathbb{X}}$ that are specified as assigned *i.e.* none of the locations outside of $u_{\mathbb{X}}$ are modified. Using this information we can connect the state before and after a procedure call completely.

Example 5.13. Assume we have a set of assigned locations $u_{\mathbb{X}}$ defined by $\{x\}$. We can refine the previous example as follows:

$$\begin{array}{ll} l : x := x + 1; & m_{l_2} = m_{l_1}[1 \leftarrow m_{l_1}[1] + 1] \wedge \\ l_2 : \textbf{call}(y); & m_{l_3}[1] = m_{l_2}[1] + 1 \wedge m_{l_3} = m_{l_2}[1 \leftarrow i] \\ l_3 : x := x + 1 & m_{l_{next}} = m_{l_3}[1 \leftarrow m_{l_3}[1] + 1] \end{array}$$

The green part of the formula represents the fact that only location x has been assigned, *i.e.* m_{l_3} is identical to m_{l_2}, except for cell 1 (corresponding to location x), which is updated with an arbitrary value, represented in the formula by a free variable i. This gives a complete knowledge of the state after a procedure call, since we know that only x has been assigned, and the postcondition says how.

This approach is well-known as *frame rule*. Thus, we are only interested in using the specification in the following. Solutions for proving the *frame rule* can be found, for example, in [Moy09]. We simply assume the existence of a function $\hat{\mathcal{VC}}_f : \hat{\mathbb{C}} \times \hat{\mathbb{E}}_b \times \hat{\Xi} \times \mathcal{P}(\mathbb{X}) \to \mathcal{P}(\mathbb{Q})$ that returns the set of formulas that must be valid in order for the frame rule $\mathcal{P}(\mathbb{X})$ to be valid, for a program $\hat{\mathbb{C}}$, a precondition $\hat{\mathbb{E}}_b$ and an environment for contract $\hat{\Xi}$.

For the definition of function $\hat{\mathcal{T}}_c$, we assume a function $\mathcal{N}_{v_n}$ returning a fresh natural variable. As before, we write $v = \mathcal{N}_{v_n}$ when we want a new natural variable v. We also refine the definition of the environment for contracts to support the *frame rule*;

$$\hat{\Xi} = \mathbb{Y} \rightharpoonup \hat{\mathbb{E}}_b \times \hat{\mathbb{E}}_b \times \mathcal{P}(\mathbb{X}),$$

where $\mathcal{P}(\mathbb{X})$ represents the set of assigned locations.

Definition 5.14. Function $\hat{\mathcal{T}}_c : \hat{\mathbb{C}} \to (\hat{\Xi} \times V \to V)$, translating a command $\hat{\mathbb{C}}$ (case of procedure call) into a formula of $\mathbb{Q}$, is defined by:

$$\hat{\mathcal{T}}_c[\![l : \textbf{call}(y)]\!](q, u_{\mathbb{Q}}, m, (\theta, \delta)) =$$

$$\begin{cases} (\llbracket q \wedge q_{Pre} \wedge q_{Post} \wedge m' = m[e_1 \leftarrow v_1]...[e_n \leftarrow v_n] \rrbracket, \\ \qquad\qquad u_{\mathbb{Q}} \cup \{q \Rightarrow q_{Pre}\}, m', (\theta[l \leftarrow m], \delta_n)) \\ \qquad \text{if } \hat{\xi}(y) = (\beta_{Pre}, \beta_{Post}, \{x_1, ..., x_n\}), \\ \quad \text{where} \\ \quad \text{(i)} \quad (q_{Pre}, (\_, \delta_{Pre})) = \hat{\mathcal{T}}_b[\![\beta_{Pre}]\!](\{Pre \leftarrow m\}, \delta), \\ \quad \text{(ii)} \quad m' = \mathcal{N}_{v_m}, \\ \quad \text{(iii)} \quad (q_{Post}, (\_, \delta_{Post}) = \hat{\mathcal{T}}_b[\![\beta_{Post}]\!](\{Pre \leftarrow m, Post \leftarrow m'\}, \delta_{Pre}), \\ \quad \text{(iv)} \quad v_1 = \mathcal{N}_{v_n}, ..., v_n = \mathcal{N}_{v_n}, \\ \quad \text{(v)} \quad (\delta_1, e_1) = \mathit{lift}(\delta_{Post}, x_1), ..., (\delta_n, e_n) = \mathit{lift}(\delta_{n-1}, x_n); \\ \\ (q, u_{\mathbb{Q}}, m', (\theta[l \leftarrow m], \delta)) \qquad \text{if } \hat{\xi}(y) = \bot, \\ \quad \text{where } m' = \mathcal{N}_{v_m}. \end{cases}$$

(i) First, we translate the precondition β_{Pre} into a formula q_{Pre}.

(ii) Then, we define a new array m' modeling the state after the procedure call.

(iii) Then, we translate the postcondition β_{Post} into a formula q_{Post}.

(iv) Then, for each assigned location, we define a new natural variable.

(v) Then, for each assigned location, we get the corresponding index in the array representing memory state.

Finally, we add to the set of verification conditions the verification condition stating that the strongest postcondition generated up to this point implies the precondition q_{Pre}. We also add to q the precondition q_{Pre}, the postcondition q_{Post} and the frame rule.

5.3.4 Verification of Hoare Triples

Using the function $\hat{\mathcal{T}}_c$, we can define function $\hat{\mathcal{VC}}_h$ returning the set of formulas (verification conditions) for a given Hoare Triple.

$$\hat{\mathcal{VC}}_h : \hat{\mathbb{C}} \times (\hat{\mathbb{E}}_b \times \hat{\mathbb{E}}_b) \times \hat{\Xi} \to \mathcal{P}(\mathbb{Q})$$

Function $\hat{\mathcal{VC}}_h$ takes the triple in the form of a command and a couple of boolean expressions (the pre- and post-conditions). As stated earlier, the precondition uses label Pre to model the current state and the postcondition uses label $Post$ to model the current state ($\hat{\mathcal{W}}_a(\hat{\xi}, \sigma)$). The definition of the function $\hat{\mathcal{VC}}_h$ is therefore as follows:

Definition 5.15. Function $\hat{\mathcal{VC}}_h : \hat{\mathbb{C}} \times (\hat{\mathbb{E}}_b \times \hat{\mathbb{E}}_b) \times \hat{\Xi} \to \mathcal{P}(\mathbb{Q})$ returning the set of verification conditions for a Hoare Triple:

$$\hat{\mathcal{VC}}_h(\varsigma, (\beta_{Pre}, \beta_{Post}), \hat{\xi}) = u_{\mathbb{Q}} \cup \{q \Rightarrow q_{Post}\}$$

where

(i) $m = \mathcal{N}_{v_m}$,

(ii) $(q_{Pre}, s) = \hat{\mathcal{T}}_b[\![\beta_{Pre}]\!](\{Pre \to m\}, \emptyset)$,

(iii) $(q, u_{\mathbb{Q}}, m', (\theta', \delta')) = \hat{\mathcal{T}}_c[\![\varsigma]\!](q_{Pre}, \emptyset, m, s)$,

(iv) $(q_{Post}, \_) = \hat{\mathcal{T}}_b[\![\beta_{Post}]\!](\theta'[Post \leftarrow m'], \delta')$.

(i) First, we define a new array variable m modeling the memory state at label Pre.

(ii) Then, we translate the precondition β_{Pre} into a formula q_{Pre}.

(iii) Then, we translate the command ς into a formula q, by taking the formula q_{Pre}, corresponding to the precondition, as assumption.

(iv) Then, we translate the post condition into a formula of q_{Post}.

The final set of formulas is composed of the set of sub-formulas obtained during the translation of the command, and the formula corresponding to the fact that the postcondition must hold after the program execution assuming the precondition.

Example 5.14. If we consider the following well defined Hoare Triple using the program shown in Example 5.11:

$$\{\ at(x_2, Pre) = 4\ \}\quad \begin{array}{l} l_1 : x_1 := x_2 + 2; \\ l_2 : \textbf{skip}; \\ l_3 : x_1 := x_1 + 3; \\ l_4 : \textbf{assert}(at(x_1, l_4) = at(x_2, l_1) + 5); \\ l_5 : x_1 := x_1 + 1; \end{array} \quad \{\ at(x_1, Post) = 10\ \}$$

We generate the associated verification condition using function $\hat{\mathcal{VC}}_h$ (in this case the state before and after the evaluation are associated to the reserved labels Pre and $Post$).

$$\begin{array}{l} m_{Pre}[2] = 4 \wedge \\ m_{l_2} = m_{Pre}[1 \leftarrow m_{Pre}[2] + 2] \wedge \\ m_{l_4} = m_{l_2}[1 \leftarrow m_{l_2}[1] + 3] \wedge \\ m_{l_4}[1] = m_{Pre}[2] + 5 \wedge \\ m_{Post} = m_{l_4}[1 \leftarrow m_{l_4}[1] + 1] \\ \Rightarrow m_{Post}[1] = 10 \end{array} \qquad \begin{array}{l} m_{Pre}[2] = 4 \wedge \\ m_{l_2} = m_{Pre}[1 \leftarrow m_{Pre}[2] + 2] \wedge \\ m_{l_4} = m_{l_2}[1 \leftarrow m_{l_2}[1] + 3] \\ \Rightarrow m_{l_4}[1] = m_{Pre}[2] + 5 \end{array}$$

Using function smt (and axioms Q-NO-UPDATE and Q-UPDATE) we can show that those formulas are provable.

Using functions $\hat{\mathcal{VC}}_h$ and $\hat{\mathcal{VC}}_f$, we can define function $\hat{\mathcal{VC}}_p$ generating all verification conditions corresponding to the contracts defined in an environmnt $\hat{\xi}$.

Definition 5.16. Function $\hat{\mathcal{VC}}_p : \hat{\Psi} \times \hat{\Xi} \to \mathcal{P}(\mathbb{Q})$, returning the set of verification conditions that must be valid in order for the procedure contracts $\hat{\Psi}$ to be valid:

$$\hat{\mathcal{VC}}_p(\hat{\psi}, \hat{\xi}) = \bigcup_{y \in dom(\hat{\xi})} \hat{\mathcal{VC}}_h(\hat{\psi}(y), (\beta_{Pre}, \beta_{Post}), \hat{\xi}) \cup \hat{\mathcal{VC}}_f(\hat{\psi}(y), \beta_{Pre}, u_{\mathbb{X}}, \hat{\xi})$$

$$\text{where } (\beta_{Pre}, \beta_{Post}, u_{\mathbb{X}}) = \hat{\xi}(y)$$

We can now give rule E-RECURSION for the verification of Hoare Triple in the case of R-WHILE*, for a set of axioms $u_{\mathbb{Q}}$.

$$\frac{\forall q \in \hat{\mathcal{VC}}_h(\varsigma, (\beta_1, \beta_2), \hat{\xi}) \cup \hat{\mathcal{VC}}_p(\hat{\psi}, \hat{\xi}).\mathrm{smt}(u_{\mathbb{Q}}, q) = V}{\vdash \{\beta_1\}\varsigma\{\beta_2\}} \quad \text{(E-RECURSION)}$$

The rule stat that, if all formulas returned by function $\hat{\mathcal{VC}}_p$ for a given set of contracts are valid, and all formulas returned by function $\hat{\mathcal{VC}}_h$ are valid, the corresponding Hoare Triple is valid. Note the similarities with rule RECURSION.

Chapter 6

Source code transformation

In Chapter 4, we presented three deductive verification methods for relational properties. Those methods can be used to prove relational properties, but are unable to use relational properties as hypotheses in the subsequent verifications. However, in modular deductive verification, using a set of assumptions is a key point as already shown in Sections 3.3 and 5.3.

To solve this limitation, we propose in this chapter a modular deductive verification of relational properties in the context of the R-WHILE* language. The verification of relational properties relies on self-composition and generation of verification conditions. The ability to use relational properties relies on axiomatization of procedure call.

The chapter is organized as follows. First we adapt in Section 6.1 the definition of relational properties, given in Section 4.1, to extended commands $\hat{\mathbb{C}}$ and extended boolean expressions $\hat{\mathbb{E}}_b$. In Section 6.2, we recall the notion of self-composition, in the context of the R-WHILE* language, combined with function $\hat{\mathcal{VC}}_h$ for the generation of verification conditions. Finally, in Section 6.3 we present our solution for using relational properties as hypotheses.

6.1 Relational Properties and Labels

Adapting the definition of relational properties to extended commands $\hat{\mathbb{C}}$ and extended boolean expressions $\hat{\mathbb{E}}_b$ is a straightforward combination of the definition of Hoare Triples given in Section 5.3 and relational properties given in Section 4.1. To distinguish the syntax and functions defined in this chapter from those in the previous chapters, we add the symbol $\tilde{\hat{}}$ (relational extended), $\tilde{}$ (relational) or $\hat{}$ (extended) on each new syntax and function.

First we give the grammar rules for relational extended arithmetic expressions $\tilde{\hat{\mathbb{E}}}_a$ and rela-

tional extended boolean expressions $\tilde{\hat{\mathbb{E}}}_b$:

$$\begin{aligned}\tilde{\alpha} ::= & \; n \\ & \mid at(x,l)\langle t\rangle \\ & \mid \tilde{\alpha}\ opa\ \tilde{\alpha}\end{aligned}$$

$$\begin{aligned}\tilde{\beta} ::= & \; true \mid false \\ & \mid \tilde{\alpha}_1\ opb\ \tilde{\alpha}_2 \\ & \mid \tilde{\beta}_1\ opl\ \tilde{\beta}_2 \mid \neg\tilde{\beta}' \\ & \mid \tilde{\beta}_0 \Rightarrow \tilde{\beta}_1 \\ & \mid p^n(\tilde{\alpha}_1, ..., \tilde{\alpha}_n)\end{aligned}$$

As for $\tilde{b}$ and $\tilde{a}$, the only difference of $\tilde{\beta}$ and $\tilde{\alpha}$ with the grammar rules proposed in Section 5.1 for the R-WHILE* language, is the use of notation $\langle t\rangle$ in arithmetic expressions for tagging locations.

In Section 5.1 we have defined evaluation functions for β and α using an environment Λ that maps labels to states of natural numbers. This environment is used to get the value of a location in a state at a given reachable label. Since we have now multiple states associated to programs identified by tags, we define the relational extended state environment $\hat{\Phi}$ that maps tags to an environment Λ.

$$\hat{\Phi} = \mathbb{T} \rightharpoonup \Lambda$$

and use metavariables $\hat{\phi}, \hat{\phi}_0, \hat{\phi}_1, ...$ to range over $\hat{\Phi}$.

We define the evaluation functions for $\tilde{\hat{\mathbb{E}}}_a$ and $\tilde{\hat{\mathbb{E}}}_b$ as follows.

Definition 6.1. Evaluation function $\tilde{\hat{\xi}}_a : \tilde{\hat{\mathbb{E}}}_a \to (\hat{\Phi} \to \mathbb{N}_\perp)$ for relational extended arithmetic expressions $\tilde{\hat{\mathbb{E}}}_a$, is defined by structural induction on relational extended arithmetic expressions:

$$\begin{aligned}\tilde{\xi}_a[\![n]\!]\hat{\phi} &= n \\ \tilde{\xi}_a[\![at(x,l)\langle t\rangle]\!]\hat{\phi} &= (((\hat{\phi}(t))l)x \\ \tilde{\xi}_a[\![\tilde{\alpha}_0\ opa\ \tilde{\alpha}_1]\!]\hat{\phi} &= \tilde{\xi}_a[\![\tilde{\alpha}_0]\!]\hat{\phi}\ opa_\perp\ \tilde{\xi}_a[\![\tilde{\alpha}_1]\!]\hat{\phi}\end{aligned}$$

Definition 6.2. Evaluation function $\hat{\tilde{\xi}}_b : \hat{\tilde{\mathbb{E}}}_b \to (\hat{\Phi} \to \mathbb{Q}_\perp)$, for relational extended boolean expression $\hat{\tilde{\mathbb{E}}}_b$, is defined by structural induction on relational extended boolean expressions:

$$\hat{\tilde{\xi}}_b[\![true]\!]\hat{\phi} = [\![T]\!]$$

$$\hat{\tilde{\xi}}_b[\![false]\!]\hat{\phi} = [\![F]\!]$$

$$\hat{\tilde{\xi}}_b[\![\alpha_0\ opb\ \alpha_1]\!]\hat{\phi} = \begin{cases} \perp & \text{if } \hat{\tilde{\xi}}_a[\![\alpha_0]\!]\hat{\phi} = \perp \text{ or } \hat{\tilde{\xi}}_a[\![\alpha_1]\!]\hat{\phi} = \perp \\ [\![\hat{\tilde{\xi}}_a[\![\alpha_0]\!]\hat{\phi}\ opb\ \hat{\tilde{\xi}}_a[\![\alpha_1]\!]\hat{\phi}]\!] \end{cases}$$

$$\hat{\tilde{\xi}}_b[\![\beta_0\ opl\ \beta_1]\!]\hat{\phi} = \begin{cases} \perp & \text{if } \hat{\tilde{\xi}}_b[\![\beta_0]\!]\hat{\phi} = \perp \text{ or } \hat{\tilde{\xi}}_b[\![\beta_1]\!]\hat{\phi} = \perp \\ [\![\hat{\tilde{\xi}}_b[\![\beta_0]\!]\hat{\phi}\ opl\ \hat{\tilde{\xi}}_b[\![\beta_1]\!]\hat{\phi}]\!] \end{cases}$$

$$\hat{\tilde{\xi}}_b[\![\neg\beta]\!]\hat{\phi} = \begin{cases} \perp & \text{if } \hat{\tilde{\xi}}_b[\![\beta]\!]\hat{\phi} = \perp \\ [\![\neg\hat{\tilde{\xi}}_b[\![\beta]\!]\hat{\phi}]\!] \end{cases}$$

$$\hat{\tilde{\xi}}_b[\![\beta_0 \Rightarrow \beta_1]\!]\hat{\phi} = \begin{cases} \perp & \text{if } \hat{\tilde{\xi}}_b[\![\beta_0]\!]\hat{\phi} = \perp \text{ or } \hat{\tilde{\xi}}_b[\![\beta_1]\!]\hat{\phi} = \perp \\ [\![\hat{\tilde{\xi}}_b[\![\beta_0]\!]\hat{\phi} \Rightarrow \hat{\tilde{\xi}}_b[\![\beta_1]\!]\hat{\phi}]\!] \end{cases}$$

$$\hat{\tilde{\xi}}_b[\![p^n(\alpha_1, ..., \alpha_n)]\!]\hat{\phi} = \begin{cases} \perp & \text{if } \hat{\tilde{\xi}}_a[\![\alpha_1]\!]\hat{\phi} = \perp \text{ or ... or } \hat{\tilde{\xi}}_a[\![\alpha_n]\!]\hat{\phi} = \perp \\ [\![\mathrm{p}(\hat{\tilde{\xi}}_a[\![\alpha_1]\!]\hat{\phi}, ..., \hat{\tilde{\xi}}_a[\![\alpha_n]\!]\hat{\phi})]\!] \end{cases}$$

As for function $\hat{\xi}_b$ in Section 5.1.2, constant terms, boolean and logical operators and predicates are implicitly translated into the MFOL language.

We now refine the definition of relational execution environment Φ_c (defined in Section 4.1) to relational extended excution environment $\hat{\Phi}_c$, mapping tags to the pair composed of a extended command and a memory state for extended commands.

$$\hat{\Phi}_c = \mathbb{T} \rightharpoonup \hat{\mathbb{C}} \times \hat{\Psi}$$

As for Φ_c we use projection functions *body* and *state* to access the command and the memory state for command.

We also refine the definition of environment Φ_a, the environment that maps tags to an environment of procedure contracts, into $\hat{\Phi}_a$, the environment that maps tags to an environment of procedure contracts of extended commands, defined by

$$\hat{\Phi}_a = \mathbb{T} \rightharpoonup \hat{\Xi}$$

We now combine the definition of function $\hat{\xi}_c$ (defined in Section 5.1.3), for the evaluation of extended commands, and function $\tilde{\xi}_c$ (defined in Section 4.1), for the evaluation of relational properties, to get function $\hat{\tilde{\xi}}_c$, for the evaluation of relational extended execution environment $\hat{\phi}_c$.

Definition 6.3. Evaluation function $\tilde{\hat{\xi}}_c : \hat{\Phi}_c \times \Phi_\Omega \times \hat{\Phi} \times \mathcal{P}(\mathbb{Q}) \rightarrow \Phi_\Omega \times \hat{\Phi}$, for relational extended execution environment $\hat{\Phi}_c$:

$$\tilde{\hat{\xi}}_c(\hat{\phi}_c, \phi, \hat{\phi}, u_\mathbb{Q}) = (\phi[t_1 \leftarrow \sigma_1]...[t_n \leftarrow \sigma_n], \hat{\phi}[t_1 \leftarrow \lambda_1]...[t_n \leftarrow \lambda_n])$$

where

(i) $\quad \{t_1, ..., t_n\} = dom(\hat{\phi}_c)$,

(ii) $\quad (\sigma_i, \lambda_i) = \dot{\hat{\xi}}_c[\![body(\hat{\phi}_c(t_i))]\!](u_\mathbb{Q}, state(\hat{\phi}_c(t_i)), (\phi(t_i), \hat{\phi}(t_i)))$.

As for function $\tilde{\xi}_c$, function $\tilde{\hat{\xi}}_c$ evaluates all commands defined in $\hat{\phi}_c$ with the associated memory state defined in environment ϕ and environment λ defined in $\hat{\phi}$ using evaluation function for extended commands $\dot{\hat{\xi}}_c$. The environments ϕ and $\hat{\phi}$ are updated with the resulting memory states and environment, mapping labels to memory states, respectively.

We are now refining some hypotheses to avoid evaluation ending in state $\Omega_\perp$ and $\perp$. Most assumptions are similar to those defined in Section 4.1, using constructions defined in Section 5.1.4:

- There is no command evaluation on undefined memory states :

$$dom(\hat{\phi}_c) \subseteq dom(\phi)$$

- The sets of tags used in the relational boolean expressions $\tilde{\beta}_1$ and $\tilde{\beta}_2$ are defined in $\hat{\phi}_c$:

$$\tilde{\mathcal{C}}_{t_b}[\![\tilde{\beta}_1]\!] \subseteq dom(\hat{\phi}_c)$$
$$\tilde{\mathcal{C}}_{t_b}[\![\tilde{\beta}_2]\!] \subseteq dom(\hat{\phi}_c)$$

  where function $\tilde{\mathcal{C}}_{t_b}$ (defined in Appendix A.1.4) returns the set of tags used in a relational boolean expression.

- The sets of locations used in the context of a tag in $\tilde{\beta}_1$ and $\tilde{\beta}_2$ are defined in the memory state associated to that tag:

$$\forall t \in \tilde{\hat{\mathcal{C}}}_{t_b}[\![\tilde{\beta}_1]\!], \tilde{\hat{\mathcal{C}}}_{v_b}[\![\tilde{\beta}_1]\!]t \subseteq dom(\phi(t))$$

$$\forall t \in \tilde{\hat{\mathcal{C}}}_{t_b}[\![\tilde{\beta}_2]\!], \tilde{\hat{\mathcal{C}}}_{v_b}[\![\tilde{\beta}_2]\!]t \subseteq dom(\phi(t))$$

  where function $\tilde{\hat{\mathcal{C}}}_{v_b}$ (defined in Appendix A.1.4) returns the set of locations associated to a given tag in a relational boolean expression.

- For a given tag, the set of labels used in the precondition $\tilde{\beta}_1$ is a subset of the set composed of label Pre. The set of labels used in the postcondition $\tilde{\beta}_2$ is a subset of the set composed of the labels we can refer through the command for that tag and labels Pre and $Post$:

$$\forall t \in dom(\phi).\tilde{\hat{\mathcal{C}}}_{l_b}[\![\beta_1]\!]t \subseteq \{Pre\}$$

$$\forall t \in dom(\phi).\tilde{\hat{\mathcal{C}}}_{l_b}[\![\beta_2]\!]t \subseteq \hat{\mathcal{W}}[\![body(\hat{\phi}_c(t))]\!]\emptyset \cup \{Pre, Post\}$$

where function $\tilde{\hat{\mathcal{C}}}_{l_b}$ (defined in Appendix A.1.4) returns the set of labels used in a boolean expression $\mathbb{E}_b$ for a given tag and function $\hat{\mathcal{W}}$ (defined in Section 5.1.4) returning the set of reachable labels from an initial set of reachable labels and a command.

- All defined environments of procedure contracts for extended commands, in an environment $\hat{\phi}_a$, are well defined; the set of variables used in the contracts is a subset of the domain of the memory state associated to the corresponding tag, the set of labels used in the precondition is a subset of set composed of label Pre, the set of labels used in the postcondition is a subset of the set composed of labels Pre and $Post$:

  $$\forall t \in dom(\hat{\phi}_a).\hat{\mathcal{W}}_a(\hat{\phi}_a(t), \phi(t))$$

  where predicate $\hat{\mathcal{W}}_a$ is defined in Section 5.3 and ensures that an environment of procedure contracts for extended commands is well defined.

- There is one set of contracts for each tag belonging to the domain of $\hat{\phi}_c$:

  $$dom(\hat{\phi}_a) = dom(\hat{\phi}_c)$$

- For each tag, environments ϕ and $\hat{\phi}_c$ forms a well defined well-defined program:

  $$\forall t \in dom(\hat{\phi}_c).\hat{\mathcal{WD}}(body(\hat{\phi}_c(t)), state(\hat{\phi}_c(t)), \phi(t), \emptyset) \wedge \{Pre, Post\} \subsetneq \hat{\mathcal{C}}_{l_c}[\![body(\hat{\phi}_c(t))]\!]$$

  where predicate $\hat{\mathcal{WD}}$ is defined in Section 5.1.4.

We can now combine the definition of extended Hoare Triple (defined in Section 5.2) and relational properties (defined in Section 4.1), and using function $\tilde{\hat{\xi}}_a$, $\tilde{\hat{\xi}}_b$ and $\tilde{\hat{\xi}}_c$ to give the definition of extended relational properties.

Definition 6.4. Extended Relational Properties

$$\forall \phi \in \Phi.\hat{\phi} \models \tilde{\beta}_1 \Rightarrow \hat{\phi}'' \models \tilde{\beta}_2$$

where

(i) $\{t_1, ..., t_n\} = dom(\phi)$,

(ii) $\hat{\phi} = \{t_1 \rightarrow [Pre \rightarrow \phi(t_1)], ..., t_n \rightarrow [Pre \rightarrow \phi(t_n)]\}$,

(iii) $(\phi', \hat{\phi}') = \tilde{\hat{\xi}}_c(\hat{\phi}_c, \phi, \hat{\phi}, u_{\mathbb{Q}})$,

(iv) $\hat{\phi}'' = \{t_1 \rightarrow \hat{\phi}'(t_1)[Post \leftarrow \phi'(t_1)], ..., t_n \rightarrow \hat{\phi}'(t_n)[Post \leftarrow \phi'(t_n)\}$.

Expression $\hat{\phi} \models \tilde{\beta}$ states that an environment $\hat{\phi}$ satisfies a relational extended boolean expression $\tilde{\beta}$ *i.e* $\tilde{\hat{\xi}}_b[\![\tilde{\beta}]\!]\hat{\phi}$ must be different from $\perp$ and $\mathrm{smt}(u_{\mathbb{Q}}, \tilde{\hat{\xi}}_b[\![\tilde{\beta}]\!]\hat{\phi})$ returns V, where $u_{\mathbb{Q}}$ is the set of given axioms. Moreover $\hat{\phi} \models \tilde{\beta}$ holds if an evaluation has not finished:

$$\exists t \in dom(\hat{\phi}).\exists l \in dom(\hat{\phi}(t)).(\hat{\phi}(t))(l) = \Omega_t,$$

i.e., there is a tag for which there is a label for which the associated state is Ω_t.

As for the definition of extended Hoare triple, where we consider the environment λ mapping labels to memory states to be initially empty, we enforce in the definition of extended relational properties the environment $\hat{\phi}$ to be initially empty (condition *(ii)* of Defintion 6.4). We use function $\hat{\tilde{\xi}}_c$ to evaluates the relational extended execution environment $\hat{\phi}_c$ (condition *(iii)* of Defintion 6.4). We update the relational extended state environment $\hat{\phi}$ with the final state of relational state environment ϕ for label $Post$ (condition *(iv)* of Defintion 6.4).

To denote an extended relational property, we reuse the notation used for relational properties defined in Section 4.1:

$$\{\tilde{\beta}_1\}\varsigma_1\langle t_1\rangle \sim \varsigma_2\langle t_2\rangle\{\tilde{\beta}_2\}.$$

Example 6.1. If we consider the following relational property:

$$\begin{gathered}\{at(x_1, Pre)\langle t_1\rangle = at(x_1, Pre)\langle t_2\rangle\}\\ l_1 : x_1 := x_1 + 1\langle t_1\rangle \sim l_1 : x_1 := x_1 + 1\langle t_2\rangle\\ \{at(x_1, Post)\langle t_1\rangle = at(x_1, Post)\langle t_2\rangle\}\end{gathered}$$

and an empty set of assumptions, we can prove the property valid by verifying the formula:

$$\forall \phi \in \Phi^2.\hat{\phi} \models at(x, Pre)\langle t_1\rangle = at(x, Pre)\langle t_2\rangle \Rightarrow \hat{\phi}' \models at(x, Post)\langle t_1\rangle = at(x, Post)\langle t_2\rangle$$

where

(i) $\quad \{t_1, t_2\} = dom(\phi)$,

(ii) $\quad \hat{\phi} = \{t_1 \to \{Pre \to \phi(t_1)\}, t_2 \to \{Pre \to \phi(t_2)\}\}$,

(iii) $\quad (\sigma_{t_1}, \lambda_{t_1}) = \dot{\hat{\xi}}_c[\![l : x := x + 1]\!](\emptyset, \emptyset, (\phi(t_1), \hat{\phi}(t_1)))$,

(iv) $\quad (\sigma_{t_2}, \lambda_{t_2}) = \dot{\hat{\xi}}_c[\![l : x := x + 1]\!](\emptyset, \emptyset, (\phi(t_2), \hat{\phi}(t_2)))$,

(v) $\quad \hat{\phi}' = \{t_1 \to \lambda_{t_1}[Post \leftarrow \sigma_{t_1}], t_2 \to \lambda_{t_2}[Post \leftarrow \sigma_{t_2}]\}$.

6.2 Self Composition

We have seen in Section 4.3 that we can use self-composition to prove relational properties by translating them into standard Hoare Triple. We present in this section how to apply this approach in the case of the R-WHILE* language.

From rule RECURSION-SELF-COMP we can refine a new rule RECURSIVE-SELF-COMP-E for proving relational properties using functions $\hat{\mathcal{VC}}_h$ and $\hat{\mathcal{VC}}_p$

$$\frac{\begin{array}{c}\forall t \in dom(\hat{\phi}_a).\forall q \in \hat{\mathcal{VC}}_p(state(\hat{\phi}_c(t)), \hat{\phi}_a(t)).\mathrm{smt}(u_{\mathbb{Q}}, q) = V\\ \forall q \in \hat{\mathcal{VC}}_h(\varsigma_1; \varsigma_2, (\hat{\tilde{\mathcal{D}}}_{t_b}[\![\tilde{\beta}_1]\!], \hat{\tilde{\mathcal{D}}}_{t_b}[\![\tilde{\beta}_2]\!]), \hat{\phi}_a(t_1) \cup \hat{\phi}_a(t_2)).\mathrm{smt}(u_{\mathbb{Q}}, q) = V\end{array}}{\vdash \{\tilde{\beta}_1\}\varsigma_1\langle t_1\rangle \sim \varsigma_2\langle t_2\rangle\{\tilde{\beta}_2\}}$$

(RECURSIVE-SELF-COMP-E)

where function $\hat{\tilde{\mathcal{D}}}_{t_b}$ (defined in Appendix A.4.1) removes all tags from a relational boolean expression of $\hat{\tilde{\mathbb{E}}}_b$. The principle of this rule is the same as for rule RECURSION-SELF-COMP;

all procedure contracts are proven valid using function $\hat{\mathcal{VC}}_p$ and smt. The relational property is proven valid by converting it into a Hoare Triple composed of the flattening of the precondition with respect to the tags, the sequence of the programs involved in the relational properties, and the flattening of the postcondition with respect to the tag. We use $\hat{\mathcal{VC}}_h$ and smt as for the verification of a standard Hoare Triple in R-WHILE*.

As already mentioned in Section 4.3, self-composition requires that some properties are satisfied. We refine here those properties in the context of the R-WHILE* language:

- The memory states for naturals must be disjoint:

  $$\forall t_1, t_2 \in dom(\hat{\phi}_c), \phi(t_1) \cap \phi(t_2) = \emptyset.$$

  In the previous section, we assumed that for each tag, the associated command, memory state for natural and memory state for commands are well defined (predicate $\hat{\mathcal{WD}}$). Thus, the set of variables used in the program (and in the called procedures) is equal to the domain of the memory state (predicate $\hat{\mathcal{WD}}$ includes predicate $\hat{\mathcal{W}}_v$). We can refine the property of disjoint memory states for naturals by ensuring the absence of shared locations between programs and procedures:

  $$\forall t_1, t_2 \in dom(\hat{\phi}_c), t_1 \neq t_2 \Rightarrow$$
  $$\left(\hat{\mathcal{C}}_{v_c}[\![body(\hat{\phi}_c(t_1))]\!] \cup \bigcup_{y \in dom(state(\hat{\phi}_c(t_1)))} \hat{\mathcal{C}}_{v_c}[\![state(\hat{\phi}_c(t_1))(y)]\!]\right)$$
  $$\cap$$
  $$\left(\hat{\mathcal{C}}_{v_c}[\![body(\hat{\phi}_c(t_2))]\!] \cup \bigcup_{y \in dom(state(\hat{\phi}_c(t_2)))} \hat{\mathcal{C}}_{v_c}[\![state(\hat{\phi}_c(t_2))(y)]\!]\right)$$
  $$= \emptyset.$$

- The set of called procedure are disjoint

  $$\forall t_1, t_2 \in dom(\hat{\phi}_c), t_1 \neq t_2 \Rightarrow dom(state(\hat{\phi}_c(t_1))) \cap dom(state(\hat{\phi}_c(t_2))) = \emptyset.$$

  This also implies that the set of contracts are disjoint.

- To ensure having no duplicated labels in the resulting self-composed program, we assume having no duplicated labels between the programs linked by the relational property:

  $$\forall t_1, t_2 \in dom(\hat{\phi}_c), t_1 \neq t_2 \Rightarrow \hat{\mathcal{U}}[\![body(\hat{\phi}_c(t_1))]\!] \cap \hat{\mathcal{U}}[\![body(\hat{\phi}_c(t_2))]\!] = \emptyset.$$

  where function $\hat{\mathcal{U}}$ (defined in Appendix A.2) takes a command and returns all labels used in the command, or $\perp$ if there are duplicated labels. As the program in $\hat{\phi}_c$ are assumed well defined, the case of $\perp$ has no need to be treated.

  Labels don't have to be renamed in procedures because labels are isolated to the body of the procedure (according to the semantics of a well-defined program), *i.e.* we cannot refer

to labels in the body of a procedure, and we cannot refer to labels outside the body of a procedure. It is therefore not necessary to guarantee the separation between the labels used in the different procedures in the context of self-composition.

Example 6.2. We consider the following relational property

$$\{at(x, Pre)\langle t_1\rangle = at(x, Pre)\langle t_2\rangle\}$$
$$\begin{array}{lll} l_1 : x := x + 5; & & l_1 : x := x + 5; \\ l_2 : \textbf{call}(y); & \langle t_1\rangle \sim & l_2 : \textbf{call}(y); \quad \langle t_2\rangle \\ l_3 : x := x + 6 & & l_3 : x := x + 6 \end{array}$$
$$\{at(x, Post)\langle t_1\rangle = at(x, Post)\langle t_2\rangle\}$$

A relational extended execution environment $\hat{\phi}_c$, where the memory states for commands are defined, for each tag, by:

$$state(\hat{\phi}_c(t_1)) = \{y \to l : x := x + 1\},$$
$$state(\hat{\phi}_c(t_2)) = \{y \to l : x := x + 1\},$$

and an environment $\hat{\phi}_a$ defined by:

$$\left\{ \begin{array}{l} t_1 \to \left\{ y \to \left(\begin{array}{l} true, \\ at(x, Pre) + 1 = at(x, Post), \\ \{x\} \end{array} \right) \right\}, \\ t_2 \to \left\{ y \to \left(\begin{array}{l} true, \\ at(x, Pre) + 1 = at(x, Post), \\ \{x\} \end{array} \right) \right\} \end{array} \right\}$$

That is for both tags, the program associated to program name y returns a state where the value of location x is equal to the value of location x before the procedure call plus 1. Moreover, the program only assigns location x. It is not hard to see that the procedure in $\hat{\phi}_c$ satisfies the contract in $\hat{\phi}_a$.

We can note that the programs and procedures in the relational extended execution environment $\hat{\phi}_c$ shares locations, program names and labels. Thus, the requirements of Self-Composition are not fulfilled. In Section 4.3 we have seen that renaming the locations is a solution to get an associated relational state environment ϕ with disjoint memory states for each tag. Moreover, renaming the command names ensures disjoint memory states for commands. In the same spirit we can simply rename the labels to ensure unique labels between commands.

If we apply those renamings and Self-Composition to the relational property, we get the following Hoare Triple:

$$\{at(x_{t_1}, Pre) = at(x_{t_2}, Pre)\} \begin{array}{l} l_{1\_t_1} : x_{t_1} := x_{t_1} + 5; \\ l_{2\_t_1} : \textbf{call}(y_{t_1}); \\ l_{3\_t_1} : x_{t_1} := x_{t_1} + 6; \\ l_{1\_t_2} : x_{t_2} := x_{t_2} + 5; \\ l_{2\_t_2} : \textbf{call}(y_{t_2}); \\ l_{3\_t_2} : x_{t_2} := x_{t_2} + 6 \end{array} \{at(x_{t_1}, Post) = at(x_{t_2}, Post)\}$$

where all locations, labels and program names have been renamed and the two programs are composed in sequence.

The renamings are also applied to the memory state for commands in the relational extended execution environment $\hat{\phi}_c$

$$state(\hat{\phi}_c(t_1)) = \{y_{t_1} \rightarrow l_{t_1} : x_{t_1} := x_{t_1} + 1\},$$
$$state(\hat{\phi}_c(t_2)) = \{y_{t_2} \rightarrow l_{t_2} : x_{t_2} := x_{t_2} + 1\},$$

and in environment $\hat{\phi}_a$

$$\left\{ \begin{array}{l} t_1 \rightarrow \left\{ y_{t_1} \rightarrow \left(\begin{array}{l} true, \\ at(x_{t_1}, Pre) + 1 = at(x_{t_1}, Post), \\ \{x_{t_1}\} \end{array} \right) \right\}, \\ t_2 \rightarrow \left\{ y_{t_2} \rightarrow \left(\begin{array}{l} true, \\ at(x_{t_2}, Pre) + 1 = at(x_{t_2}, Post), \\ \{x_{t_2}\} \end{array} \right) \right\} \end{array} \right\}$$

To prove the validity of the previous triple we can use function $\hat{\mathcal{VC}}_h$:

$$\hat{\mathcal{VC}}_h(\varsigma_s, \left(\begin{array}{l} at(x_{t_1}, Pre) = at(x_{t_2}, Pre), \\ at(x_{t_1}, Post) = at(x_{t_2}, Post) \end{array} \right), \hat{\phi}_a(t_1) \cup \hat{\phi}_a(t_2))$$

In practice, the renaming must only be applied to environment $\hat{\phi}_a$. Applying the renaming to the environment $\hat{\phi}_c$ is not required as it is not used by function $\hat{\mathcal{VC}}_h$ which takes as a parameter the self-composed programs and an environment of contract, mapping programs names to contracts. Moreover, only procedures called by the self-composed programs should be subject to renaming, as the contract of the other procedures are not used by function $\hat{\mathcal{VC}}_h$. The only case were applying the renaming to $\hat{\phi}_c$ is required is in the case where procedure calls are inlined.

Assuming that x_{t_1} is mapped to index 1 and x_{t_2} to index 2, we get the following VC:

$$\begin{array}{l} m_{Pre}[1] = m_{Pre}[2] \wedge \\ m_{l_{2\_t_1}} = m_{Pre}[1 \leftarrow m_{Pre}[1] + 5] \wedge \\ m_{l_{3\_t_1}}[1] = m_{l_{2\_t_1}}[1] + 1 \wedge m_{l_{3\_t_1}} = m_{l_{2\_t_1}}[1 \leftarrow i_1] \wedge \\ m_{l_{1\_t_2}} = m_{l_{3\_t_1}}[1 \leftarrow m_{l_{3\_t_1}}[1] + 6] \wedge \\ m_{l_{2\_t_2}} = m_{l_{1\_t_2}}[2 \leftarrow m_{l_{1\_t_2}}[2] + 5] \wedge \\ m_{l_{3\_t_2}}[2] = m_{l_{2\_t_2}}[2] + 1 \wedge m_{l_{3\_t_2}} = m_{l_{2\_t_2}}[2 \leftarrow i_2] \wedge \\ m_{Post} = m_{l_{3\_t_2}}[2 \leftarrow m_{l_{3\_t_2}}[2] + 6] \Rightarrow \\ m_{Post}[1] = m_{Post}[2] \end{array}$$

We have in blue the part corresponding to the program associated to tag t_1, and in red the part corresponding to the program associated to tag t_2. By using axioms Q-NO-UPDATE and Q-UPDATE, we get the following valid formula:

$$\begin{array}{l} m_{Pre}[1] = m_{Pre}[2] \Rightarrow \\ m_{Pre}[1] + 5 + 1 + 6 = m_{Pre}[2] + 5 + 1 + 6 \end{array}$$

Note that we omit some VC's corresponding to the proof that the preconditions of the procedure calls hold. Those VC's are trivial since the preconditions correspond to $true$.

For relational properties linking single bodies of procedures whose contracts have been verified, *i.e* relational properties of the form:

$$\{\tilde{\beta}_1\}... \sim state(\hat{\phi}_c(t_i))(y)\langle t_i\rangle \sim ...\{\tilde{\beta}_2\}$$

where the contract $\hat{\phi}_a(t_i)(y)$ is valid, it is not required to prove the formulas corresponding to the fact that the precondition of called procedures hold, the invariants (when there are loops inside the bodies) hold, assertions hold in the case of the linked procedure body $state(\hat{\phi}_c(t_i))(y)$. Those formulas have already be verified during the verification of the contracts $\hat{\phi}_a(t_i)(y)$. This became clear when combining the previous relational property and the contract of the linked procedure ($\hat{\phi}_a(t_i)(y) = (\beta_{pre}, \beta_{post}, \_)$) into an equivalent relational property:

$$\{\tilde{\beta}_1 \wedge \beta_{pre}\langle t_i\rangle\}... \sim \hat{\phi}_c(t_i)(y)\langle t_i\rangle \sim ...\{\tilde{\beta}_2 \wedge \beta_{post}\langle t_i\rangle\}$$

We know that for precondition β_{pre}, all preconditions of called procedures, loop invariants and assertions in $\hat{\phi}_c(t_i)(y)$ hold. Moreover, β_{pre} can be assumed in the relational precondition, as during the verification of procedure contract for tag t_i it is verified to hold before each call to y.

Unfortunately, the current formalization of function $\hat{\mathcal{VC}}_h$ is not suitable for filtering formulas. This optimization will therefore not be highlighted in the following and we require that all generated formulas must be proven valid. This implies that the relational precondition implies the precondition of linked single procedures.

6.3 Axiomatisation of Relational Properties

In the previous Section 6.2 we have shown how to use Self-Composition to prove relational properties in the context of the R-WHILE* language using standard contracts. However, as already mentioned for Section 4.3, using the contract of procedures in the proof of relational properties is not as powerful as using relational properties.

Example 6.3. We consider the following relational property using the syntax of Section 4.1

$$\{x\langle t_1\rangle = x\langle t_2\rangle\} \begin{array}{l} x := x + 5; \\ \textbf{call}(y); \\ x := x + 6 \end{array} \langle t_1\rangle \sim \begin{array}{l} x := x + 5; \\ \textbf{call}(y); \\ x := x + 6 \end{array} \langle t_2\rangle\{x\langle t_1\rangle = x\langle t_2\rangle\}$$

and a relational extended execution environment ϕ_c, where the memory states for command are defined, for each tag, by:

$$state(\phi_c(t_1)) = \{y \to x := x + 1\},$$

$$state(\phi_c(t_2)) = \{y \to x := x + 1\}.$$

It would be convenient to use a relational property stating that for two calls to procedure y, if the value in location x is the same in both tags before execution, then the value in location x is the same in both tags after execution.

$$\{x\langle t_1\rangle = x\langle t_2\rangle\} state(\phi_c(t_1)(y))\langle t_1\rangle \sim state(\phi_c(t_2)(y))\langle t_2\rangle \{x\langle t_1\rangle = x\langle t_2\rangle\}$$

By contrast, finding a standard contract for the call of y such that the property can be proven appears non-trivial, even if we have shown in the previous section that with more expressive boolean expressions it is possible.

Therefore, we propose in this section an approach to prove relational properties using (other) relational properties, in the same manner as we use contracts of procedure for Hoare Triples. First, we propose in Section 6.3.1 some rules, extending the proof system proposed in Relational Hoare Logic shown in Section 4.2, as an intuition. We then propose in Section 6.3.2 an encoding of relational properties in First-Order Language MFOL such that they can be used in the context of Self-Composition in the R-WHILE* language.

6.3.1 Using Relational Properties in Relational Hoare Logic

First, we define the environment of relational properties $\tilde{\Xi}$ (equivalent to Ξ for Hoare Triple):

$$\tilde{\Xi} = \mathcal{P}(\mathbb{Y} \times \mathbb{T}) \rightharpoonup \tilde{\mathbb{E}}_b \times \tilde{\mathbb{E}}_b,$$

and use metavariables $\tilde{\xi}, \tilde{\xi}_0, \tilde{\xi}_1, ...$ to range over $\tilde{\Xi}$.

For a given set of pairs, composed of a program name and a tag, environment $\tilde{\Xi}$ associates a pair of relational boolean expressions, building a relational contract. The meaning of a relational contract is similar to that of a procedure contract; for a given set of pairs u, composed of program names and tags, if $\phi \models fst(\tilde{\xi}(u))$ is verified, then for each pair $(y, t) \in u$, calling the procedure y on the state associated to the tag t in ϕ results in a relational state environment ϕ', such that $\phi' \models snd(\tilde{\xi}(u))$ is verified.

We assume the relational boolean expressions composing the relational contracts satisfy the same properties as those presented in Section 4.1 for relational property.

Example 6.4. If we consider an environment $\tilde{\xi}$ defined by,

$$\{\{(y, t_1), (y, t_2)\} \to (x\langle t_1\rangle = x\langle t_2\rangle, x\langle t_1\rangle = x\langle t_2\rangle)\},$$

the equivalent set of relational properties would be

$$\left\{ \begin{array}{c} \{x\langle t_1\rangle = x\langle t_2\rangle\} \\ state(\phi_c(t_1))(y)\langle t_1\rangle \sim state(\phi_c(t_2))(y)\langle t_2\rangle \\ \{x\langle t_1\rangle = x\langle t_2\rangle\} \end{array} \right\}$$

Using relational contracts:

Using environment $\tilde{\Xi}$, we can extend rules from the proof system for Hoare Triple, shown in Section 3.3, to use relational properties in Relational Hoare Logic. The approach is the same as in Section 4.2 to build rules for Relational Hoare Logic.

First, we can define rule R-RCALL, an extension of rule CALL for using relational contracts:

$$\frac{\tilde{\xi}((y_1,t_1),(y_2,t_2)) = (\tilde{b}_1,\tilde{b}_2)}{\tilde{\xi} \vdash \{\tilde{b}_1\}\textbf{call}(y_1)\langle t_1\rangle \sim \textbf{call}(y_2)\langle t_2\rangle\{\tilde{b}_2\}} \quad \text{(R-RCALL)}$$

The rule means that if we have in the assumption the relational property about the two procedure calls y_1 and y_2, the property is valid.

We also define rule R-RRECURSION an extension of rule RECURSION

$$\frac{\begin{array}{c}\tilde{\xi} \vdash \{\tilde{b}_1\}c_1\langle t_1\rangle \sim c_2\langle t_2\rangle\{\tilde{b}_2\} \\ \forall a = \{(y_1,t_1),(y_2,t_2)\} \in dom(\tilde{\xi}). \\ \tilde{\xi} \vdash \{fst(\tilde{\xi}(a))\}state(\phi_c(t_1))(y_1) \sim state(\phi_c(t_2))(y_2)\{snd(\tilde{\xi}(a))\}\end{array}}{\vdash \{\tilde{b}_1\}c_1\langle t_1\rangle \sim c_2\langle t_2\rangle\{\tilde{b}_2\}} \quad \text{(R-RRECURSION)}$$

meaning that $\vdash \{\tilde{b}_1\}c_1\langle t_1\rangle \sim c_2\langle t_2\rangle\{\tilde{b}_2\}$ is valid, if for a set of assumptions (relational procedure contracts) $\tilde{\xi}$, there is a derivation such that $\tilde{\xi} \vdash \{\tilde{b}_1\}c_1\langle t_1\rangle \sim c_2\langle t_2\rangle\{\tilde{b}_2\}$ and each relational procedure contract in $\tilde{\xi}$ is a conclusion of Relational Hoare Logic, extended with rule R-RCALL.

Example 6.5. Using rules R-RCALL and R-RRECURSION Example 6.3 can be proven using relational properties.

$$\vdash \{x\langle t_1\rangle = x\langle t_2\rangle\} \begin{array}{l} x := x+5; \\ \textbf{call}(y); \\ x := x+6 \end{array} \langle t_1\rangle \sim \begin{array}{l} x := x+5; \\ \textbf{call}(y); \\ x := x+6 \end{array} \langle t_2\rangle\{x\langle t_1\rangle = x\langle t_2\rangle\}$$

First, we use rule R-RRECURSION to get the following sub-proofs:

$$\tilde{\xi} \vdash \{x\langle t_1\rangle = x\langle t_2\rangle\} \begin{array}{l} x := x+5; \\ \textbf{call}(y); \\ x := x+6 \end{array} \langle t_1\rangle \sim \begin{array}{l} x := x+5; \\ \textbf{call}(y); \\ x := x+6 \end{array} \langle t_2\rangle\{x\langle t_1\rangle = x\langle t_2\rangle\}, \quad \text{(sub-proof 1)}$$

and

$$\tilde{\xi} \vdash \{x\langle t_1\rangle = x\langle t_2\rangle\}state(\phi_c(t_1))(y)\langle t_1\rangle \sim state(\phi_c(t_2))(y)\langle t_2\rangle\{x\langle t_1\rangle = x\langle t_2\rangle\}, \quad \text{(sub-proof 2)}$$

and $\tilde{\xi}$ is chosen as follows:

$$\{\{(y,t_1),(y,t_2)\} \to (x\langle t_1\rangle = x\langle t_2\rangle, x\langle t_1\rangle = x\langle t_2\rangle)\}\,.$$

Proof of sub-proof 1: Using rule for sequence R-SEQUENCE we get the following sub-proofs:

$$\tilde{\xi} \vdash \{x\langle t_1\rangle = x\langle t_2\rangle\}x := x+5\langle t_1\rangle \sim x := x+5\langle t_2\rangle\{x\langle t_1\rangle = x\langle t_2\rangle\} \quad \text{(sub-proof 1.1)}$$

$$\tilde{\xi} \vdash \{x\langle t_1\rangle = x\langle t_2\rangle\}\mathbf{call}(y)\langle t_1\rangle \sim \mathbf{call}(y)\langle t_2\rangle\{x\langle t_1\rangle = x\langle t_2\rangle\} \qquad \text{(sub-proof 1.2)}$$

$$\tilde{\xi} \vdash \{x\langle t_1\rangle = x\langle t_2\rangle\}x := x + 6\langle t_1\rangle \sim x := x + 6\langle t_2\rangle\{x\langle t_1\rangle = x\langle t_2\rangle\} \qquad \text{(sub-proof 1.3)}$$

Using rule R-CONSEQUENCE and axiom R-ASSIGN, we prove sub-proof 1.1 and sub-proof 1.3. Using rule R-RCALL, we prove sub-proof 1.2.

Proof of sub-proof 2: Using rule R-CONSEQUENCE and axiom R-ASSIGN.

Combining relational contracts:

Until now we have always considered relational properties linking 2 programs. If we now consider the case where more than two properties are linked, it may be interesting to combine properties with less or the same number of programs. Therefore, we define rule R-RCOMBINE, an extension of rule R-COMBINE.

$$\frac{\begin{array}{c}\tilde{\xi} \vdash \{\tilde{b}_{11}\}\mathbf{call}(y_{i_1})\langle t_{i_1}\rangle \sim ... \sim \mathbf{call}(y_{j_1})\langle t_{j_1}\rangle\{\tilde{b}_{21}\} \\ ... \quad \tilde{\xi} \vdash \{\tilde{b}_{1k}\}\mathbf{call}(y_{i_k})\langle t_{i_k}\rangle \sim ... \sim \mathbf{call}(y_{j_k})\langle t_{j_k}\rangle\{\tilde{b}_{2k}\} \\ \{(y_{i_1}, t_{i_1}), ..., (y_{j_1}, t_{j_1})\} \subseteq \{(y_1, t_1), ..., (y_n, t_n)\} \\ ... \quad \{(y_{i_k}, t_{i_k}), ..., (y_{j_k}, t_{j_1})\} \subseteq \{(y_1, t_1), ..., (y_n, t_n)\} \\ \tilde{\mathcal{C}}_{t_b}[\![b_{11}]\!] \subseteq \{t_{i_1}, ..., t_{j_1}\} \qquad \tilde{\mathcal{C}}_{t_b}[\![b_{21}]\!] \subseteq \{t_{i_1}, ..., t_{j_1}\} \\ ... \quad \tilde{\mathcal{C}}_{t_b}[\![b_{1k}]\!] \subseteq \{t_{i_k}, ..., t_{j_k}\} \qquad \tilde{\mathcal{C}}_{t_b}[\![b_{2k}]\!] \subseteq \{t_{i_k}, ..., t_{j_k}\}\end{array}}{\tilde{\xi} \vdash \{\tilde{b}_{11} \wedge ... \wedge \tilde{b}_{1k}\}\mathbf{call}(y_1)\langle t_1\rangle \sim ... \sim \mathbf{call}(y_n)\langle t_n\rangle\{\tilde{b}_{21} \wedge ... \wedge \tilde{b}_{2k}\}} \qquad \text{(R-RCOMBINE)}$$

The rule means that we can prove a relational property linking n procedure calls by proving k relational properties, each linking a set of procedure calls that is a subset of the set composed of the n initial procedure calls. Note that this rule can be generalized to any command.

Example 6.6. We consider the following relational property linking three procedure calls

$$\tilde{\xi} \vdash \left\{ \begin{array}{c} x\langle t_1\rangle = x\langle t_2\rangle \wedge \\ x\langle t_2\rangle = x\langle t_3\rangle \end{array} \right\} \mathbf{call}(y)\langle t_1\rangle \sim \mathbf{call}(y)\langle t_2\rangle \sim \mathbf{call}(y)\langle t_3\rangle \left\{ \begin{array}{c} x\langle t_1\rangle = x\langle t_2\rangle \wedge \\ x\langle t_2\rangle = x\langle t_3\rangle \end{array} \right\}$$

and an environment $\tilde{\xi}$ defined by

$$\left\{ \begin{array}{l} \{(y, t_1), (y, t_2)\} \to (x\langle t_1\rangle = x\langle t_2\rangle, x\langle t_1\rangle = x\langle t_2\rangle), \\ \{(y, t_2), (y, t_3)\} \to (x\langle t_2\rangle = x\langle t_3\rangle, x\langle t_2\rangle = x\langle t_3\rangle) \end{array} \right\}$$

Using rule R-RCOMBINE we get the following sub-proofs:

$$\tilde{\xi} \vdash \{x\langle t_1\rangle = x\langle t_2\rangle\}\mathbf{call}(y)\langle t_1\rangle \sim \mathbf{call}(y)\langle t_2\rangle\{x\langle t_1\rangle = x\langle t_2\rangle\},$$

and

$$\tilde{\xi} \vdash \{x\langle t_2\rangle = x\langle t_3\rangle\}\mathbf{call}(y)\langle t_2\rangle \sim \mathbf{call}(y)\langle t_3\rangle\{x\langle t_2\rangle = x\langle t_3\rangle\}.$$

Using rule R-RCALL we can prove the sub-proofs.

Using relational contracts from different tags:

Finally, when the relational execution environment ϕ_c shares a set of equal memory states for programs:

$$\exists t_1, t_2 \in dom(\hat{\phi}_c).state(\hat{\phi}_c(t_1)) = state(\hat{\phi}_c(t_2)),$$

we can define rule R-TAG-L allowing some flexibility in the way we use tags:

$$\frac{\tilde{\xi} \vdash \{\tilde{\mathcal{R}}_{t_b}[\![\tilde{b}_1]\!](t_1, t_k)\}\mathbf{call}(y_1)\langle t_k\rangle \sim \mathbf{call}(y_2)\langle t_2\rangle\{\tilde{\mathcal{R}}_{t_b}[\![\tilde{b}_2]\!](t_1, t_k)\} \quad t_k \neq t_2 \quad state(\phi_c(t_k)) = state(\phi_c(t_1))}{\tilde{\xi} \vdash \{\tilde{b}_1\}\mathbf{call}(y_1)\langle t_1\rangle \sim \mathbf{call}(y_2)\langle t_2\rangle\{\tilde{b}_2\}} \text{ (R-TAG-L)}$$

The rule mean that for two tags (t_1 and t_k), if their environment for command are equal, proving a relational property linking a procedure for one tag (t_1) is equivalent to prove the relational property for the same procedure for the other tag (t_k). Notice that the symmetric version (i.e. renaming t_2 as t_k) of the rules (called R-TAG-R) is also true.

Example 6.7. We consider the following relational property linking three procedure calls

$$\tilde{\xi} \vdash \left\{ \begin{array}{c} x\langle t_1\rangle = x\langle t_2\rangle \wedge \\ x\langle t_2\rangle = x\langle t_3\rangle \end{array} \right\} \mathbf{call}(y)\langle t_1\rangle \sim \mathbf{call}(y)\langle t_2\rangle \sim \mathbf{call}(y)\langle t_3\rangle \left\{ \begin{array}{c} x\langle t_1\rangle = x\langle t_2\rangle \wedge \\ x\langle t_2\rangle = x\langle t_3\rangle \end{array} \right\}$$

and an environment $\tilde{\xi}$ defined by

$$\left\{ \ \{(y, t_1), (y, t_2)\} \to (x\langle t_1\rangle = x\langle t_2\rangle, x\langle t_1\rangle = x\langle t_2\rangle)) \ \right\}$$

Using rule R-RCOMBINE we get the following sub-proofs:

$$\tilde{\xi} \vdash \{x\langle t_1\rangle = x\langle t_2\rangle\}\mathbf{call}(y)\langle t_1\rangle \sim \mathbf{call}(y)\langle t_2\rangle\{x\langle t_1\rangle = x\langle t_2\rangle\},$$

and

$$\tilde{\xi} \vdash \{x\langle t_2\rangle = x\langle t_3\rangle\}\mathbf{call}(y)\langle t_2\rangle \sim \mathbf{call}(y)\langle t_3\rangle\{x\langle t_2\rangle = x\langle t_3\rangle\}.$$

If we have:

$$state(\hat{\phi}_c(t_1)) = state(\hat{\phi}_c(t_2)) = state(\hat{\phi}_c(t_3)),$$

using rule R-TAG-L and R-TAG-R on the second sub-proof, we get:

$$\tilde{\xi} \vdash \{x\langle t_1\rangle = x\langle t_2\rangle\}\mathbf{call}(y)\langle t_1\rangle \sim \mathbf{call}(y)\langle t_2\rangle\{x\langle t_1\rangle = x\langle t_2\rangle\},$$

$$\tilde{\xi} \vdash \{x\langle t_1\rangle = x\langle t_2\rangle\}\mathbf{call}(y)\langle t_1\rangle \sim \mathbf{call}(y)\langle t_2\rangle\{x\langle t_1\rangle = x\langle t_2\rangle\}.$$

Using rule R-RCALL we can prove the sub-proofs using only one relational contract. Without rule R-TAG-L and R-TAG-R, we would need two relational contracts, like in Example 6.6.

With rules R-TAG-L, R-RCOMBINE, R-RRECURSION and R-RCALL it becomes possible to use relational contracts as part of the verification of relational property with Relational Hoare Logic. In the following section we show how to get the same results in case of the R-WHILE* language and Self-Composition.

6.3.2 Using Relational Properties with Self-Composition

First, we define the environment of relational properties (equivalent to $\hat{\Xi}$ for Hoare Triples):

$$\tilde{\hat{\Xi}} = \mathcal{P}(\mathbb{Y} \times \mathbb{T}) \rightharpoonup \tilde{\hat{\mathbb{E}}}_b \times \tilde{\hat{\mathbb{E}}}_b$$

As previously, for a given set of pairs, composed of a program name and a tag, environment $\tilde{\hat{\Xi}}$ associate a pair of relational extended boolean expressions, corresponding to a relational contract. The relational extended boolean expressions composing the relational contract satisfy the same properties as those presented in Section 6.1, with the exception that the first relational boolean expression only uses label Pre, and the second relational boolean expression only uses label Pre and $Post$. We use metavariables $\tilde{\hat{\xi}}, \tilde{\hat{\xi}}_0, \tilde{\hat{\xi}}_1, ...$ to range over $\tilde{\hat{\Xi}}$.

Using relational properties in the context of verification condition generation could be performed in a similar way as for standard contracts *i.e.* we replace a set of procedure calls by the corresponding *relational* postcondition and prove that the *relational* precondition holds. However, such an approach can be hard to apply, since finding the right combination (rule R-RCOMBINE) of procedure calls that must be replaced by the corresponding property is not straightforward. Moreover, the corresponding precondition must hold.

Therefore, we use an alternative method that is commonly used in modular deductive verification. The solution consists in using predicates as connectors between an axiomatization and the program, like in the examples shown on Figure 1.1 and Section 2.2.2. It then is left to the function smt to choose the right combination and instantiation of the right formulas.

Similar work can be found in [LM08] for the axiomatization of methods with equivalent results in the context of Spec# [BLS05], in [HKLR13] for axiomatization of equivalent programs in Boogie, and in [DM06] for the axiomatization of pure methods in the Java Modeling Language [LRL$^+$00].

Axiomatization

The first step of the approach consists in translating the relational property into a formula of $\mathbb{Q}$. We propose to model the locations by integer variables and procedure calls by predicates. The predicate takes as a parameter all the variables corresponding to the locations that belong to the tag of the corresponding procedure call.

Example 6.8. If we consider a relational contract $\tilde{\hat{\xi}}$ defined by

$$\left\{ \{(y, t_1), (y, t_2)\} \rightarrow \begin{pmatrix} at(x, Pre)\langle t_1 \rangle = at(x, Pre)\langle t_2 \rangle, \\ at(x, Post)\langle t_1 \rangle = at(x, Post)\langle t_2 \rangle \end{pmatrix} \right\}$$

we can translate the contract for $\{(y, t_1), (y, t_2)\}$ into following formula:

$$\begin{gathered} \forall v_{t_1 Pre}, v_{t_1 Post}, v_{t_2 Pre}, v_{t_2 Post}, v_{trace_1 id}, v_{t_2 id} : \text{nat}, \\ (p_{y\_t_1}(v_{t_1 Pre}, v_{t_1 Post}) \wedge p_{y\_t_2}(v_{t_2 Pre}, v_{t_2 Post}) \wedge v_{t_1 Pre} = v_{t_2 Pre}) \Rightarrow \\ v_{t_1 Post} = v_{t_2 Post} \end{gathered}$$

Variables $v_{t_1 Pre}$ and $v_{t_1 Post}$ represent location x for tag t_1 respectively at labels Pre and $Post$. Variables $v_{t_2 Pre}$ and $v_{t_2 Post}$ represent location x for tag t_2 respectively at labels Pre and $Post$. Predicates $p_{y\_t_1}$ and $p_{y\_t_2}$ represent respectively a procedure call of y for tags t_1 and t_2.

If for an environment $\hat{\phi}_c$, all memory states for programs $\hat{\psi}$ are the same (rule R-TAG-L and R-TAG-R):

$$\forall t_1, t_2 \in dom(\hat{\phi}_c).state(\hat{\phi}_c(t_1)) = state(\hat{\phi}_c(t_2))$$

we can refine the previous translation by using only one predicate per program name

$$\begin{array}{c} \forall v_{t_1 Pre}, v_{t_1 Post}, v_{t_2 Pre}, v_{t_2 Post}, v_{t_1 id}, v_{t_2 id} : \text{nat}, \\ (p_y(v_{t_1 Pre}, v_{t_1 Post}, v_{t_1 id}) \wedge p_y(v_{t_2 Pre}, v_{t_2 Post}, v_{t_2 id}) \wedge \\ v_{t_1 id} \neq v_{t_2 id} \wedge \\ v_{t_1 Pre} = v_{t_2 Pre}) \Rightarrow v_{t_1 Post} = v_{t_2 Post} \end{array} \quad \text{(AXIOM-RELA)}$$

To model the fact that we have distinct procedure calls, each shared predicate takes as parameter an additional identifier, which is instantiated by distinct variables (hence the $v_{t_1 id} \neq v_{t_2 id}$ in AXIOM-RELA) for each distinct call in the relational property.

In general, only one set of procedures is defined in a complete program. The case where different sets of procedures are defined can be transformed into a single set of procedure by performing a union of the memory states for commands. Therefore, we assume in the following that for a relational extended execution environment $\hat{\phi}_c$, all memory states for commands are the same for all defined tags:

$$\forall t_1, t_2 \in dom(\hat{\phi}_c).state(\hat{\phi}_c(t_1)) = state(\hat{\phi}_c(t_2)) \quad \text{(HYP-PRO-STATE)}$$

Connection between axiomatization and the procedures

The second step consists in using the generated formula AXIOM-RELA as an axiom. Since each procedure call is modeled by a predicate, we can use those predicates to link the formula to the procedure calls by assuming it after the call. Therefore, we define a new command **assume**(β), adding to the set of axioms an extended boolean expression β. The semantics is as follows.

Definition 6.5. Evaluation function

$$\hat{\xi}_c : \hat{\mathbb{C}} \to ((\mathbb{N} \times \Psi \times \mathcal{P}(\mathbb{Q}) \times (\Sigma_\Omega \times \Lambda)) \to (\Sigma_\Omega \times \Lambda \times \mathcal{P}(\mathbb{Q})))$$

for extended commands $\hat{\xi}_c$ (case of command **assume**), is defined by:

$$\hat{\xi}_c[\![l : \textbf{assume}(\beta)]\!](n, (\psi, u_\mathbb{Q}))\sigma\lambda =$$
$$\begin{cases} (\Omega_\perp, \emptyset, \emptyset) & \text{if } \hat{\xi}_b[\![\beta]\!](\lambda[l \leftarrow \sigma]) = \perp \\ (\sigma, \lambda[l \leftarrow \sigma], \{\hat{\xi}_b[\![\beta]\!](\lambda[l \leftarrow \sigma])\} \cup u_\mathbb{Q}) & \text{otherwise} \end{cases}$$

The definition is similar to the case of command **assert**. The difference is that the boolean expression is not checked valid but the associated formula is added to the set of axioms $u_\mathbb{Q}$,

if different from $\perp$. When the translation of the boolean expression returns $\perp$, the evaluation function return $\Omega_\perp$ since $\perp$ is not part of the syntax of MFOL. Notice that the signature of function $\hat{\xi}_c$ has changed; the set of axiom $u_{\mathbb{Q}}$ can be changed by the commands and must be returned. This modification implies also a modification in the definition of $\tilde{\hat{\xi}}_c$, which is similar.

Example 6.9. Using command **assume**, we can connect the program to the axiomatized relational property AXIOM-RELA as follows:

$$\{at(x, Pre)\langle t_1\rangle = at(x, Pre)\langle t_2\rangle\}$$
$$\begin{array}{l} l_1 : x := x + 5; \\ l_2 : \textbf{call}(y); \\ l_n : \textbf{assume}(p_y(at(x, l_2), at(x, l_n), 1)); \\ l_3 : x := x + 6 \end{array} \langle t_1\rangle \sim \begin{array}{l} l_1 : x := x + 5; \\ l_2 : \textbf{call}(y); \\ l_n : \textbf{assume}(p_y(at(x, l_2), at(x, l_n), 2)); \\ l_3 : x := x + 6 \end{array} \langle t_2\rangle$$
$$\{at(x, Post)\langle t_1\rangle = at(x, Post)\langle t_2\rangle\}$$

Note that the two occurrences of predicate p_y take two different identifiers.

As for the evaluation function $\hat{\xi}_c$ for command **assert**, the definition of function $\hat{\mathcal{T}}_c$ for command **assume** is similar to the one for command **assert**. The difference consists in the absence of a verification condition for verifying that the boolean expression holds.

Definition 6.6. Function $\hat{\mathcal{T}}_c : \hat{\mathbb{C}} \to (\Xi \times V \to V)$, translating a command $\hat{\mathbb{C}}$ (case of command **assume**) into an formula of $\mathbb{Q}$, is defined by:

$$\hat{\mathcal{T}}_c[\![l : \textbf{assume}(\beta)]\!](\xi, (q, u_{\mathbb{Q}}, m, (\theta, \delta))) = (q \wedge q', u_{\mathbb{Q}}, m, s)$$
$$\text{where } (q', s) = \hat{\mathcal{T}}_b[\![\beta]\!](\theta[l \leftarrow m], \delta)$$

Example 6.10. If we apply Self-Composition as shown in Section 6.2 on the relational property of Example 6.9, we get the following Hoare Triple:

$$\{at(x_{t_1}, Pre) = at(x_{t_2}, Pre)\}$$
$$\begin{array}{l} l_{1\_t_1} : x_{t_1} := x_{t_1} + 5; \\ l_{2\_t_1} : \textbf{call}(y_{t_1}); \\ l_{n\_t_1} : \textbf{assume}(p_y(at(x_{t_1}, l_{2\_t_1}), at(x_{t_1}, l_{n\_t_1}), 1)); \\ l_{3\_t_1} : x_{t_1} := x_{t_1} + 6; \\ l_{1\_t_2} : x_{t_2} := x_{t_2} + 5; \\ l_{2\_t_2} : \textbf{call}(y_{t_2}); \\ l_{n\_t_2} : \textbf{assume}(p_y(at(x_{t_2}, l_{2\_t_2}), at(x_{t_2}, l_{n\_t_2}), 2)); \\ l_{3\_t_2} : x_{t_2} := x_{t_2} + 6 \end{array}$$
$$\{at(x_{t_1}, Post) = at(x_{t_2}, Post)\}$$

Using function $\hat{\mathcal{VC}}_h$ we get the following verification condition:

$$
\begin{array}{l}
m_{Pre}[1] = m_{Pre}[2] \wedge \\
m_{l_{2\_t_1}} = m_{Pre}[1 \leftarrow m_{Pre}[1] + 5] \wedge \\
p_y(m_{l_{2\_t_1}}[1], m_{l_{n\_t_1}}[1], 1) \wedge m_{l_{n\_t_1}} = m_{l_{2\_t_1}}[1 \leftarrow i_1] \wedge \\
m_{l_{1\_t_2}} = m_{l_{n\_t_1}}[1 \leftarrow m_{l_{n\_t_1}}[1] + 6] \wedge \\
m_{l_{2\_t_2}} = m_{l_{1\_t_2}}[2 \leftarrow m_{l_{1\_t_2}}[2] + 5] \wedge \\
p_y(m_{l_{2\_t_2}}[2], m_{l_{n\_t_2}}[2], 2) \wedge m_{l_{n\_t_2}} = m_{l_{2\_t_2}}[2 \leftarrow i_2] \wedge \\
m_{Post} = m_{l_{n\_t_2}}[2 \leftarrow m_{l_{n\_t_2}}[2] + 6] \Rightarrow \\
m_{Post}[1] = m_{Post}[2]
\end{array}
$$

Notice that we have assumed that the contract corresponding to program names y_{t_1} and y_{t_2} is composed only of the frame rule. Thus, in addition to the predicates, we have formula $m_{l_{n\_t_1}} = m_{l_{2\_t_1}}[1 \leftarrow i_1]$ and $m_{l_{n\_t_2}} = m_{l_{2\_t_2}}[2 \leftarrow i_2]$ connecting the array modeling the states before and after the procedure call.

By using axiom Q-NO-UPDATE and Q-UPDATE and removing some statements, we get the following formula:

$$
\begin{array}{l}
m_{Pre}[1] = m_{Pre}[2] \wedge \\
p_y(m_{Pre}[1] + 5, m_{l_{3\_t_1}}[1], 1) \\
p_y(m_{Pre}[2] + 5, m_{l_{3\_t_2}}[2], 2) \Rightarrow \\
m_{l_{3\_t_1}}[1] + 6 = m_{l_{3\_t_2}}[2] + 6
\end{array}
$$

By instantiating axiom AXIOM-RELA as follows

$$
\begin{array}{c}
(p_y(m_{Pre}[1] + 5, m_{l_{3\_t_1}}[1], 1) \wedge p_y(m_{Pre}[2] + 5, m_{l_{3\_t_2}}[2], 2) \wedge \\
1 \neq 2 \wedge m_{Pre}[1] + 5 = m_{Pre}[2] + 5) \Rightarrow m_{l_{3\_t_1}}[1] = m_{l_{3\_t_2}}[2]
\end{array}
$$

we can prove validity of the formula.

We can define the following rule for resuming the use of a set of relational properties defined in an environment $\tilde{\hat{\xi}}$ in the case of the proof of a relational property:

$$
\frac{\begin{array}{c}(u_{\mathbb{Q}_r}, a) = Axio(\tilde{\hat{\xi}}) \\ \forall q \in \hat{\mathcal{VC}}_h(Use(\varsigma_1; \varsigma_2, a), (\tilde{\hat{\mathcal{D}}}_{t_b}[\![\tilde{\beta}_1]\!], \tilde{\hat{\mathcal{D}}}_{t_b}[\![\tilde{\beta}_2]\!]), \hat{\phi}_a(t_1) \cup \hat{\phi}_a(t_2)).\mathrm{smt}(u_{\mathbb{Q}} \cup u_{\mathbb{Q}_r}, q) = V\end{array}}{\vdash \{\tilde{\beta}_1\}\varsigma_1\langle t_1\rangle \sim \varsigma_2\langle t_2\rangle\{\tilde{\beta}_2\}}
$$

where function $Axio$ takes an environment of relational properies $\tilde{\hat{\xi}}$ and returns the set of equivalent axiomatized properties $u_{\mathbb{Q}_r}$ (added to the set of axioms $u_{\mathbb{Q}}$ of function smt), and a mapping $a : \mathbb{Y} \rightharpoonup \mathcal{P}(\mathbb{P})$ from command name to a set of predicates (the correspondence between procedures and the set of predicates used in the axiomatizations). Using this mapping, function Use adds assumptions in the Self-Composed programs such that the set of axiomatizations $u_{\mathbb{Q}_r}$ can be used by function smt to prove the verification conditions. Notice that the rule is not complete

for simplicity; the proofs for the relational properties defined in $\tilde{\tilde{\xi}}$ and the standard contract for each tag defined in the extended execution environment $\hat{\phi}_a$ are absent.

Finally, we can notice that a similar rule can be defined for standard Hoare Triples such that relational properties can be used.

$$\frac{(u_{\mathbb{Q}_r}, a) = Axio(\tilde{\hat{\xi}}) \qquad \forall q \in \hat{\mathcal{VC}}_h(Use(\varsigma, a), \beta_1, \beta_2, \hat{\xi}).\mathrm{smt}(u_{\mathbb{Q}} \cup u_{\mathbb{Q}_r}, q) = V}{\vdash \{\beta_1\}\varsigma\{\beta_2\}}$$

In this chapter, we have presented our solution for using relational properties as a contract on multiple procedures in the context of the R-WHILE* language. In the following Chapter 7, we adapt this approach to the C language, the ACSL specification language and the FRAMA-C platform.

Chapter 7

Relational Properties for C programs

This chapter focuses on the verification of relational properties in the context of the C programming language, the specification language ACSL, the FRAMA-C platform and the WP plugin presented in Chapter 2.

As the specification language ACSL has no support for relational contracts, we have first extended the specification language with a new clause and new constructs. Then, since the WP plugin only supports simple function contracts, we have adapted the approach proposed in Sections 6.2 and 6.3 to the C programming language and ACSL, in a FRAMA-C plugin [1] called RPP (Relational Property Prover). RPP translates the relational contracts into C code and standard ACSL annotations analysable by WP.

This chapter is organized as follows. Section 7.1 presents the extensions we added to the specification language ACSL to express relational properties. In this section, we assume the reader is familiar with the concepts and syntax of the ACSL specification language introduced in Section 2.1. Section 7.2 presents the transformation required to make the ACSL extension suitable for the verification condition generator WP, and performed by the RPP plugin. Finally, Section 7.3 presents some applications of RPP.

7.1 Specification language

As mentioned earlier, the ACSL specification language has no support for expressing relational contracts. Therefore, we extended the language and introduce relational properties by external `relational` annotations. The grammar is introduced on Figure 7.1 for predicates and Figure 7.2 for terms.

As the supported subset of C includes functions with parameters, we first present in Section 7.1.1 the grammar on a subset of the C syntax, equivalent to the R-WHILE* language, *i.e.* functions without parameters and return value. Then, in Section 7.1.2 we present the extension in case of functions with parameters

[1] `https://frama-c.com/download/frama-c-plugin-development-guide.pdf`

call-id ::= *id*

funct-param ::= *relational-call-terms*$^+$

funct-name ::= *poly-id*

funct-call ::= `\call`(*funct-name*, *funct-param*, *call-id*)

call-parameter ::= *funct-call*$^+$

call-set ::= `\callset`(*call-parameter*)

relational-pred ::= `\true` | `\false`
| *relational-terms* `==` *relational-terms*
| *relational-terms* `!=` *relational-terms*
| *relational-terms* `<=` *relational-terms*
| *relational-terms* `>=` *relational-terms*
| *relational-terms* `>` *relational-terms*
| *relational-terms* `<` *relational-terms*
| *relational-pred* `&&` *relational-pred*
| *relational-pred* `||` *relational-pred*
| *relational-pred* `==>` *relational-pred*
| `!` *relational-pred*
| `\forall` *binders* `;` *relational-pred*
| `\exists` *binders* `;` *relational-pred*

external-relational-annot ::= `/*@ relational` *relational-clause* `; */`

relational-clause ::= `\forall` *binders* `;` *call-set* `==>` *relational-pred*
| *call-set* `==>` *relational-pred*
| *relational-pred*

Figure 7.1 – Grammar for predicates in relational clauses

literal	::=	`\true` | `\false` | *int* | *float*
relational-label	::=	`Post_` *call-id*
	|	`Pre_` *call-id*
bin-op	::=	`+` | `-` | `*` | `/`
result-reference	::=	`\callresult`(*call-id*)
pure-function-param	::=	*relational-call-terms*$^+$
pure-funct-name	::=	*poly-id*
pure-funct-call	::=	`\callpure`(*pure-funct-name*, *pure-funct-param*)
relational-call-terms	::=	*literal*
	|	*pure-funct-call*
	|	*relational-call-terms bin-op relational-call-terms*
relational-terms	::=	*literal*
	|	*relational-terms bin-op relational-terms*
	|	*result-reference*
	|	`\at`(*poly-id*, *relational-label*)
	|	*pure-funct-call*

Figure 7.2 – Grammar for terms in relational clauses

7.1.1 From R-WHILE* to C

We now show on Figure 7.3 an example of a `relational` property as defined in the previous section, in order to introduce the various constructions of the grammar.

```
1 int x;
2
3 /*@ assigns y \from y;*/
4 void h();
5
6 /*@ relational R1:
7         \callset(\call(h,id1),\call(h,id2)) ==>
8         \at(x,Pre_id1) < \at(x,Pre_id2) ==> \at(x,Post_id1) < \at(x,Post_id2);
9 */
```

Figure 7.3 – Annotated C function with `relational` annotations

Specification of relational properties in the case of a subset of C, equivalent to R-WHILE*, is similar to the definition of relational properties shown in Section 6.1. An external relational annotation is composed of two parts. A set of calls *call-set* defining the related function calls, and the relational property itself, given as an ACSL predicate in the *relational-pred* part. Note that the relational annotation is added after the function definition to ensure that this function is declared.

In the property `R1` of Figure 7.3, two function calls to `h` are explicitly specified in the `\callset`

construct, using construct `\call`. Since we might refer to memory locations in either the pre- or the post-state of any call implied in the relational property, we need to be able to make explicit references to these states, and not only to the state of a single call. Therefore, each call has its own identifier *call-id*, equivalent to the tag in $\mathbb{T}$ we have seen in the previous sections. In the case of Figure 7.3, we have identifiers `id1` and `id2` associated to each call. Each such *call-id* gives rise to two logic labels. Namely, `Pre_`*call-id* refers to the pre-state of the corresponding call, and `Post_`*call-id* to its post-state. These labels can in particular be used in the ACSL term `\at(e,`L`)` that indicates that the term `e` must be evaluated in the context of the program state linked to logic label L, as is the case in our example to indicate that the value of global variable `x` is evaluated in four different states:

- `Pre_id1`, the state before the execution of function `h` with tag `id1`
- `Post_id1`, the state after the execution of function `h` with tag `id1`
- `Pre_id2`, the state before the execution of function `h` with tag `id2`
- `Post_id2`, the state after the execution of function `h` with tag `id2`

7.1.2 Functions with Parameters

In case of relational properties linking function with parameters and returned values, a relational clause is composed of three parts. An example is shown on Figure 7.4.

```
1  int y;
2
3  /*@ assigns \result \from x,y;*/
4  int f(int x);
5
6  /*@ relational R1:
7          \forall int x1,x2;
8          \callset(\call(f,x1,id1),\call(f,x2,id2)) ==>
9          \at(y,Pre_id1) < \at(y,Pre_id2) ==>
10         x1 < x2 ==> \callresult(id1) < \callresult(id2);
11 */
```

Figure 7.4 – Annotated C function with parameter and `relational` annotations

In addition to the two parts previously presented, we also declare a set of universally quantified variables, that will be used to express the arguments of the calls that are involved in the clause. Moreover, a new construct `\callresult`, that takes a *call-id* as parameter can be used to refer to the value returned by the corresponding call defined in *call-set*.

In the case of a pure function, *i.e.* a function only "assigning" the return value and whose return value only depends on the parameters, it is possible to use the `\callpure` contructions to denote the value returned by a pure function. This allows specifying relational properties over pure functions without the overhead required for handling side-effects. Nested `\callpure` are allowed. To ensure that a function has no side effects, an `assigns \result \from` `param` clause

must be used (Section 2.1), where `param` corresponds to a subset of the set of parameters of the function. An example is shown on Figure 7.5.

```
1 /*@ assigns \result \from x;*/
2 int f(int x);
3
4 /*@ relational R1:
5       \forall int x1,x2; x1 < x2 ==> \callpure(f,x1) < \callpure(f,x2);
6 */
```

Figure 7.5 – Annotated pure C function with **`relational`** annotations

A similar specification approach exists for defining relational properties on Java pure methods [DM06] in the JML specification language.

7.2 Code Transformation

As for the presentation of the specification language, we divide the presentation of the code transformation. First, we present the transformation on a subset of the C syntax, equivalent to the R-WHILE* language. Then, we present the transformation on function with parameters and return value. Finally, we show the transformation in case of pointers.

7.2.1 From R-WHILE* to C

As said before in Section 7.1.1, the new syntax for relational properties enables us to speak about the value of global variables at various states of the execution, thanks to the logic labels bound to each call involved in the **`\callset`** of the property. This is for instance the case in the relational property of Figure 7.6, where we give a body to the function `h` of Figure 7.3. The property indicates that `h` is monotonic with respect to global variable `y`, in the sense that if a first call to `h` is done in a state `Pre_id1` where the value of `y` is strictly less than in the pre-state `Pre_id2` of a second call, this will also be the case in the respective post-states `Post_id1` and `Post_id2`.

```
1 int y;
2
3 /*@ assigns y \from y;*/
4 void h(){
5   int a = 10;
6   y = y + a;
7   return;
8 }
9 /*@ relational R1:
10      \callset(\call(h,id1), \call(h,id2))  ==>
11      \at(y,Pre_id1) < \at(y,Pre_id2) ==> \at(y,Post_id1) < \at(y,Post_id2);
12 */
```

Figure 7.6 – Relational property on a function with side-effect

However, this new syntax is not supported by the WP plugin. Thus, we apply the self-composition transformation presented in Section 6.2 to property `R1` over function `h` to get the code shown on Figure 7.7.

```
1 int y_id1;
2 int y_id2;
3
4 void relational_wrapper_1(void){
5   int a_1 = 10;
6   y_id1 = y_id1 + a_1;
7   int a_2 = 10;
8   y_id2 = y_id2 + a_2;
9   /*@ assert Rpp:
10        \at(y_id1,Pre) < \at(y_id2,Pre) ==> \at(y_id1,Here) < \at(y_id2,Here);
11  */
12  return;
13 }
```

Figure 7.7 – Self-Composition on a function with side-effect

To fully perform the self-composition transformation, we have to generate a new function, commonly called *wrapper* function. The wrapper function inlines the calls occurring in the relational property under analysis, with a suitable renaming of local and global variables to avoid interferences between the calls, as each function call must operate on its own memory state, separated from the other calls in order for self-composition to work. Notice the creation of global variables as needed to let each part of the wrapper use its own set of copies (lines 1–2 in Figure 7.7).

However, to avoid useless creation of global variables (renaming variables that are not used by the function), we require that each function involved in a relational property has been equipped with a proper set of ACSL **`assigns`** clauses, including **`\from`** components. This constraint let us determine the parts of the global state that are accessed (either for writing or for reading) by the functions under analysis and that must be subject to duplication. In case of function `h`, only global variable `y` is read and written by the function. Thus, only variable `y` must be duplicated.

Then, in the spirit of calculational proofs [LP13], we state an assertion equivalent to the relational property (lines 7–8 in Figure 7.7). The proof of such an assertion is possible with the deductive verification tool WP.

```
1 /*@ axiomatic Relational_axiom_1 {
2     predicate h_acsl(int y_pre, int y_post, int id);
3
4     lemma Relational_lemma_1:
5      \forall int y_id2_pre, y_id2_post, y_id1_pre, y_id1_post, id1, id2;
6       h_acsl(y_id2_pre, y_id2_post, id2) ==> h_acsl(y_id1_pre, y_id1_post, id1)
7       ==> id1 != id2 ==> y_id1_pre < y_id2_pre ==> y_id1_post < y_id2_post;
8  }*/
```

Figure 7.8 – Axiomatisation of a relational property on a function with side-effect

To be able to use the property, we apply the axiomatization presented in Section 6.3 to property `R1` over function `h` to obtain the code shown on Figure 7.8.

We have to generate a new global annotation. This ACSL axiomatic definition introduces a logical reformulation of the relational property as a lemma over otherwise unspecified predicates (`h_acsl` in the example) as presented in Section 6.3. We declare a predicate that takes as parameters the relevant parts of the program states that are involved in the property. In the example, this is shown on lines 5–7 in Figure 7.8, where we have four quantified variables representing the value of global variable `y` before and after both calls involved in the relational property. Moreover, each predicate takes an integer parameter modeling the identifier that must be unique.

```
1 int y;
2
3 /*@ assigns y \from y;*/
4 void p(){
5   h();
6   return;
7 }
8
9 /*@ relational R2:
10      \callset(\call(p,id1), \call(p,id2))  ==>
11      \at(y,Pre_id1) < \at(y,Pre_id2) ==> \at(y,Post_id1) < \at(y,Post_id2);
12 */
```

Figure 7.9 – Relational property on a function calling a function with side-effect

Using this reformulation, a relational property linking a function calling function `h` can use property `R1`. For example, if we consider function `p`, shown on Figure 7.9, calling function `h`. We can prove relational property `R2`, similar to property `R1`.

```
1 int y_id1; int y_id2;
2
3 /*@ assigns y_id1 \from y_id1;*/
4 void h_id1();
5
6 /*@ assigns y_id2 \from y_id2;*/
7 void h_id2();
8
9 void relational_wrapper_2(void){
10   l1:h_id1();
11   /*@ assert h_acsl(\at(y_id1,l1),\at(y_id1,Here),1);
12   */
13   l2:h_id2();
14   /*@ assert h_acsl(\at(y_id1,l2),\at(y_id1,Here),2);
15   */
16   /*@ assert Rpp:
17         \at(y_id1,Pre) < \at(y_id2,Pre) ==> \at(y_id1,Here) < \at(y_id2,Here);
18   */
19   return;
20 }
```

Figure 7.10 – Self-Composition on a function calling a function with side-effect

Applying the self-composition transformation presented in Section 6.2 to property `R2` over

function `p` gives the code shown on Figure 7.10. As for property `R1`, the wrapper function inlines the calls occurring in the relational property under analysis, and applies some renaming. Moreover, the function calls are also renamed. Notice the creation of function prototypes with the associated contract, on line 1–2 of Figure 7.10.

In addition, new assertions are generated after the functions calls, on line 11 and 14 of Figure 7.10. As for the **assume** construct used in Section 6.3, they specify that there is an exact correspondence between the original C function and its newly generated logical ACSL counterpart so that the axiomatics of Figure 7.8 can be used. However, the ACSL syntax contains no construct for defining assumptions, therefore we have to use assertions that are to be considered as axioms and are not proven. For instance, assertion on line 11 of Figure 7.10, states that predicate `h_acsl` holds with two arguments representing the values of `y` before and after the execution of `h`.

7.2.2 Functions with Parameters

In Section 7.1.2 we introduced a new construct `\callpure(f,args)`, denoting the value returned by the call to a pure function `f` with arguments `<args>`. In Figure 7.11, property `R1` at lines 9–10 expresses the maximum (function `max`) of a pair using absolute values (function `abs`).

```
1  /*@ requires x > INT_MIN;
2      assigns  \result \from x;
3  */
4  int abs (int x){
5    return (x >= 0) ? x : (-x);
6  }
7
8  /*@ assigns \result \from x,y;*/
9  int max(int x,int y){
10   return (x >= y) ? x : y;
11 }
12
13 /*@ relational R1:
14       \forall int x,y; INT_MIN < x-y <= INT_MAX ==>
15       \callpure(max,x,y) == (x+y+\callpure(abs,x - y))/2;
16 */
```

Figure 7.11 – Relational property on functions with parameters

Applying the self-composition transformation to property `R1` over function `max` and `abs` gives the code of Figure 7.12. The transformation is as before, with in addition, the *wrapper* function taking the quantified variables as parameters. These parameters are used to initialize the variables corresponding to the parameters of the functions.

The axiomatization, shown on Figure 7.13, is also as before, with in addition, the declaration of the predicate taking as parameters the returned value and the formal parameters of the C function.

```
1 void relational_wrapper(int x, int y){
2   int ret_var_1, ret_var_2;
3   int x_1 = x; int y_1 = y;
4   int x_2 = x-y;
5   ret_var_1 = (x_1 >= y_1) ? x_1 : y_1;
6   ret_var_2 = (x_2 >= 0) ? x_2 : (-(x_2));
7   /*@ assert  ret_var_1 ==  ((x + y) + ret_var_2) / 2;*/
8   return;
9 }
```

Figure 7.12 – Self-Composition on functions with parameters

```
1 /*@ axiomatic Relational_axiom {
2   predicate max_acsl(int x, int y, int result, int id);
3   predicate abs_acsl(int x, int result, int id);
4   lemma Relational_lemma{L}:
5       \forall int x, y,r1,r2,id1,id2;
6        max_acsl(x, y,r1,id1) ==> abs_acsl(x - y, r2,id2)
7        ==> r1 ==  ((x + y) + r2) / 2;
8 }*/
```

Figure 7.13 – Axiomatisation of a relational property on functions with parameters

7.2.3 Support of Pointers

In Section 7.2.1, we have shown how to specify relational properties in presence of side effects over global variables, and how the transformations for both proving and using a property are performed. The support of pointer dereference is similar with some nuance. An example of a relational property on a function `k` using pointers (monotonicity with respect to the content of a pointer) is given in Figure 7.14, where `k` is specified to assign `*y` using only its initial content.

```
1 int* y
2
3 /*@ assigns *y \from *y;*/
4 void k(){
5   *y = *y + 1;
6   return;
7 }
8
9 /*@ relational R1:
10      \callset(\call(k,id1), \call(k,id2))  ==>
11      \at(*y,Pre_id1) < \at(*y,Pre_id2)  ==> \at(*y,Post_id1) < \at(*y,Post_id2);
12 */
```

Figure 7.14 – Relational property on a function with pointers

As proven in [BDR11] Self-Composition works if the memory footprint of each call is separated from the others; considering the memory states defined now by a pair

$$\Sigma = \Sigma_v \times \Sigma_p$$

composed of a memory state $\Sigma_v = \mathbb{X} \rightharpoonup \mathbb{N}$ mapping locations to naturals, and a heap $\Sigma_p =$

$\mathbb{N} \rightharpoonup \mathbb{N}$ mapping pointers (naturals) to naturals. We assume unable to access to the value of locations using pointers *i.e* pointers allow only accessing the heap.

The requirement of Self-Composition for disjoint memory states $\bar{\sigma}_1 \cap \bar{\sigma}_2 = \emptyset$ can be refined in separated locations $\sigma_1 \cap \sigma_2 = \emptyset$, and separated heaps $\sigma_{p_1} \cap \sigma_{p_2} = \emptyset$. Thus, we must ensure that pointers that are accessed during distinct calls point to different memory locations. As above, such accesses are given by **assigns** clauses, combined with construct **\from**, in the contract of the corresponding C functions. Memory separation is enforced using ACSL's built-in predicate **\separated**. For the wrapper function, we add a **requires** clause stating the appropriate **\separated** locations. This can be seen on Figure 7.15, line 3, where we request that the copies of pointer `y` used for the inlining of both calls to `k` points to two separated areas in the memory.

```
1 int *y_id1; int *y_id1;
2
3 /*@ requires \separated(y_id1, y_id2);*/
4 void relational_wrapper_1(){
5   *y_id1 = *y_id1 + 10;
6   *y_id2 = *y_id2 + 10;
7   /*@ assert Rpp:
8       \at(*y_id1,Pre) < \at(*y_id2,Pre) ==> \at(*y_id1,Here) < \at(*y_id2,Here);*/
9   return;
10 }
```

Figure 7.15 – Self-Composition on functions with pointers

We also need to refine the declaration of the predicate in presence of pointer accesses. First, the predicate now needs to explicitly take as parameters the pre- and post-states of the C function. In ACSL, this is done by specifying *logic labels* as special parameters, surrounded by braces, as shown in line 2 of Figure 7.16. Second, a **reads** clause allows one to specify the footprint of the predicate, that is, the set of memory accesses that the validity of the predicate depends on (line 2). Similarly, the lemma on lines 4–11 takes 4 logic labels as parameters, since it relates two calls to `k`, each of them having a pre- and a post-state.

```
1 /*@ axiomatic Relational_axiom_1 {
2       predicate k_acsl{pre, post}(int *y, int id)  reads \at(*y,pre), \at(*y,post);
3
4       lemma Relational_lemma_1 {pre_id1, post_id1, pre_id2, post_id2}:
5          \forall int *y_id1, int *y_id2, id1,id2;
6          \separated(y_id1,y_id2)
7          ==> k_acsl{pre_id1, post_id1}(y_id1,id1)
8          ==> k_acsl{pre_id2, post_id2}(y_id2,id2)
9          ==> id1 != id2
10         ==> \at(*y_id1,pre_id1) < \at(*y_id2,pre_id2)
11         ==> \at(*y_id1,post_id1) < \at(*y_id2,post_id2);
12 }*/
```

Figure 7.16 – Axiomatisation of a relational property on functions with pointers

Notice that the memory separation assumption restrict the relational properties to the case where pointers are always different, which does not reflect the initial relational property.

7.3 Relational Property Prover (RPP)

We have seen in the previous Sections 7.1 and 7.2 how to express relational properties on C functions and how we can generate C code and plain ACSL specifications in order to take advantage of a standard verification condition generator for proving relational properties and use them as hypotheses in subsequent verification tasks.

To check that this approach works in practice, we have implemented this approach in a plugin of FRAMA-C called RPP [2]. The transformations performed by RPP are published in [BKGP17, BKG+18], with slight differences from what was presented in Section 6.3 on how relational properties are made usable. Typically, the use of unique identifiers (to ensure different calls) are absent from the current implementation, implying some limitation on the type of supported relational properties that can be used soundly in other proofs. Supported properties includes:

- properties linking functions with different names *i.e.* the resulting axiomatisation linking only different predicates such that the identifiers are not required.
- properties whose precondition ensures disjoint function calls *i.e* the relational precondition implies that the axiomatisation is instantiated with different calls.

The fact that a property can be used soundly is not checked by the tool, it is left to the user to ensure that the property is supported.

Despite these limitations, we have tested with success our tool on different benchmarks [3]. These tests aim at confirming:

- the ability to specify various relational properties over a large class of functions;
- the capacity to prove and use such properties using the generated transformation;
- the support of a large range of function implementations;
- the ability to use other techniques (runtime checks, test generation for invalidating the property) when WP fails to prove a corresponding property.

Subsection 7.3.1 will present our own benchmark composed of a mix of different types of relational properties. This benchmark is mainly designed to validate the first two items. Subsection 7.3.2 will show how RPP has performed on the benchmark proposed in [SD16]. This will confirm the second and third points. Finally, we will present in Subsection 7.3.3 and 7.3.4 our use of the E-ACSL and STADY plugins assessing the last point.

7.3.1 Internal Examples

As stated previously, we have tested RPP on a set of relational properties extracted from real case studies, shown on Figure 7.17. This includes in particular monotonicity (row 1), factorial (row 2), order on function (row 3), idempotent (row 4), encryption (row 5), properties found

[2] https://github.com/lyonel2017/Frama-C-RPP

[3] https://github.com/lyonel2017/RPP-Examples-TAP-2018

Num	Relational Property	Specified / Generated	Verified	Used	Side effect	Loop	Recursive
1	$\forall x1, x2 \in \mathbb{Z}:$ $x1 < x2 \Rightarrow f(x1) < f(x2)$	✓	✓	✓	✓	✗	✗
2	$\forall\, x;$ $f(x+1) = f(x) * (x+1)$	✓	✓	✓	✓	✗	✓
3	$\forall x,\ f_1(x) \leq f_2(x) \leq f_3(x)$	✓	✓	✗	✗	✗	✗
4	$\forall x, f(f(x)) = f(x)$	✓	✓	✗	✗	✓	✗
5	$\forall\, Msg, Key;$ $Decrypt(Encrypt(Msg, Key), Key) = Msg$	✓	✓	✓	✓	✓	✗
6	$\forall\, t, sub_{t1}, ..., sub_{tn};$ $t = sub_{t1} \cup ... \cup sub_{tn} \Rightarrow$ $max(t) = max(max(sub_{t1}), ..., max(sub_{tn}))$	✓	✓	✗	✓	✓	✗
7	$\forall\, A, B;$ $(A+B)^\intercal = (A^\intercal + B^\intercal)$	✓	✓	✗	✗	✓	✗
8	$\det(A) = \det(A^\intercal)$	✓	✓	✗	✗	✓	✗
9	$\forall x1, x2, y, f(x1 + x2, y) = f(x1, y) + f(x2, y)$	✓	✓	✓	✗	✗	✓
10	$\forall a, b, c,\ \mathrm{Med}(a, b, c) = \mathrm{Med}(a, c, b)$	✓	✓	✗	✗	✗	✗

Figure 7.17 – Summary of relational properties considered by RPP

in map/reduce, as the one in row 6, stating that the choice of the partitioning for the initial set of data should not play a role in the final result. The benchmark is also composed of more academic examples like linear algebraic properties of matrices, over functions containing loops (rows 7 and 8), additivity row 9, or the property of row 10, that states the symmetry of the median of three numbers.

Figure 7.17 summarizes the results obtained on the benchmark. The first three columns indicate respectively whether the corresponding property could be specified and the corresponding code transformation generated, proved and used as a hypothesis in other proofs. The last three columns show what kind of C constructs are used in the implementation of the functions under analysis, namely side effects, presence of loops (which are always difficult for WP-related verification techniques, due to the need for loop invariants), and presence of recursive functions.

7.3.2 Comparator Functions

We also evaluated RPP on the benchmark proposed in [SD16]. It is composed of a collection of flawed and corrected implementations of comparators over a variety of data types written in Java, inspired from a collection of Stackoverflow [4] questions. Translating the Java code into C was straightforward and fully preserved the semantics of the functions. We focused on the same properties as [SD16], that is anti-symmetry (P1), transitivity (P2) and extensionality (P3) (mentioned in Section 1.3). Mathematically, these properties can be expressed as such:

[4] `https://stackoverflow.com`

$$P1 : \forall\, s1, s2.\ compare(s1, s2) = -compare(s2, s1)$$
$$P2 : \forall\, s1, s2, s3.\ compare(s1, s2) > 0 \wedge compare(s2, s3) > 0$$
$$\Rightarrow compare(s1, s3) > 0$$
$$P3 : \forall\, s1, s2, s3.\ compare(s1, s2) = 0 \Rightarrow (compare(s1, s3) = compare(s2, s3))$$

Results are depicted in Table 7.18. For each comparator, we indicate whether the properties P1, P2 and P3 hold according to RPP (✓ and ✗ show whether the property was proved valid by WP). We get similar results as [SD16], with the exception of PokerHand, for which the generated wrapper function seems currently out of reach for WP (limits of scalability due to the combinatorial explosion of self-composition). However, by rewriting the function in a more modular way and using the capacity to use relational properties, WP was able to handle the example.

7.3.3 Counterexample Generation

For the properties that do not hold in the comparator benchmark, we have been able to find counterexamples thanks to the proposed encoding of a relational property by self-composed code and using another FRAMA-C plugin, STADY [PKB+16]. STADY[5] is a testing-based counterexample generator. In particular, STADY tries to find an input vector that will falsify an ACSL annotation for which WP could not decide whether it holds, thereby showing that the code is not conforming to the specification.

We apply STADY to try to find a test input such that the **`assert`** clause at the end of the `wrapper` function is false. The results are shown in the STADY columns of Figure 7.18. Obviously, STADY does not try to find counterexamples for properties that are proved valid by WP. For properties that are not proved valid, ✓ indicates that a counterexample is found (within a timeout of 30 seconds), while ✄ indicated the only case where a counterexample is not generated before a 30-second timeout. A longer timeout (60 minutes) did not improve the situation in that case. Symbol ✐ denotes two cases where the code translation uses features that are currently not yet supported by STADY. As shown in the table, thanks to the RPP translation, STADY was able to find counterexamples for almost all unproven properties. Notice that some examples required minor modifications so that STADY can be used. In particular, to be able to use testing, we need to add bodies for unimplemented functions. Other modifications consisted in reducing the input space to a representative smaller domain (by limiting the size of an input array) for some examples to facilitate counterexample generation [PKB+16].

7.3.4 Runtime Assertion Checking

The code transformation technique of RPP also enables runtime verification of relational properties through the E-ACSL plugin [DKS13, VSK17]. More precisely, the E-ACSL plugin

[5] See `https://github.com/gpetiot/Frama-C-StaDy`

	Proof (WP)			Counterex. gen. (STADY)		
Benchmark	P1	P2	P3	P1	P2	P3
ArrayInt-false.c	✓	✓	✗	–	–	✓
ArrayInt-true.c	✓	✓	✓	–	–	–
CatBPos-false.c	✗	✗	✗	✓	✓	✓
Chromosome-false.c	✓	✗	✗	–	✂	✓
Chromosome-true.c	✓	✓	✓	–	–	–
ColItem-false.c	✗	✗	✗	✓	✓	✓
ColItem-true.c	✓	✓	✓	–	–	–
Contact-false.c	✓	✗	✗	–	✓	✓
Container-false-v1.c	✗	✓	✓	✓	–	–
Container-false-v2.c	✗	✗	✗	✓	✓	✓
Container-true.c	✓	✓	✓	–	–	–
DataPoint-false.c	✗	✗	✗	✓	✓	✓
FileItem-false.c	✓	✓	✗	–	–	✓
FileItem-true.c	✓	✓	✓	–	–	–
IsoSprite-false-v1.c	✗	✗	✗	✓	✓	✓
IsoSprite-false-v2.c	✗	✗	✓	✓	✓	–
Match-false.c	✗	✓	✗	✓	–	✓
Match-true.c	✓	✓	✓	–	–	–
NameComparator-false.c	✗	✓	✓	✓	–	–
NameComparator-true.c	✓	✓	✓	–	–	–
Node-false.c	✓	✓	✗	–	–	✓
Node-true.c	✓	✓	✓	–	–	–
NzbFile-false.c	✗	✓	✓	✓	–	–
NzbFile-true.c	✓	✓	✓	–	–	–
PokerHand-false.c	✓	✗	✗	–	✎	✎
PokerHand-true.c	✓	✓	✓	–	–	–
Solution-false.c	✓	✓	✗	–	–	✓
Solution-true.c	✓	✓	✓	–	–	–
TextPosition-false.c	✓	✗	✗	–	✓	✓
TextPosition-true.c	✓	✓	✓	–	–	–
Time-false.c	✗	✓	✓	✓	–	–
Time-true.c	✓	✓	✓	–	–	–
Word-false.c	✗	✗	✓	✓	✓	–
Word-true.c	✓	✓	✓	–	–	–

Figure 7.18 – Comparator properties analysed with WP and STADY after RPP translation

translates ACSL annotations into C code that will check them at runtime and abort execution if one of the annotations fails. We tested the E-ACSL plugin on the test inputs generated by STADY in order to check that each generated counterexample does indeed violate the relational property. As expected, the obtained results validate those of the previous section. Since coun-

terexample generation with STADY [PKB+16] basically includes a runtime assertion checking step for each test datum considered during the test generation process, we do not present the results of this step in separate columns.

Chapter 8

Direct Translation of Relational Properties

In Chapter 6, we have shown how to prove and use relational properties in the context of self-composition and the R-WHILE* language. In Chapter 7 we implemented the approach of Chapter 6 in the context of the C language and the FRAMA-C platform. The tool, called RPP, was tested on different benchmarks.

During the implementation of RPP we noticed that performing the renaming required by self-composition and the self-composition itself can be tedious to perform. We have also seen in Section 7.2.3 that in case of pointers, self-composition requires some additional assumptions about memory separation [BDR11]. The generation of such hypotheses can become cumbersome as the complexity of the code under analysis grows and make the resulting formula harder to prove valid.

Therefore, we propose in this chapter an alternative approach for the verification of relational properties by translating the properties directly, without self-composition. The new method requires no renaming or separation of the memory states. Moreover, the new translation allows to refining the axiomatization of relational properties proposed in Section 6.3.2.

The chapter is organized as follows. Section 8.1 presents the translation of relational properties into formalus of MFOL, using concepts introduced in Section 5.3. Section 8.2 presents an alternative axiomatization of relational properties solving some issues of the method proposed in Section 6.3.2.

8.1 Direct Translation of Relational Properties

In this section, we show how we can generate verification conditions for proving relational properties without Self-Composition. We first define the functions for embedding relational extended arithmetic and boolean expressions into MFOL. Those functions are almost equivalent to the functions $\mathcal{T}_a$ and $\mathcal{T}_b$ (defined in Section 5.3.2) for translating extended arithmetic and boolean expressions. Then, we define function $\tilde{\tilde{\mathcal{VC}}}_r$ returning the verification conditions that

must be valid such that a relational property is valid. The function uses function $\hat{\mathcal{T}}_c$ (defined in Section 5.3.3) for computing the strongest postcondition.

8.1.1 Translation of $\tilde{\hat{\mathbb{E}}}_a$ and $\tilde{\hat{\mathbb{E}}}_b$

Example 8.1. If we consider the following relational extended boolean expression

$$at(x, l)\langle t_1\rangle + at(x, l)\langle t_2\rangle = 10,$$

we want a translation that returns a formula of the form

$$m_{t_1}[1] + m_{t_2}[1] = 10,$$

where m_{t_1} represents the memory state at label l for tag t_1, m_{t_2} represents the memory state at label l for tag t_2 *i.e.* for different tags, the same label has different array variables that model the memory state.

For the natural numbers representing a location, we propose that the choice is shared *i.e* for different tags, the same location has the same natural number representing a location. This will be important for the following Section 8.2. Such a choice is sound because we consider the case where address of location cannot be handled by the language *i.e.* we have no pointers of locations.

To get such a result, we first define mapping Υ from tags to mapping Θ (defined in Section 5.3.2), that maps labels to array variables,

$$\Upsilon = \mathbb{T} \to \Theta,$$

and we use metavariables $\upsilon, \upsilon_0, \upsilon_1, ...$ to range over the set Υ.

Then, we define R, the pair of environment Υ and Δ (defined in Section 5.3.1) mapping location to natural numbers

$$R = \Upsilon \times \Delta.$$

Finally, we define function $lift_r$, returning the constant term associated to location x for a given environment δ, and the array variable associated to label l and a tag t for a given environment υ:

Definition 8.1. Function $lift_r : R \times \mathbb{L} \times \mathbb{X} \times \mathbb{T} \to R \times \mathbb{M} \times {\mathbb{E}_q}^{\text{nat}}$ returning the constant term associated to location x for a given environment δ, and the array variable associated to label l and a tag t for a given environment υ:

$$lift_r((\upsilon,\delta),l,x,t) = \begin{cases} ((\upsilon[t \leftarrow \theta],\delta'),m,e) & \text{if } \upsilon(t) = \bot \\ \quad \text{where} & \\ \qquad ((\theta,\delta'),m,e) = lift_s((\emptyset,\delta),l,x) & \\ ((\upsilon[t \leftarrow \theta],\delta'),m,e) & \text{otherwise} \\ \quad \text{where} & \\ \qquad ((\theta,\delta'),m,e) = lift_s((\upsilon(t),\delta),l,x) & \end{cases}$$

Note that function $lift_r$ is a generalization of function $lift_s$ (defined in Section 5.3.2), supporting tags. If we call function $lift_r$ with an environment υ defined for tag t and label l, and an environment δ defined for location x, and a label l, a location x and a tag t, we get as result the associated array variable $(\upsilon(t))(l)$, the natural $\delta(x)$ and the environment υ and δ unchanged:

$$lift_r((\{t_1 \to \{l_1 \to m_1\}\},\{x_1 \to 1\}),l_1,x_1,t_1) = ((\{t_1 \to \{l_1 \to m_1\}\},\{x_1 \to 1\}),m_1,1)$$

If there are undefined binding, the environments are lifted;

$$lift_r((\{t_1 \to \{l_1 \to m_1\}\},\{x_1 \to 1\}),l_2,x_2,t_1) = ((\{t_1 \to \{l_1 \to m_1, l_2 \to m_2\}\},\{x_1 \to 1, x_2 \to 2\}),m_2,2)$$

$$lift_r((\{t_1 \to \{l_1 \to m_1\}\},\{x_1 \to 1\}),l_1,x_1,t_2) = ((\{t_1 \to \{l_1 \to m_1\}, t_2 \to \{l_1 \to m_2\}\},\{x_1 \to 1\}),m_2,1)$$

Using function $lift_r$, we can define function $\tilde{\tilde{\mathcal{T}}}_a$ for translating an relational extended arithmetic expression $\tilde{\tilde{\mathbb{E}}}_a$ into a term of $\mathbb{E}_q$, and function $\tilde{\tilde{\mathcal{T}}}_b$ translating a relational extended boolean expression $\tilde{\tilde{\mathbb{E}}}_b$ into an formula of $\mathbb{Q}$.

Definition 8.2. Function $\tilde{\hat{\mathcal{T}}}_a : \tilde{\hat{\mathbb{E}}}_a \rightarrow (R \rightarrow \mathbb{E}_q \times R)$, translating a relational extended arithmetic expression $\tilde{\hat{\mathbb{E}}}_a$ into a term,of $\mathbb{E}_q$, is defined by structural induction on relational extended arithmetic expressions:

$$\tilde{\hat{\mathcal{T}}}_a[\![n]\!]r = ([\![n]\!], r)$$
$$\tilde{\hat{\mathcal{T}}}_a[\![at(x,l)\langle t\rangle]\!]r = ([\![m[n]]\!], r') \text{ where } (r', m, n) = \mathit{lift}_r(r, l, x, t)$$

$$\tilde{\hat{\mathcal{T}}}_a[\![\tilde{\alpha}_0 \ opa \ \tilde{\alpha}_1]\!]r = ([\![e_1 \ opa \ e_2]\!], r'')$$
$$\text{where } (e_1, r') = \tilde{\hat{\mathcal{T}}}_a[\![\tilde{\alpha}_0]\!]r \text{ and } (e_2, r'') = \tilde{\hat{\mathcal{T}}}_a[\![\tilde{\alpha}_1]\!]r'.$$

Definition 8.3. Function $\tilde{\hat{\mathcal{T}}}_b : \tilde{\hat{\mathbb{E}}}_b \rightarrow (R \rightarrow \mathbb{Q} \times R)$, translating a relational extended boolean expression $\tilde{\hat{\mathbb{E}}}_b$ into an formula of $\mathbb{Q}$, is defined by structural induction on relational extended boolean expression:

$$\tilde{\hat{\mathcal{T}}}_b[\![true]\!]r = ([\![T]\!], r)$$
$$\tilde{\hat{\mathcal{T}}}_b[\![false]\!]r = ([\![F]\!], r)$$

$$\tilde{\hat{\mathcal{T}}}_b[\![\tilde{\alpha}_0 \ opb \ \tilde{\alpha}_1]\!]r = ([\![e_0 \ opb \ e_1]\!], r'')$$
$$\text{where } (e_0, r') = \tilde{\hat{\mathcal{T}}}_a[\![\tilde{\alpha}_0]\!]r \text{ and } (e_1, r'') = \tilde{\hat{\mathcal{T}}}_a[\![\tilde{\alpha}_1]\!]r'$$

$$\tilde{\hat{\mathcal{T}}}_b[\![\tilde{\beta}_0 \ opl \ \tilde{\beta}_1]\!]r = ([\![q_0 \ opl \ q_1]\!], r'')$$
$$\text{where } (q_0, r') = \tilde{\hat{\mathcal{T}}}_b[\![\tilde{\beta}_0]\!]r \text{ and } (q_1, r'') = \tilde{\hat{\mathcal{T}}}_b[\![\tilde{\beta}_1]\!]r'$$

$$\tilde{\hat{\mathcal{T}}}_b[\![\neg\tilde{\beta}]\!]r = ([\![\neg q]\!], r') \text{ where } (q, r') = \tilde{\hat{\mathcal{T}}}_b[\![\tilde{\beta}]\!]r$$

$$\tilde{\hat{\mathcal{T}}}_b[\![p^n(\tilde{\alpha}_1, ..., \tilde{\alpha}_n)]\!]r = ([\![p^n(e_1, ..., e_n)]\!], r_n)$$
$$\text{where } (e_1, r_1) = \hat{\mathcal{T}}_a[\![\tilde{\alpha}_1]\!]r \text{ and ... and } (e_n, r_n) = \hat{\mathcal{T}}_a[\![\tilde{\alpha}_n]\!]r_{n-1}.$$

As said earlier, the definitions are almost identical to the definitions of function $\hat{\mathcal{T}}_a$ for translating an extended arithmetic expression $\hat{\mathbb{E}}_a$ into a term of $\mathbb{E}_q$, and function $\hat{\mathcal{T}}_b$ translating a extended boolean expression $\hat{\mathbb{E}}_b$ into an formula of $\mathbb{Q}$. The only difference is to call the function lift_r instead of lift_s in function $\hat{\mathcal{T}}_a$.

8.1.2 Verification of Relational Properties

We have given in Section 5.3.4 the definition of function $\hat{\mathcal{VC}}_h$ for the generation of verification conditions for a Hoare Triple (defined in Section 5.3.4). Basically, the verification condition states that the strongest postcondition, obtained from the precondition and the program, implies the postcondition.

In the case of relational properties, a similar function $\hat{\tilde{\mathcal{VC}}}_r$ for the generation of verification conditions can be defined. The function translates the relational precondition and, for each defined tag in a relational extended execution environment, the associated command into a formula (the relational strongest postcondition) that must imply the formula corresponding to the relational postcondition. As for function $\hat{\mathcal{VC}}_h : \hat{\mathbb{C}} \times (\hat{\mathbb{E}}_b \times \hat{\mathbb{E}}_b) \times \hat{\Xi} \rightarrow \mathcal{P}(\mathbb{Q})$, taking a command, a pre- and post-condition, and an environment of contract, function $\hat{\tilde{\mathcal{VC}}}_r : \hat{\Phi}_c \times (\hat{\tilde{\mathbb{E}}}_b \times \hat{\tilde{\mathbb{E}}}_b) \times \hat{\Phi}_a \rightarrow \mathcal{P}(\mathbb{Q})$ takes a relational extended execution environment ($\hat{\Phi}_c = \mathbb{T} \rightharpoonup \hat{\mathbb{C}} \times \hat{\Psi}$), a relational pre- and post-condition and an environment that maps tags to an environment of contracts ($\hat{\Phi}_a = \mathbb{T} \rightharpoonup \hat{\Xi}$).

Definition 8.4. Function $\hat{\tilde{\mathcal{VC}}}_r : \hat{\Phi}_c \times (\hat{\tilde{\mathbb{E}}}_b \times \hat{\tilde{\mathbb{E}}}_b) \times \hat{\Phi}_a \rightarrow \mathcal{P}(\mathbb{Q})$ returning the set of verification conditions that must be valid in order for the relational property to be valid:

$$\hat{\tilde{\mathcal{VC}}}_r(\hat{\phi}_c, (\tilde{\beta}_{Pre}, \tilde{\beta}_{Post}), \hat{\phi}_a) = u_{\mathbb{Q}} \cup \{[\![q_{Pre} \wedge q_{t_1} \wedge ... \wedge q_{t_n} \Rightarrow q_{Post}]\!]\}$$

where

(i) $\{t_1, ..., t_n\} = dom(\hat{\phi}_c)$,

(ii) $m_{t_1} = \mathcal{N}_{v_m}, ..., m_{t_n} = \mathcal{N}_{v_m}$,

(iii) $\upsilon = \{t_1 \rightarrow \{Pre \rightarrow m_{t_1}\}, ..., t_n \rightarrow \{Pre \rightarrow m_{t_n}\}\}$,

(iv) $(q_{Pre}, (\upsilon', \delta)) = \hat{\tilde{\mathcal{T}}}_b[\![\tilde{\beta}_{Pre}]\!](\upsilon, \emptyset)$,

(v)
$$\begin{cases} (q_{t_1}, u_{\mathbb{Q}_{t_1}}, m'_{t_1}, (\theta_{t_1}, \delta_{t_1})) = \hat{\mathcal{T}}_c[\![body(\hat{\phi}_c(t_1))]\!](\hat{\phi}_a(t_1), (T, \emptyset, m_{t_1}, (\upsilon'(t_1), \delta)), \\ ..., \\ (q_{t_n}, u_{\mathbb{Q}_{t_n}}, m'_{t_n}, (\theta_{t_n}, \delta_{t_n})) = \\ \qquad\qquad \hat{\mathcal{T}}_c[\![body(\hat{\phi}_c(t_n))]\!](\hat{\phi}_a(t_n), (T, \emptyset, m_{t_n}, (\upsilon'(t_n), \delta_{t_{n-1}})), \end{cases}$$

(vi) $\upsilon'' = \{t_1 \rightarrow \theta_{t_1}[Post \leftarrow m'_{t_1}]\}), ..., \{t_n \rightarrow \theta_{t_n}[Post \leftarrow m'_{t_n}]\}$,

(vii) $(q_{Post}, \_) = \hat{\tilde{\mathcal{T}}}_b[\![\tilde{\beta}_{Post}]\!](\upsilon'', \delta_{t_n})$,

(viii) $u_{\mathbb{Q}} = \bigcup_i \{[\![q_{Pre} \Rightarrow q]\!] | q \in u_{\mathbb{Q}_{t_i}}\}$

(ii) First, for each defined tag t_i in $\hat{\phi}_c$, we define a new array variable m_{t_i} modeling the state before the evaluation of the command $body(\hat{\phi}_c(t_i))$.

(iii) Then, we define environment υ associating to each defined tag in $\hat{\phi}_c$ an environment that maps label Pre, modeling the state before the evaluation of the command $body(\hat{\phi}_c(t_i))$, to the associated array variable.

(iv) Then, we translate the relational extended boolean expression $\tilde{\beta}_{Pre}$ into a formula q_{Pre} in the context of υ.

(v) Then, for each defined tag t_i in $\hat{\phi}_c$, we translate the command $body(\hat{\phi}_c(t_i))$ into a formula q_{t_i}. Note that the environment δ, mapping locations to naturals, is shared.

(vi) Then, we define environment υ'' associating for each defined tag t_i in $\hat{\phi}_c$ the environment θ_{t_i}, obtained from the translation of command $body(\hat{\phi}_c(t_i))$, and the mapping label $Post$, modeling the state after the evaluation of the command $body(\hat{\phi}_c(t_i))$, to the array variable m'_{t_i}.

(vii) Then, we translate the relational extended boolean expression $\tilde{\beta}_{Post}$ into a formula $\mathbb{Q}_{Post}$ in the context of υ''.

(vii) Then, we define the set of verification conditions $u_{\mathbb{Q}}$ composed of the set of verification conditions $u_{\mathbb{Q}_{t_i}}$, where we add q_{Pre} as hypothesis; the verification conditions for each command are generated only from the precondition T (true), so we add the formula corresponding to the relational precondition.

The set of verification conditions that must be proven valid for proving that the relational property holds is composed of the set of verification conditions $u_{\mathbb{Q}}$ and the verification conditions stating that the relational strongest postcondition (q_{Pre} and all q_{t_i}) implies the relational postcondition (q_{Post}).

Using function $\hat{\tilde{\mathcal{VC}}}_r$ and function $\hat{\mathcal{VC}}_p$ (for the generation of verification conditions for a procedure contract) we can define rule RECURSIVE-RELATIONAL for proving relational properties and using standard procedure contracts.

$$\frac{\forall t \in dom(\hat{\phi}_a).\forall q \in \hat{\mathcal{VC}}_p(state(\hat{\phi}_c(t)), \hat{\phi}_a(t)).\mathrm{smt}(u_{\mathbb{Q}}, q) = V \qquad \forall q \in \hat{\tilde{\mathcal{VC}}}_r(\hat{\phi}_c, (\tilde{\beta}_1, \tilde{\beta}_2), \hat{\phi}_a).\mathrm{smt}(u_{\mathbb{Q}}, q) = V}{\vdash \{\tilde{\beta}_1\}\varsigma_1\langle t_1\rangle \sim \varsigma_2\langle t_2\rangle\{\tilde{\beta}_2\}} \text{(RECURSIVE-RELATIONAL)}$$

Note that this rule requires no modification of the initial property *i.e* composing in sequence the programs linked by the property, removing the tags from the boolean expression, in opposition to rule RECURSIVE-SELF-COMP-E where self-composition is used.

Since we create for each tag a fresh array variable modelling the memory state and each program is handled separately, no additional hypotheses are required. By contrast, self-composition required that the set of variables, labels and command names used in the programs are disjoint.

Example 8.2. We take again the example used to present Self-composition in the context of R-WHILE* in Section 6.2:

$$\begin{array}{c} \{at(x_1, Pre)\langle t_1\rangle = at(x_1, Pre)\langle t_2\rangle\} \\ \begin{array}{lcl} l_1 : x_1 := x_1 + 5; & & l_1 : x_1 := x_1 + 5; \\ l_2 : \textbf{call}(y); & \langle t_1\rangle \sim & l_2 : \textbf{call}(y); \quad \langle t_2\rangle \\ l_3 : x_1 := x_1 + 6 & & l_3 : x_1 := x_1 + 6 \end{array} \\ \{at(x_1, Post)\langle t_1\rangle = at(x_1, Post)\langle t_2\rangle\}, \end{array}$$

with a relational extended execution environment defined, for each tag, by:

$$\hat{\phi}_c(t_1) = \left(\begin{array}{l} l_1 : x_1 := x_1 + 5; \\ l_2 : \textbf{call}(y); \\ l_3 : x_1 := x_1 + 6 \end{array} , \{y \to l : x_1 := x_1 + 1\} \right),$$

$$\hat{\phi}_c(t_2) = \hat{\phi}_c(t_1),$$

and an environment that maps tags to an environment of contracts $\hat{\phi}_a$ defined by:

$$\left\{ \begin{array}{l} t_1 \to \left\{ y \to \left(\begin{array}{l} true, \\ at(x_1, Pre) + 1 = at(x_1, Post), \\ \{x_1\} \end{array} \right) \right\}, \\ t_2 \to \left\{ y \to \left(\begin{array}{l} true, \\ at(x_1, Pre) + 1 = at(x_1, Post), \\ \{x_1\} \end{array} \right) \right\} \end{array} \right\}$$

We can use function $\tilde{\hat{\mathcal{VC}}}_r$ to generate the verification conditions that must be valid such that the relational property is valid:

$$\tilde{\hat{\mathcal{VC}}}_r(\hat{\phi}_c, \left(\begin{array}{l} at(x_1, Pre)\langle t_1 \rangle = at(x_1, Pre)\langle t_2 \rangle, \\ at(x_1, Post)\langle t_1 \rangle = at(x_1, Post)\langle t_2 \rangle \end{array} \right), \hat{\phi}_a)$$

Assuming that x_1 is mapped to index 1, we get the following VC:

$$\begin{array}{l} m_{Pre\_t_1}[1] = m_{Pre\_t_2}[1] \wedge \\ T \wedge \\ m_{l_2\_t_1} = m_{Pre\_t_1}[1 \leftarrow m_{Pre\_t_1}[1] + 5] \wedge \\ m_{l_3\_t_1}[1] = m_{l_2\_t_1}[1] + 1 \wedge m_{l_3\_t_1} = m_{l_2\_t_1}[1 \leftarrow i_1] \wedge \\ m_{Post\_t_1} = m_{l_3\_t_1}[1 \leftarrow m_{l_3\_t_1}[1] + 6] \wedge \\ T \wedge \\ m_{l_2\_t_2} = m_{Pre\_t_2}[1 \leftarrow m_{Pre\_t_2}[1] + 5] \wedge \\ m_{l_3\_t_2}[1] = m_{l_2\_t_2}[1] + 1 \wedge m_{l_3\_t_2} = m_{l_2\_t_2}[1 \leftarrow i_2] \wedge \\ m_{Post\_t_2} = m_{l_3\_t_2}[1 \leftarrow m_{l_3\_t_2}[1] + 6] \Rightarrow \\ m_{Post\_t_1}[1] = m_{Post\_t_2}[1] \end{array}$$

We have in blue the part that corresponds to the program with tag t_1 and in red the part that corresponds to program with tag t_2. Note that, there is no relation between the blue and the red parts. By using axioms Q-NO-UPDATE and Q-UPDATE, we get the following valid formula:

$$\begin{array}{l} m_{Pre\_t_1}[1] = m_{Pre\_t_2}[1] \Rightarrow \\ m_{Pre\_t_1}[1] + 5 + 1 + 6 = m_{Pre\_t_2}[1] + 5 + 1 + 6 \end{array}$$

In the previous example, the blue and red parts in the resulting formulas have the same shape. Only the name of the variables are different. This is due to the relational property that links two instances of the same program. Thus, in the case of relational properties relating the same program ς n times, we can only call one time function $\hat{\mathcal{T}}_c$ for program ς and rename the variable in the resulting formula with fresh variables to get the same result.

Such an approach is proposed in [SS14] for an efficient verification of non-interference (Section 1.3.2) by minimizing the number of calls to a verification condition generator; a simple renaming of all variables in a formula is in generale of lower complexity then the generation of verification conditions.

Finally, we can notice that it is not always required to verify all verification conditions returned by function $\tilde{\hat{\mathcal{VC}}}_r$, for the same reasons as for Self-Composition (Section 6.2).

8.1.3 Relational Properties and Pointers

We have seen in Section 7.2.3 the Self-Composition transformation and the axiomatization for proving and using relational properties with pointers. The proposed Self-Composition transformation required that pointers associated to different tags are separated. As mentioned earlier, the generation of such hypotheses can be cumbersome. Moreover, the separation hypothesis restrict the relational properties that can be proven.

A direct translation into verification conditions would not require separation hypotheses between pointers from different tags. Function $\tilde{\hat{\mathcal{VC}}}_r$ handles the memory states associated to different tags separately, thus a pointer associated to a tag t is only used in the context of memory states associated to tag t. Pointers associated to different tags could have the same value in a relational property.

Notice that if we want to support pointers of locations the current transformation requires some refinements. In the actual model the value of a locations x for two tags is represented by accessing two different arrays at the same index: for example $m_{l\ t_1}[1]$ for the value of location x at label l for tag t_1, and $m_{l\ t_2}[1]$ for the value of location x at label l for tag t_2. Assume we have a relational property comparing two pointers of location x for two different tags. The pointer of location x for tag t_1 would be represent by the natural 1 and for tag t_2 by natural 1. However, in case of a non-deterministic allocator, they is no guarantie that location x is associated twice to the same position in the memory (for two different tags). To ensures that the pointers of location are not comparable in case of a non-deterministic allocator, a possible solution would be to use a memory models with regions [Bar11], *i.e.* the access to the array would no be a simpl natural, but a composition of region (modeling the tag) and a natural value: for example $m_{l\ t_1}[adress(t_1, 1)]$ for the value of location x at label l for tag t_1, and $m_{l\ t_2}[adress(t_2, 1)]$ for the value of location x at label l for tag t_2. A similar memory model already existe in the tool WP.

8.1.4 Implementation in RPP

The verification of relational properties by direct translation into verification conditions has been partially implemented in the plugin RPP. We used a direct communication with the WP plugin

without going through C and ACSL constructs. This led to a much smaller and simpler implementation, compared to the equivalent implementation of the Self-Composition based approach.

The current implementation does not support pointers and the use of relational properties; only a part of the benchmark presented in Section 7.3.1 and 7.3.2 is supported. A comparison between the two implemented approaches is therefore left as future work.

8.2 Extended Axiomatization

We have presented in Section 6.3.2 how to use relational contracts by translating them into an axiomatization. Part of the translation consists in modelling by predicates the procedures connected by the property. The signature of those predicates depends on the property itself, more precisely depending on which locations are examined in the property.

Example 8.3. We consider the following relational contract relating the procedure y for tags t_1 and t_2:

$$\{(y,t_1),(y,t_2)\} \rightarrow \begin{pmatrix} at(x_1,Pre)\langle t_1\rangle < at(x_1,Pre)\langle t_2\rangle, \\ at(x_1,Post)\langle t_1\rangle < at(x_1,Post)\langle t_2\rangle \end{pmatrix} \qquad \text{(R-CONTRACT-1)}$$

We have seen in Section 6.3.2 that an equivalent axiomatization would be:

$$\begin{gathered} \forall v_{1\_t_1Pre}, v_{1\_t_1Post}, v_{1\_t_2Pre}, v_{1\_t_2Post}, v_{t_1id}, v_{t_2id} : \text{nat}, \\ (p(v_{1\_t_1Pre}, v_{1\_t_1Post}, v_{1\_t_1id}) \wedge p(v_{1\_t_2Pre}, v_{1\_t_2Post}, v_{1\_t_2id}) \wedge \\ v_{t_1id} \neq v_{t_2id} \wedge \\ v_{1\_t_1Pre} < v_{1\_t_2Pre}) \Rightarrow v_{1\_t_2Post} < v_{1\_t_2Post}. \end{gathered}$$

We now consider a second relational contract relating again procedure y for tags t_1 and t_2:

$$\{(y,t_1),(y,t_2)\} \rightarrow \begin{pmatrix} at(x_2,Pre)\langle t_1\rangle < at(x_2,Pre)\langle t_2\rangle \wedge \\ at(x_3,Pre)\langle t_1\rangle < at(x_3,Pre)\langle t_2\rangle, \\ at(x_2,Post)\langle t_1\rangle < at(x_2,Post)\langle t_2\rangle \wedge \\ at(x_3,Post)\langle t_1\rangle < at(x_3,Post)\langle t_2\rangle \end{pmatrix} \qquad \text{(R-CONTRACT-2)}$$

and the equivalent axiomatization:

$$\begin{gathered} \forall v_{2\_t_1Pre}, v_{2\_t_1Post}, v_{3\_t_1Pre}, v_{3\_t_1Post}, v_{2\_t_2Pre}, v_{2\_t_2Post}, v_{3\_t_2Pre}, v_{3\_t_2Post}, \\ v_{t_1id}, v_{t_2id} : \text{nat}, \\ (p'(v_{2\_t_1Pre}, v_{2\_t_1Post}, v_{3\_t_1Pre}, v_{3\_t_1Post}, v_{t_1id}) \wedge \\ p'(v_{2\_t_2Pre}, v_{2\_t_2Post}, v_{3\_t_2Pre}, v_{3\_t_2Post}, v_{t_2id}) \wedge \\ v_{t_1id} \neq v_{t_2id} \wedge \\ v_{2\_t_1Pre} < v_{2\_t_2Pre} \wedge v_{3\_t_1Pre} < v_{3\_t_2Pre}) \Rightarrow v_{2\_t_1Post} < v_{2\_t_2Post} \wedge v_{3\_t_1Post} < v_{3\_t_2Post}, \end{gathered}$$

We note that the predicates modelling the same procedure call are different for each axiomatization. This is due to the relational properties that links different sets of locations. The axiomatization for property R-CONTRACT-1 links locations x_1 for two different tags and two

different labels. The axiomatization for property R-CONTRACT-2 links locations x_2 and x_3 for two different tags and two different labels.

Applying the approach proposed in Section 6.3.2 to connect the previous axiomatizations to a relational property on programs calling procedure y gives the following result:

$$\left\{ \begin{array}{l} at(x_1, Pre)\langle t_1\rangle < at(x_1, Pre)\langle t_2\rangle \wedge \\ at(x_2, Pre)\langle t_1\rangle < at(x_2, Pre)\langle t_2\rangle \wedge \\ at(x_3, Pre)\langle t_1\rangle < at(x_3, Pre)\langle t_2\rangle \end{array} \right\}$$

$l_1 : x_1 := x_1 + 5;$
$l_2 : \textbf{call}(y);$
$l_n : \textbf{assume}(p(at(x_1, l_2), at(x_1, l_n), 1) \wedge \quad \langle t_1\rangle$
$\qquad p'(at(x_2, l_2), at(x_2, l_n), at(x_3, l_2), at(x_3, l_n), 1));$
$l_3 : x_1 := x_1 + 6$

$$\sim$$

$l_1 : x_1 := x_1 + 5;$
$l_2 : \textbf{call}(y);$
$l_n : \textbf{assume}(p(at(x_1, l_2), at(x_1, l_n), 2) \wedge \quad \langle t_2\rangle$
$\qquad p'(at(x_2, l_2), at(x_2, l_n), at(x_3, l_2), at(x_3, l_n), 2));$
$l_3 : x_1 := x_1 + 6$

$$\left\{ \begin{array}{l} at(x_1, Post)\langle t_1\rangle < at(x_1, Post)\langle t_2\rangle \wedge \\ at(x_2, Post)\langle t_1\rangle < at(x_2, Post)\langle t_2\rangle \wedge \\ at(x_3, Post)\langle t_1\rangle < at(x_3, Post)\langle t_2\rangle \end{array} \right\}$$

We have to assume after each procedure call both predicates associated to both axiomatizations.

8.2.1 Alternative axiomatization

We propose in the following an alternative axiomatization to avoid defining different predicates for the same procedure for each relational property. The refinement consists in passing the array variable (modelling the memory states) as parameter to the predicate instead of the locations related by the property. Thus, the predicates takes always three parameters; the array variables modelling the memory state before and after the procedure call, and the identifier. A consequence is that variables modelling locations are replaced by access to arrays at the corresponding index. Thus, the axiomatization will depend on the environment mapping locations to naturals. As we have chosen, in Section 8.1, to share the environment δ for all tags, the environments mapping locations to naturals is the same for each tag.

Example 8.4. For relational contracts R-CONTRACT-1 and R-CONTRACT-2 we can give the following alternative axiomatizations assuming environment $\delta = \{x_1 \to 1, x_2 \to 2, x_3 \to 3\}$:

$$\begin{array}{c} \forall m_{t_1 Pre}, m_{t_1 Post}, m_{t_2 Pre}, m_{t_2 Post} : \text{array}, v_{t_1 id}, v_{t_2 id} : \text{nat}, \\ (p(m_{t_1 Pre}, m_{t_1 Post}, v_{t_1 id}) \wedge p(m_{t_2 Pre}, m_{t_2 Post}, v_{t_2 id}) \wedge \\ v_{t_1 id} \neq v_{t_2 id} \wedge \\ m_{t_1 Pre}[1] < m_{t_2 Pre}[1]) \Rightarrow m_{t_1 Post}[1] < m_{t_2 Post}[1] \end{array} \qquad \text{(E-AXIOM-RELA-1)}$$

$$\begin{array}{c}\forall m_{t_1Pre}, m_{t_1Post}, m_{t_2Pre}, m_{t_2Post} : \text{array}, v_{t_1id}, v_{t_2id} : \text{nat},\\ (p(m_{t_1Pre}, m_{t_1Post}, v_{t_1id}) \wedge p(m_{t_2Pre}, m_{t_2Post}, v_{t_2id}) \wedge\\ v_{t_1id} \neq v_{t_2id} \wedge\\ m_{t_1Pre}[2] = m_{t_2Pre}[2] \wedge m_{t_1Pre}[3] = m_{t_2Pre}[3]) \Rightarrow\\ m_{t_1Post}[2] = m_{t_2Post}[2] \wedge m_{t_1Post}[3] = m_{t_2Post}[3]\end{array} \quad \text{(E-AXIOM-RELA-2)}$$

8.2.2 Connection between axiomatization and the procedures

The predicates in the axiomatization of relational properties take as parameter array variables. Thus, to be able to use the axiomatization, we have to refine the definition of predicates in extended boolean expression $\hat{\mathbb{E}}_b$ (defined in Section 5.1.2).

The set of predicate identifiers $\mathbb{P}$ is now composed of the set of identifiers $\mathbb{P}_{(n_1,n_2)}$ for predicates taking a pair of n_1 labels and n_2 memory location.

$$\mathbb{P} = \bigcup_{n_1,n_2 \in \mathbb{N}} \mathbb{P}_{(n_1,n_2)}$$

As in Section 5.1.2, we assume that all sets $\mathbb{P}_{(n_1,n_2)}$ are disjoint to get well typed predicates by definition:

$$\forall i_1, i_2, j_1, j_2 \in \mathbb{N}.(i_1 \neq i_2 \vee j_1 \neq j_2) \Rightarrow \mathbb{P}_{(i_1,j_1)} \cap \mathbb{P}_{(i_2,j_2)} = \emptyset$$

For the sake of simplicity we give in the following only the definition of function $\hat{\mathcal{T}}_b$, translating extended boolean expressions into formulas of MFOL, for the case of predicates with labels. The definition of evaluation function for extended boolean expression $\hat{\xi}_b$ is similar to the definition for $\hat{\mathcal{T}}_b$.

Definition 8.5. Function $\hat{\mathcal{T}}_b : \hat{\mathbb{E}}_b \rightarrow (S \rightarrow \mathbb{Q} \times S)$, translating a boolean expression $\hat{\mathbb{E}}_b$ into an formula of $\mathbb{Q}$, is defined, for the case of predicates with labels, by:

$$\hat{\mathcal{T}}_b[\![p^{n_1,n_2}((l_1, ..., l_{n_1})(\alpha_1, ..., \alpha_{n_2}))]\!]s = [\![(\mathrm{p}(m_1, ..., m_{n_1}, e_1, ..., e_{n_2})]\!], s_m)$$

where

$$(s_1, m_1) = \mathit{lift}_m(s, l_1) \; ... \; (s_{n_1}, m_{n_1}) = \mathit{lift}_m(s_{n_1-1}, l_{n_1})$$
$$(e_1, s_{n_1+1}) = \hat{\mathcal{T}}_a[\![\alpha_1]\!]s_{n_1} \; ... \; (e_{n_2}, s_m) = \hat{\mathcal{T}}_a[\![\alpha_n]\!]s_{m-1}$$

where function lift_m returning the array variables associated to a given label is defined as follows:

Definition 8.6. Function $lift_m : S \times \mathbb{L} \to S \times \mathbb{M}$, returning the array variable associated to label l for a given environment θ:

$$lift_m((\theta, \delta), l) = \begin{cases} ((\theta[l \leftarrow m], \delta), m) & \text{if } \theta(l) = \bot \\ \quad \text{where} & \\ \qquad m = \mathcal{N}_{v_m} & \\ ((\theta, \delta), \theta(x)) & \text{otherwise} \end{cases}$$

Example 8.5. We take the relational properties of the previous example and apply the approach for connecting the axiomatizations to procedure call using predicates with labels. We get the following result:

$$\left\{ \begin{array}{l} at(x_1, Pre)\langle t_1\rangle = at(x_1, Pre)\langle t_2\rangle \wedge \\ at(x_2, Pre)\langle t_1\rangle = at(x_2, Pre)\langle t_2\rangle \wedge \\ at(x_3, Pre)\langle t_1\rangle = at(x_3, Pre)\langle t_2\rangle \end{array} \right\}$$

$$\begin{array}{l} l_1 : x_1 := x_1 + 5; \\ l_2 : \textbf{call}(y); \\ l_n : \textbf{assume}(p((l_2, l_n), (1))); \\ l_3 : x_1 := x_1 + 6 \end{array} \langle t_1\rangle \sim \begin{array}{l} l_1 : x_1 := x_1 + 5; \\ l_2 : \textbf{call}(y); \\ l_n : \textbf{assume}(p((l_2, l_n), (2))); \\ l_3 : x_1 := x_1 + 6 \end{array} \langle t_2\rangle$$

$$\left\{ \begin{array}{l} at(x_1, Post)\langle t_1\rangle = at(x_1, Post)\langle t_2\rangle \wedge \\ at(x_2, Post)\langle t_1\rangle = at(x_2, Post)\langle t_2\rangle \wedge \\ at(x_3, Post)\langle t_1\rangle = at(x_3, Post)\langle t_2\rangle \end{array} \right\}$$

As the axiomatizations share the same predicate, only one predicate is assumed after the procedure calls.

The verification of the relational property is done using function $\tilde{\tilde{\mathcal{VC}}}_r$. Assuming that x_1 is mapped to index 1, x_2 is mapped to index 2 and x_3 is mapped to index 3 we get the following VC:

$$\begin{array}{c} m_{Pre\_t_1}[1] = m_{Pre\_t_2}[1] \wedge m_{Pre\_t_1}[2] = m_{Pre\_t_2}[2] \wedge m_{Pre\_t_1}[3] = m_{Pre\_t_2}[3] \wedge \\ T \wedge \\ m_{l_2\_t_1} = m_{Pre\_t_1}[1 \leftarrow m_{Pre\_t_1}[1] + 5] \wedge \\ p(m_{l_2\_t_1}, m_{l_3\_t_1}, 1) \wedge \\ m_{Post\_t_1} = m_{l_3\_t_1}[1 \leftarrow m_{l_3\_t_1}[1] + 6] \wedge \\ T \wedge \\ m_{l_2\_t_2} = m_{Pre\_t_2}[1 \leftarrow m_{Pre\_t_2}[1] + 5] \wedge \\ p(m_{l_2\_t_2}, m_{l_3\_t_2}, 2) \wedge \\ m_{Post\_t_2} = m_{l_3\_t_2}[1 \leftarrow m_{l_3\_t_2}[1] + 6] \Rightarrow \\ (m_{Post\_t_1}[1] = m_{Post\_t_2}[1] \wedge m_{Post\_t_1}[2] = m_{Post\_t_2}[2] \wedge m_{Post\_t_1}[3] = m_{Post\_t_2}[3]) \end{array}$$

By using axiom Q-NO-UPDATE and Q-UPDATE and removing some statements, we get the following formula:

$$m_{Pre\_t_1}[1] = m_{Pre\_t_2}[1] \wedge m_{Pre\_t_1}[2] = m_{Pre\_t_2}[2] \wedge m_{Pre\_t_1}[3] = m_{Pre\_t_2}[3] \wedge$$
$$p_y(m_{Pre\_t_1}[1 \leftarrow m_{Pre\_t_1}[1] + 5], m_{l_3\_t_1}, 1)$$
$$p_y(m_{Pre\_t_2}[1 \leftarrow m_{Pre\_t_2}[1] + 5], m_{l_3\_t_2}, 2) \Rightarrow$$
$$(m_{l_3\_t_1}[1] + 6 = m_{l_3\_t_2}[1] + 6 \wedge m_{l_3\_t_1}[2] = m_{l_3\_t_2}[2] \wedge m_{l_3\_t_1}[3] = m_{l_3\_t_2}[3])$$

By instantiating axiom E-AXIOM-RELA-1 as follows

$$(p_y(m_{Pre\_t_1}[1 \leftarrow m_{Pre\_t_1}[1] + 5], m_{l_3\_t_1}, 1) \wedge p_y(m_{Pre\_t_2}[1 \leftarrow m_{Pre\_t_2}[1] + 5], m_{l_3\_t_2}, 2) \wedge$$
$$1 \neq 2 \wedge$$
$$m_{Pre\_t_1}[1 \leftarrow m_{Pre\_t_1}[1] + 5][1] = m_{Pre\_t_2}[1 \leftarrow m_{Pre\_t_2}[1] + 5][1]) \Rightarrow$$
$$m_{l_3\_t_1}[1] = m_{l_3\_t_2}[1]$$

and axiom E-AXIOM-RELA-2

$$(p_y(m_{Pre\_t_1}[1 \leftarrow m_{Pre\_t_1}[1] + 5], m_{l_3\_t_1}, 1) \wedge p_y(m_{Pre\_t_2}[1 \leftarrow m_{Pre\_t_2}[1] + 5], m_{l_3\_t_2}, 2) \wedge$$
$$1 \neq 2 \wedge$$
$$m_{Pre\_t_1}[1 \leftarrow m_{Pre\_t_1}[1] + 5][2] = m_{Pre\_t_2}[1 \leftarrow m_{Pre\_t_2}[1] + 5][2] \wedge$$
$$m_{Pre\_t_1}[1 \leftarrow m_{Pre\_t_1}[1] + 5][3] = m_{Pre\_t_2}[1 \leftarrow m_{Pre\_t_2}[1] + 5][3])) \Rightarrow$$
$$m_{l_3\_t_1}[2] = m_{l_3\_t_2}[2] \wedge m_{l_3\_t_1}[3] = m_{l_3\_t_2}[3]$$

we can prove validity of the formula.

Notice that this axiomatization of relational properties does not work with Self-Composition. The renaming performed to ensure separated memory state may differ from one verification (using Self-Composition) to another. Thus, it is required that the renamed locations are given as parameter to the predicates.

Chapter 9

Conclusion

9.1 Summary

In this thesis we have provided two solutions regarding the problem of relational property verification: the support of modular relational property verification in existing verification tools, and the verification and use of relational properties in deductive verification.

In the context of a simple while language with recursive procedure calls, labels and predicates, called R-WHILE*, we have designed a technique for proving and using relational properties in deductive verification. The approach is based on Self-Composition (Section 6.2) for proving relational properties and introduces axiomatized relational properties to allow the use of relational properties in other deductive verification activities (Section 6.3).

The approach has been implemented in a tool called RPP, in the context of the C language and the FRAMA-C platform. RPP provides an extension to the ACSL language for expressing relational properties (Section 7.1). The extended annotations are translated into standard ACSL annotations and C code such that the WP plugin can be used (Section 7.2). The tool has been evaluated over a set of examples.

The implementation of Self-Composition in RPP has shown that even if Self-Composition is theoretically a simple approach, in practice, it requires a huge amount of work to fulfil the required assumption (Section 6.2). Therefore, we have designed an alternative verification approach (Section 8.1) that is not based on code transformation, but translates a relational property directly into verification conditions, like for the verification of standard Hoare Triples (Section 5.3.4). This alternative verification approach allows a refinement of the method for using relational properties (presented in Section 8.2). Moreover, this approach opens some perspectives that are discussed in the following section.

9.2 Perspectives

We discuss in this section some research directions that would provide interesting extensions to the work presented in this thesis.

9.2.1 Relational Properties Specification

In Section 7.1 we proposed an extended ACSL syntax for the specification of relational properties. This syntax can be sometimes a little bit heavy. The relational pre- and post-conditions are not clearly separated. Moreover, the use of specific label `Pre_`*call-id* appears cumbersome: we use a label and a tag to refer to a memory state.

A more interesting specification consists in having separated clauses in a global relational contract **`relational`**, as shown on Figure 9.1. Such a separation has already been proposed in [U+17] in the context of JML and Java programs.

```
1  int y;
2
3  int f(int x);
4  int g(int x);
5
6  /*@ relational R1:
7       callset \call{l1,l2}(f,id1), \call{l3,l4}(g,id2);
8       requires \param(x,id1) == \param(x,id2) && \at(y,l1) == \at(y,l3);
9       ensures \callresult(id1) == \callresult(id2)  && \at(y,l2) == \at(y,l4);
10 */
```

Figure 9.1 – Annotated C functions with extended global **`relational`** contract

A relational global contract would be composed of three parts:

- The set of related functions, written using a **`callset`** clause.
- The relational precondition, written using a **`requires`** clause.
- The relational postcondition, written using a **`ensures`** clause.

In the context of relational properties on functions, we are only interested in the state before and after each function call. Therefore, we can define for each function call, two labels to denote the state before and after the corresponding call. In case of the example shown on Figure 9.1, on line 7, we have defined for function `f` labels `l1` for the pre-state and `l2` for the post-state. For function `g`, we have defined labels `l3` and `l4`. Those labels can be used in the relational pre- and post-condition in term **`\at`**`(e,`L`)` that indicates that the term `e` must be evaluated in the context of the program state linked to logic label L. To refer to the parameter of a function, we use a new construct **`\param`**, taking as parameter the name of a formal parameter and a call identifier. In case of the example shown on Figure 9.1, we have defined, on line 8, that the formal paramter `x` associated to the identifier `id1` (function `f`) is equal to the formal parameter `x` associated to the identifier `id2` (function `g`).

The use of labels for naming the pre- and post-states of a function call in a relational property also allows some other perspectives.

Calling Relationships between Functions

Using the labels defined for a function call in a relational property, we can define calling relationships between functions. An example is shown on Figure 9.2, where the post-state of function `f`

```
1  int y;
2
3  void f(); void g(); void h(); void k(); void i();
4
5  /*@ relational R1:
6      callset
7        \call{l1,l2}(f,id1),                    \call{l4,l5}(h,id2),
8        \call{l2,l3}(g,id3),      \call{l5,l6}(k,id4),      \call{l5,l7}(i,id5);
9      requires \at(y,l1) == \at(y,l4);
10     ensures \at(y,l2) == \at(y,l5) && \at(y,l5) == \at(y,l6);
11 */
```

Figure 9.2 – Calling relationships between functions

is shared with the pre-state of function `g`. The post-state of function `h` is shared with the pre-state of functions `k` and `i`. Figure 9.3 shows the equivalent call graphs.

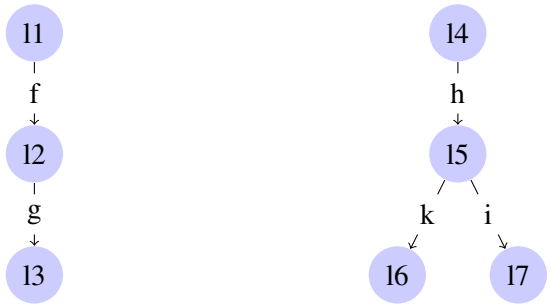

Figure 9.3 – Call graphs

Many combinations can be explored. Moreover, we can imagine defining built-in function that can be called inside such properties. Figure 9.4 shows an example where we have a **\havoc** built-in, similar to the havoc predicate in Boogie [BCD+05], stating that store `l2` and `l3` are equal except for the locations `y`, which are mapped to an arbitrary value in store `l3`.

```
1  int y;
2
3  void f(); void h();
4
5  /*@ relational R1:
6       callset \call{l1,l2}(f,id1),\havoc{l2,l3}(y), \call{l3,l4}(h,id2);
7       requires .....;
8       ensures ....;
9  */
```

Figure 9.4 – Calling relationships between functions with a built-in function

Equivalent Functions

In the current method for using relational contract, it is not possible to share properties between functions that are equivalent *i.e.* if the pre-state for two different functions are equal, the post-state are equal. It would be interesting to define an equivalence relation between functions, such that a relational property verified for one function is also valid for the equivalent function. Figure 9.5 shows an example of how such an equivalent relation could be specified by using a construct `\equiv(lx,ly)` stating that the memory states associated to labels `lx` and `ly` are equal.

```
1 void f();
2 void g();
3
4 /*@ relational R1:
5       callset \call{l1,l2}(f,id1), \call{l3,l4}(g,id2);
6       requires \equiv(l1,l3);
7       ensures \equiv(l2,l4);
8 */
```

Figure 9.5 – Equivalence relation between functions

An associated axiomatization would be of the form

```
lemma Relational_lemma_1 {l1,l2}: f{l1,l2}(id) <==> g{l1,l2}(id);
```

where `f{l1,l2}(id)` and `g{l1,l2}(id)` are respectively the predicates associated to function `f` and `g`.

9.2.2 Relational properties for Loops

We have seen in Example 4.5 that relational properties on loops can be useful in the verification of relational properties. Such properties present several challanges:

- The specification of relational loop invariant. A relational loop invariant can potentially relate different loops from different function bodies. However, in general it is not possible to refer to a specific loop. The problem can be faced by using a naming system for loops, for example using the label naming the statement.
- The verification of relational loop invariant. We have presented in Chapter 4 some solutions for this problem, limited to synchronized loop. Thus, loop must be synchronized by unrolling. More advances solutions can be found in [KKU18, U$^+$17] requiring no synchronization.
- The use of relational loop invariant in subsequent proofs. We see two solution to this problem:
  - Axiomatization of relational loop invariants. As for procedure contracts, presented in Section 6.3.2, we define an axiomatization of relational loop invariants using predicates to represent the linked loops. Those predicates are assumed after the loops to connect the axiomatizations to the loops.

– Convert loops to procedure calls; we convert the loops into function calls, as proposed in [KKU18], using similar transformations to those proposed in [HKLR13]. The transformation consists in replacing the loop by a recursive procedure call, and is formalized by the following rule:

$$\frac{\{\tilde{b_1}\}\textbf{call}(y)\langle t_1\rangle \sim c_2\langle t_2\rangle\{\tilde{b_2}\}}{\{\tilde{b_1}\}\textbf{while } b_1 \textbf{ do } \{c_1\}\langle t_1\rangle \sim c_2\langle t_2\rangle\{\tilde{b_2}\}}$$

where $state(\phi_c(t_1))(y) = \textbf{if } b_1 \textbf{ then } \{c_1; \textbf{call}(y)\} \textbf{ else } \{\textbf{skip}\}$. The benefit of converting loops to procedure calls is that the work proposed in Section 6.3.2 for using relational procedure contracts can be used. Moreover, the problem of specification does not arise since we have seen how relational procedure contracts can be defined.

9.2.3 Verification of Functional Dependencies

Functional dependencies clauses (presented in Section 2.1) are not verified in the current implementation of the WP plugin. Moreover, the implementation of Self-Composition in RPP is based on such clauses, as said in Section 7.2.1. Thus, the Self-Composition transformation may be erroneous. Therefore, it would be interesting to be able to verify such clauses by expressing functional dependencies as relational properties, similar to what is proposed in [CMPP11]. For a potentially modified location x that is specified to depend upon locations x_1,...x_k, the property can be expressed by the following relational property (for a given program c):

$$\left\{ \begin{array}{c} x_1\langle 1\rangle = x_1\langle 2\rangle \\ \wedge \\ \ldots \\ x_k\langle 1\rangle = x_k\langle 2\rangle \end{array} \right\} c\langle 1\rangle \sim c\langle 2\rangle\{x\langle 1\rangle = x\langle 2\rangle\}$$

Appendices

Appendix A

Tool Functions

A.1 Collector Functions

The following sections present the set of functions used for collecting locations $\mathbb{X}$, command names $\mathbb{Y}$, labels $\mathbb{L}$ and tags $\mathbb{T}$ from (relational/extended) arithmetic expressions, (relational/extended) boolean expressions and (extended) commands.

A.1.1 Locations

The sets of functions for collecting locations $\mathbb{X}$ from (extended) arithmetic expressions, (extended) boolean expressions and (extended) commands.

Definition A.1. Function $\mathcal{C}_{v_a} : \mathbb{E}_a \to \mathcal{P}(\mathbb{X})$, returning the set of memory locations used in an arithmetic expression $\mathbb{E}_a$, is defined by structural induction on arithmetic expressions:

$$\begin{aligned} \mathcal{C}_{v_a}[\![n]\!] &= \emptyset \\ \mathcal{C}_{v_a}[\![x]\!] &= \{x\} \\ \mathcal{C}_{v_a}[\![a_0 \ opa \ a_1]\!] &= \mathcal{C}_{v_a}[\![a_0]\!] \cup \mathcal{C}_{v_a}[\![a_1]\!]. \end{aligned}$$

Definition A.2. Function $\hat{\mathcal{C}}_{v_a} : \hat{\mathbb{E}}_a \to \mathcal{P}(\mathbb{X})$, returning the set of memory locations used in an extended arithmetic expression $\hat{\mathbb{E}}_a$, is defined by structural induction on extended arithmetic expressions:

$$\begin{aligned} \hat{\mathcal{C}}_{v_a}[\![n]\!] &= \emptyset \\ \hat{\mathcal{C}}_{v_a}[\![at(x,l)]\!] &= \{x\} \\ \hat{\mathcal{C}}_{v_a}[\![\alpha_0 \ opa \ \alpha_1]\!] &= \hat{\mathcal{C}}_{v_a}[\![\alpha_0]\!] \cup \hat{\mathcal{C}}_{v_a}[\![\alpha_1]\!]. \end{aligned}$$

Definition A.3. Function $\mathcal{C}_{v_b} : \mathbb{E}_b \to \mathcal{P}(\mathbb{X})$, returning the set of memory locations used in a boolean expression $\mathbb{E}_b$, is defined by structural induction on boolean expressions:

$$\begin{aligned}
\mathcal{C}_{v_b}[\![true]\!] &= \emptyset \\
\mathcal{C}_{v_b}[\![false]\!] &= \emptyset \\
\mathcal{C}_{v_b}[\![a_0 \ opb \ a_1]\!] &= \mathcal{C}_{v_a}[\![a_0]\!] \cup \mathcal{C}_{v_a}[\![a_1]\!] \\
\mathcal{C}_{v_b}[\![b_0 \ opl \ b_1]\!] &= \mathcal{C}_{v_b}[\![b_0]\!] \cup \mathcal{C}_{v_b}[\![b_1]\!] \\
\mathcal{C}_{v_b}[\![\neg b]\!] &= \mathcal{C}_{v_b}[\![b]\!].
\end{aligned}$$

Definition A.4. Function $\hat{\mathcal{C}}_{v_b} : \hat{\mathbb{E}}_b \to \mathcal{P}(\mathbb{X})$, returning the set of memory locations used in an extended boolean expression $\hat{\mathbb{E}}_b$, is defined by structural induction on extended boolean expressions:

$$\begin{aligned}
\hat{\mathcal{C}}_{v_b}[\![true]\!] &= \emptyset \\
\hat{\mathcal{C}}_{v_b}[\![false]\!] &= \emptyset \\
\hat{\mathcal{C}}_{v_b}[\![\alpha_0 \ opb \ \alpha_1]\!] &= \hat{\mathcal{C}}_{v_a}[\![\alpha_0]\!] \cup \hat{\mathcal{C}}_{v_a}[\![\alpha_1]\!] \\
\hat{\mathcal{C}}_{v_b}[\![\beta_0 \ opl \ \beta_1]\!] &= \hat{\mathcal{C}}_{v_b}[\![\beta_0]\!] \cup \hat{\mathcal{C}}_{v_b}[\![\beta_1]\!] \\
\hat{\mathcal{C}}_{v_b}[\![\neg\beta]\!] &= \hat{\mathcal{C}}_{v_b}[\![\beta]\!] \\
\hat{\mathcal{C}}_{v_b}[\![p^n(\alpha_1, .., \alpha_n)]\!] &= \hat{\mathcal{C}}_{v_a}[\![\alpha_0]\!] \cup .. \cup \hat{\mathcal{C}}_{v_a}[\![\alpha_n]\!].
\end{aligned}$$

Definition A.5. Function $\mathcal{C}_{v_c} : \mathbb{C} \to \mathcal{P}(\mathbb{X})$, returning the set of memory locations used in a command $\mathbb{C}$, is defined by structural induction on commands:

$$\begin{aligned}
\mathcal{C}_{v_c}[\![\textbf{skip}]\!] &= \emptyset \\
\mathcal{C}_{v_c}[\![x := a]\!] &= \{x\} \cup \mathcal{C}_{v_a}[\![a]\!] \\
\mathcal{C}_{v_c}[\![c_0; c_1]\!] &= \mathcal{C}_{v_c}[\![c_1]\!] \cup \mathcal{C}_{v_c}[\![c_0]\!] \\
\mathcal{C}_{v_c}[\![\textbf{assert}(b)]\!] &= \mathcal{C}_{v_b}[\![b]\!] \\
\mathcal{C}_{v_c}[\![\textbf{if } b \textbf{ then } \{c_0\} \textbf{ else } \{c_1\}]\!] &= \mathcal{C}_{v_b}[\![b]\!] \cup \mathcal{C}_{v_c}[\![c_0]\!] \cup \mathcal{C}_{v_c}[\![c_1]\!] \\
\mathcal{C}_{v_c}[\![\textbf{while } b \textbf{ do } \{c\}]\!] &= \mathcal{C}_{v_b}[\![b]\!] \cup \mathcal{C}_{v_c}[\![c]\!] \\
\mathcal{C}_{v_c}[\![\textbf{call}(y)]\!] &= \emptyset.
\end{aligned}$$

Definition A.6. Function $\hat{\mathcal{C}}_{v_c} : \xi_c \to \mathcal{P}(\mathbb{X})$, returning the set of memory locations used in an extended command $\hat{\mathbb{C}}$, is defined by structural induction on extended commands:

$$\begin{aligned}
\hat{\mathcal{C}}_{v_c}[\![l : \textbf{skip}]\!] &= \emptyset \\
\hat{\mathcal{C}}_{v_c}[\![l : x := a]\!] &= \{x\} \cup \mathcal{C}_{v_a}[\![a]\!] \\
\hat{\mathcal{C}}_{v_c}[\![\varsigma_0; \varsigma_1]\!] &= \hat{\mathcal{C}}_{v_c}[\![\varsigma_0]\!] \cup \hat{\mathcal{C}}_{v_c}[\![\varsigma_1]\!] \\
\hat{\mathcal{C}}_{v_c}[\![l : \textbf{assert}(\beta)]\!] &= \hat{\mathcal{C}}_{v_b}[\![\beta]\!] \\
\hat{\mathcal{C}}_{v_c}[\![l : \textbf{if } b \textbf{ then } \{\varsigma_0\} \textbf{ else } \{\varsigma_1\}]\!] &= \mathcal{C}_{v_b}[\![b]\!] \cup \hat{\mathcal{C}}_{v_c}[\![\varsigma_0]\!] \cup \hat{\mathcal{C}}_{v_c}[\![\varsigma_1]\!] \\
\hat{\mathcal{C}}_{v_c}[\![l : \textbf{while } b \textbf{ do } \{\varsigma\}]\!] &= \mathcal{C}_{v_b}[\![b]\!] \cup \hat{\mathcal{C}}_{v_c}[\![\varsigma]\!] \\
\hat{\mathcal{C}}_{v_c}[\![l : \textbf{call}(y)]\!] &= \emptyset.
\end{aligned}$$

A.1.2 Command Names

The set of functions for collecting command names $\mathbb{Y}$ from (extended) commands.

Definition A.7. Function $\mathcal{C}_f : \mathbb{C} \to \mathcal{P}(\mathbb{Y})$, returning the set of program names used in a command $\mathbb{C}$, is defined by structural induction on commands:

$$\begin{aligned}
\mathcal{C}_f[\![\textbf{skip}]\!] &= \emptyset \\
\mathcal{C}_f[\![x := a]\!] &= \emptyset \\
\mathcal{C}_f[\![c_0; c_1]\!] &= \mathcal{C}_f[\![c_0]\!] \cup \mathcal{C}_f[\![c_1]\!] \\
\mathcal{C}_f[\![\textbf{assert}(b)]\!] &= \emptyset \\
\mathcal{C}_f[\![\textbf{if } b \textbf{ then } \{c_0\} \textbf{ else } \{c_1\}]\!] &= \mathcal{C}_f[\![c_0]\!] \cup \mathcal{C}_f[\![c_1]\!] \\
\mathcal{C}_f[\![\textbf{while } b \textbf{ do } \{c\}]\!] &= \mathcal{C}_f[\![c]\!] \\
\mathcal{C}_f[\![\textbf{call}(y)]\!] &= \{y\}.
\end{aligned}$$

Definition A.8. Function $\hat{\mathcal{C}}_f : \hat{\mathbb{C}} \to \mathcal{P}(\mathbb{Y})$, returning the set of program names used in an extended command $\hat{\mathbb{C}}$, is defined by structural induction on extended commands:

$$\begin{aligned}
\hat{\mathcal{C}}_f[\![l : \textbf{skip}]\!] &= \emptyset \\
\hat{\mathcal{C}}_f[\![l : x := a]\!] &= \emptyset \\
\hat{\mathcal{C}}_f[\![\varsigma_0; \varsigma_1]\!] &= \hat{\mathcal{C}}_f[\![\varsigma_0]\!] \cup \hat{\mathcal{C}}_f[\![\varsigma_1]\!] \\
\hat{\mathcal{C}}_f[\![l : \textbf{assert}(\beta)]\!] &= \emptyset \\
\hat{\mathcal{C}}_f[\![l : \textbf{if } b \textbf{ then } \{\varsigma_0\} \textbf{ else } \{\varsigma_1\}]\!] &= \hat{\mathcal{C}}_f[\![\varsigma_0]\!] \cup \hat{\mathcal{C}}_f[\![\varsigma_1]\!] \\
\hat{\mathcal{C}}_f[\![l : \textbf{while } b \textbf{ do } \{\varsigma\}]\!] &= \hat{\mathcal{C}}_f[\![\varsigma]\!] \\
\hat{\mathcal{C}}_f[\![l : \textbf{call}(y)]\!] &= \{y\}.
\end{aligned}$$

A.1.3 Labels

The set of functions used for collecting labels $\mathbb{L}$ from extended arithmetic expressions, extended boolean expressions and extended commands.

Definition A.9. Function $\hat{\mathcal{C}}_{l_a} : \hat{\mathbb{E}}_a \to \mathcal{P}(\mathbb{L})$, returning the set of labels used in an extended arithmetic expression $\hat{\mathbb{E}}_a$, is defined by structural induction on extended arithmetic expressions:

$$\begin{aligned}
\hat{\mathcal{C}}_{l_a}[\![n]\!] &= \emptyset \\
\hat{\mathcal{C}}_{l_a}[\![at(x,l)]\!] &= \{l\} \\
\hat{\mathcal{C}}_{l_a}[\![\alpha_0 \ opa \ \alpha_1]\!] &= \hat{\mathcal{C}}_{l_a}[\![\alpha_0]\!] \cup \hat{\mathcal{C}}_{l_a}[\![\alpha_1]\!].
\end{aligned}$$

Definition A.10. Function $\hat{\mathcal{C}}_{l_b} : \hat{\mathbb{E}}_b \to \mathcal{P}(\mathbb{L})$, returning the set of labels used in an extended boolean expression $\hat{\mathbb{E}}_b$, is defined by structural induction on extended boolean expressions:

$$\begin{aligned}
\hat{\mathcal{C}}_{l_b}[\![true]\!] &= \emptyset \\
\hat{\mathcal{C}}_{l_b}[\![false]\!] &= \emptyset \\
\hat{\mathcal{C}}_{l_b}[\![\alpha_0 \ opb \ \alpha_1]\!] &= \hat{\mathcal{C}}_{l_a}[\![\alpha_0]\!] \cup \hat{\mathcal{C}}_{l_a}[\![\alpha_1]\!] \\
\hat{\mathcal{C}}_{l_b}[\![\beta_0 \ opl \ \beta_1]\!] &= \hat{\mathcal{C}}_{l_b}[\![\beta_0]\!] \cup \hat{\mathcal{C}}_{l_b}[\![\beta_1]\!] \\
\hat{\mathcal{C}}_{l_b}[\![\neg\beta]\!] &= \hat{\mathcal{C}}_{l_b}[\![\beta]\!] \\
\hat{\mathcal{C}}_{l_b}[\![p^n(\alpha_1, .., \alpha_n)]\!] &= \hat{\mathcal{C}}_{l_a}[\![\alpha_0]\!] \cup .. \cup \hat{\mathcal{C}}_{v_a}[\![\alpha_n]\!].
\end{aligned}$$

Definition A.11. Function $\hat{\mathcal{C}}_{l_c} : \hat{\mathbb{C}} \to \mathcal{P}(\mathbb{L})$, returning the set of labels defined in an extended command $\hat{\mathbb{C}}$, is defined by structural induction on extended commands:

$$\begin{aligned}
\hat{\mathcal{C}}_{l_c}[\![l : \textbf{skip}]\!] &= \{l\} \\
\hat{\mathcal{C}}_{l_c}[\![l : x := a]\!] &= \{l\} \\
\hat{\mathcal{C}}_{l_c}[\![\varsigma_0; \varsigma_1]\!] &= \hat{\mathcal{C}}_{l_c}[\![\varsigma_0]\!] \cup \hat{\mathcal{C}}_{l_c}[\![\varsigma_1]\!] \\
\hat{\mathcal{C}}_{l_c}[\![l : \textbf{assert}(\beta)]\!] &= \{l\} \\
\hat{\mathcal{C}}_{l_c}[\![l : \textbf{if } b \textbf{ then } \{\varsigma_0\} \textbf{ else } \{\varsigma_1\}]\!] &= \hat{\mathcal{C}}_{l_c}[\![\varsigma_0]\!] \cup \hat{\mathcal{C}}_{l_c}[\![\varsigma_1]\!] \cup \{l\} \\
\hat{\mathcal{C}}_{l_c}[\![l : \textbf{while } b \textbf{ do } \{\varsigma\}]\!] &= \hat{\mathcal{C}}_{l_c}[\![\varsigma]\!] \cup \{l\} \\
\hat{\mathcal{C}}_{l_c}[\![l : \textbf{call}(y)]\!] &= \{l\}.
\end{aligned}$$

A.1.4 Tags

Tags

The set of functions for collecting tags $\mathbb{T}$ from relational (extended) arithmetic expressions, relational (extended) boolean expressions.

Definition A.12. Function $\tilde{\mathcal{C}}_{t_a} : \tilde{\mathbb{E}}_a \to \mathcal{P}(\mathbb{T})$, returning the set of tags used in a relational arithmetic expression $\tilde{\mathbb{E}}_a$, is defined by structural induction on relational arithmetic expressions:

$$\begin{aligned}
\tilde{\mathcal{C}}_{t_a}[\![n]\!] &= \emptyset \\
\tilde{\mathcal{C}}_{t_a}[\![x\langle t\rangle]\!] &= \{t\} \\
\tilde{\mathcal{C}}_{t_a}[\![\tilde{a}_0 \ opa \ \tilde{a}_1]\!] &= \tilde{\mathcal{C}}_{t_a}[\![\tilde{a}_0]\!] \cup \tilde{\mathcal{C}}_{t_a}[\![\tilde{a}_1]\!]
\end{aligned}$$

Definition A.13. Function $\tilde{\mathcal{C}}_{t_b} : \tilde{\mathbb{E}}_b \to \mathcal{P}(\mathbb{T})$, returning the set of tags used in a relational boolean expression $\tilde{\mathbb{E}}_b$, is defined by structural induction on relational boolean expressions:

$$\begin{aligned}
\tilde{\mathcal{C}}_{t_b}[\![true]\!] &= \emptyset \\
\tilde{\mathcal{C}}_{t_b}[\![false]\!] &= \emptyset \\
\tilde{\mathcal{C}}_{t_b}[\![\tilde{a}_0 \ opb \ \tilde{a}_1]\!] &= \tilde{\mathcal{C}}_{t_a}[\![\tilde{a}_0]\!] \cup \tilde{\mathcal{C}}_{t_a}[\![\tilde{a}_1]\!] \\
\tilde{\mathcal{C}}_{t_b}[\![\tilde{b}_0 \ opl \ \tilde{b}_1]\!] &= \tilde{\mathcal{C}}_{t_b}[\![\tilde{b}_0]\!] \cup \tilde{\mathcal{C}}_{t_b}[\![\tilde{b}_1]\!] \\
\tilde{\mathcal{C}}_{t_b}[\![\neg\tilde{b}]\!] &= \tilde{\mathcal{C}}_{t_b}[\![\tilde{b}]\!].
\end{aligned}$$

Definition A.14. Function $\tilde{\hat{\mathcal{C}}}_{t_a} : \tilde{\hat{\mathbb{E}}}_a \to \mathcal{P}(\mathbb{T})$, returning the set of tags used in a relational extended arithmetic expression $\tilde{\hat{\mathbb{E}}}_a$, is defined by structural induction on relational extended boolean expressions:

$$\begin{aligned}
\tilde{\hat{\mathcal{C}}}_{t_a}[\![n]\!] &= \emptyset \\
\tilde{\hat{\mathcal{C}}}_{t_a}[\![at(x,l)\langle t\rangle]\!] &= \{t\} \\
\tilde{\hat{\mathcal{C}}}_{t_a}[\![\alpha_0 \ opa \ \alpha_1]\!] &= \tilde{\hat{\mathcal{C}}}_{t_a}[\![\alpha_0]\!] \cup \tilde{\hat{\mathcal{C}}}_{t_a}[\![\alpha_1]\!].
\end{aligned}$$

Definition A.15. Function $\tilde{\hat{\mathcal{C}}}_{t_b} : \tilde{\hat{\mathbb{E}}}_b \to (\mathbb{T} \to \mathcal{P}(\mathbb{T}))$, returning the set of tags used in a relational extended boolean expression $\tilde{\hat{\mathbb{E}}}_b$, is defined by structural induction on relational exended boolean expressions:

$$\begin{aligned}
\tilde{\hat{\mathcal{C}}}_{t_b}[\![true]\!] &= \emptyset \\
\tilde{\hat{\mathcal{C}}}_{t_b}[\![false]\!] &= \emptyset \\
\tilde{\hat{\mathcal{C}}}_{t_b}[\![\alpha_0 \ opb \ \alpha_1]\!] &= \tilde{\hat{\mathcal{C}}}_{t_a}[\![\alpha_0]\!] \cup \tilde{\hat{\mathcal{C}}}_{t_a}[\![\alpha_1]\!] \\
\tilde{\hat{\mathcal{C}}}_{t_b}[\![\beta_0 \ opl \ \beta_1]\!] &= \tilde{\hat{\mathcal{C}}}_{t_b}[\![\beta_0]\!] \cup \tilde{\hat{\mathcal{C}}}_{t_b}[\![\beta_1]\!] \\
\tilde{\hat{\mathcal{C}}}_{t_b}[\![\neg\beta]\!] &= \tilde{\hat{\mathcal{C}}}_{t_b}[\![\beta]\!] \\
\tilde{\hat{\mathcal{C}}}_{t_b}[\![p^n(\alpha_1,..,\alpha_n)]\!] &= \tilde{\hat{\mathcal{C}}}_{t_a}[\![\alpha_0]\!] \cup .. \cup \tilde{\hat{\mathcal{C}}}_{t_a}[\![\alpha_n]\!].
\end{aligned}$$

Locations

The sets of functions for collecting locations $\mathbb{X}$ from relational (extended) arithmetic expressions and relational (extended) boolean expressions, associated to a given tag.

Definition A.16. Function $\tilde{\mathcal{C}}_{v_a} : \tilde{\mathbb{E}}_a \to (\mathbb{T} \to \mathcal{P}(\mathbb{X}))$, returning the set of variables used in a relational arithmetic expression $\tilde{\mathbb{E}}_a$ for a given tag, is defined by structural induction on relational arithmetic expressions:

$$\begin{aligned}
\tilde{\mathcal{C}}_{v_a}[\![n]\!]t' &= \emptyset \\
\tilde{\mathcal{C}}_{v_a}[\![x\langle t\rangle]\!]t' &= \{x\} \qquad \text{if } t = t' \\
\tilde{\mathcal{C}}_{v_a}[\![x\langle t\rangle]\!]t' &= \emptyset \qquad \text{if } t \neq t' \\
\tilde{\mathcal{C}}_{v_a}[\![\tilde{a}_0 \ opa \ \tilde{a}_1]\!]t' &= \tilde{\mathcal{C}}_{v_a}[\![\tilde{a}_0]\!]t' \cup \tilde{\mathcal{C}}_{v_a}[\![\tilde{a}_1]\!]t'.
\end{aligned}$$

Definition A.17. Function $\tilde{\mathcal{C}}_{v_b} : \tilde{\mathbb{E}}_b \to (\mathbb{T} \to \mathcal{P}(\mathbb{X}))$, returning the set of variables used in a relational boolean expression $\tilde{\mathbb{E}}_b$ for a given tag, is defined by structural induction on relational boolean expressions:

$$\begin{aligned}
\tilde{\mathcal{C}}_{v_b}[\![true]\!]t' &= \emptyset \\
\tilde{\mathcal{C}}_{v_b}[\![false]\!]t' &= \emptyset \\
\tilde{\mathcal{C}}_{v_b}[\![\tilde{a}_0 \ opb \ \tilde{a}_1]\!]t' &= \tilde{\mathcal{C}}_{v_a}[\![\tilde{a}_0]\!]t' \cup \tilde{\mathcal{C}}_{v_a}[\![\tilde{a}_1]\!]t' \\
\tilde{\mathcal{C}}_{v_b}[\![\tilde{b}_0 \ opl \ \tilde{b}_1]\!]t' &= \tilde{\mathcal{C}}_{v_b}[\![\tilde{b}_0]\!]t' \cup \tilde{\mathcal{C}}_{v_b}[\![\tilde{b}_1]\!]t' \\
\tilde{\mathcal{C}}_{v_b}[\![\neg\tilde{b}]\!]t' &= \tilde{\mathcal{C}}_{v_b}[\![\tilde{b}]\!]t'.
\end{aligned}$$

Definition A.18. Function $\tilde{\hat{\mathcal{C}}}_{v_a} : \tilde{\hat{\mathbb{E}}}_a \to (\mathbb{T} \to \mathcal{P}(\mathbb{X}))$, returning the set of variables used in a relational extended arithmetic expression $\tilde{\hat{\mathbb{E}}}_a$ for a given tag, is defined by structural induction on relational extended arithmetic expressions:

$$\begin{aligned}
\tilde{\hat{\mathcal{C}}}_{v_a}[\![n]\!]t' &= \emptyset \\
\tilde{\hat{\mathcal{C}}}_{v_a}[\![at(x,l)\langle t\rangle]\!]t' &= \{x\} \qquad \text{if } t = t' \\
\tilde{\hat{\mathcal{C}}}_{v_a}[\![at(x,l)\langle t\rangle]\!]t' &= \emptyset \qquad \text{if } t \neq t' \\
\tilde{\hat{\mathcal{C}}}_{v_a}[\![\alpha_0 \ opa \ \alpha_1]\!]t' &= \tilde{\hat{\mathcal{C}}}_{v_a}[\![\alpha_0]\!]t' \cup \tilde{\hat{\mathcal{C}}}_{v_a}[\![\alpha_1]\!]t'.
\end{aligned}$$

Definition A.19. Function $\tilde{\hat{\mathcal{C}}}_{v_b} : \tilde{\hat{\mathbb{E}}}_b \to (\mathbb{T} \to \mathcal{P}(\mathbb{X}))$, returning the set of variables used in a relational extended boolean expression $\tilde{\hat{\mathbb{E}}}_b$ for a given tag, is defined by structural induction on relational extended boolean expressions:

$$\begin{aligned}
\tilde{\hat{\mathcal{C}}}_{v_b}[\![true]\!]t' &= \emptyset \\
\tilde{\hat{\mathcal{C}}}_{v_b}[\![false]\!]t' &= \emptyset \\
\tilde{\hat{\mathcal{C}}}_{v_b}[\![\alpha_0 \ opb \ \alpha_1]\!]t' &= \tilde{\hat{\mathcal{C}}}_{v_a}[\![\alpha_0]\!]t' \cup \tilde{\hat{\mathcal{C}}}_{v_a}[\![\alpha_1]\!]t' \\
\tilde{\hat{\mathcal{C}}}_{v_b}[\![\beta_0 \ opl \ \beta_1]\!]t' &= \tilde{\hat{\mathcal{C}}}_{v_b}[\![\beta_0]\!]t' \cup \tilde{\hat{\mathcal{C}}}_{v_b}[\![\beta_1]\!]t' \\
\tilde{\hat{\mathcal{C}}}_{v_b}[\![\neg\beta]\!]t' &= \tilde{\hat{\mathcal{C}}}_{v_b}[\![\beta]\!]t' \\
\tilde{\hat{\mathcal{C}}}_{v_b}[\![p^n(\alpha_1,..,\alpha_n)]\!]t' &= \tilde{\hat{\mathcal{C}}}_{v_a}[\![\alpha_0]\!]t' \cup .. \cup \tilde{\hat{\mathcal{C}}}_{v_a}[\![\alpha_n]\!]t'.
\end{aligned}$$

Labels

The sets of functions for collecting labels $\mathbb{L}$ from relational extended arithmetic expressions and relational extended boolean expressions, associated to a given tag.

Definition A.20. Function $\tilde{\hat{\mathcal{C}}}_{l_a} : \tilde{\hat{\mathbb{E}}}_a \to (\mathbb{T} \to \mathcal{P}(\mathbb{L}))$, returning the set of labels used in a relational extended arithmetic expression $\tilde{\hat{\mathbb{E}}}_a$ for a given tag, is defined by structural induction on relalational extended arithmetic expressions:

$$\begin{aligned}
\tilde{\hat{\mathcal{C}}}_{l_a}[\![n]\!]t' &= \emptyset \\
\tilde{\hat{\mathcal{C}}}_{l_a}[\![at(x,l)\langle t\rangle]\!]t' &= \{l\} \qquad \text{if } t = t' \\
\tilde{\hat{\mathcal{C}}}_{l_a}[\![at(x,l)\langle t\rangle]\!]t' &= \emptyset \qquad \text{if } t \neq t' \\
\tilde{\hat{\mathcal{C}}}_{l_a}[\![\alpha_0 \ opa \ \alpha_1]\!]t' &= \tilde{\hat{\mathcal{C}}}_{l_a}[\![\alpha_0]\!]t' \cup \tilde{\hat{\mathcal{C}}}_{l_a}[\![\alpha_1]\!]t'.
\end{aligned}$$

Definition A.21. Function $\tilde{\hat{\mathcal{C}}}_{l_b} : \tilde{\hat{\mathbb{E}}}_b \to (\mathbb{T} \to \mathcal{P}(\mathbb{L}))$, returning the set of labels used in a relational extended boolean expression $\tilde{\hat{\mathbb{E}}}_b$ for a given tag, is defined by structural induction on relational extended boolean expressions:

$$\begin{aligned}
\tilde{\hat{\mathcal{C}}}_{l_b}[\![true]\!]t' &= \emptyset \\
\tilde{\hat{\mathcal{C}}}_{l_b}[\![false]\!]t' &= \emptyset \\
\tilde{\hat{\mathcal{C}}}_{l_b}[\![\alpha_0 \ opb \ \alpha_1]\!]t' &= \tilde{\hat{\mathcal{C}}}_{l_a}[\![\alpha_0]\!]t' \cup \tilde{\hat{\mathcal{C}}}_{l_a}[\![\alpha_1]\!]t' \\
\tilde{\hat{\mathcal{C}}}_{l_b}[\![\beta_0 \ opl \ \beta_1]\!]t' &= \tilde{\hat{\mathcal{C}}}_{l_b}[\![\beta_0]\!]t' \cup \tilde{\hat{\mathcal{C}}}_{l_b}[\![\beta_1]\!]t' \\
\tilde{\hat{\mathcal{C}}}_{l_b}[\![\neg\beta]\!]t' &= \tilde{\hat{\mathcal{C}}}_{l_b}[\![\beta]\!]t' \\
\tilde{\hat{\mathcal{C}}}_{l_b}[\![p^n(\alpha_1,..,\alpha_n)]\!]t' &= \tilde{\hat{\mathcal{C}}}_{l_a}[\![\alpha_0]\!]t' \cup .. \cup \tilde{\hat{\mathcal{C}}}_{l_a}[\![\alpha_n]\!]t'.
\end{aligned}$$

A.2 Unique labels

The following section presents the functions $\hat{\mathcal{U}}$, returning the set of labels used in an extended command $\hat{\mathbb{C}}$. The function returns $\perp$ if there are duplicated labels in the command. Function $lift_u$ (defined in Section 5.1.4) is used to merge two sets of labels, and returns $\perp$ if the intersection of the sets is not empty.

Definition A.22. Function $\hat{\mathcal{U}} : \hat{\mathbb{C}} \to \mathcal{P}(\mathbb{L})_\perp$, returning the set of labels used in an extended command $\hat{\mathbb{C}}$ or $\perp$ if there are duplicated labels, is defined by structural induction on extended commands:

$$\begin{aligned}
\hat{\mathcal{U}}[\![l : \textbf{skip}]\!] &= \{l\} \\
\hat{\mathcal{U}}[\![l : x := a]\!] &= \{l\} \\
\hat{\mathcal{U}}[\![\varsigma_0; \varsigma_1]\!] &= \mathit{lift}_u(\hat{\mathcal{U}}[\![\varsigma_0]\!], \hat{\mathcal{U}}[\![\varsigma_1]\!]) \\
\hat{\mathcal{U}}[\![l : \textbf{assert}(\beta)]\!] &= \{l\} \\
\hat{\mathcal{U}}[\![l : \textbf{if } b \textbf{ then } \{\varsigma_0\} \textbf{ else } \{\varsigma_1\}]\!] &= \mathit{lift}_u(\{l\}, \mathit{lift}_u(\hat{\mathcal{U}}[\![\varsigma_0]\!], \hat{\mathcal{U}}[\![\varsigma_1]\!])) \\
\hat{\mathcal{U}}[\![l : \textbf{call}(y)]\!] &= \{l\} \\
\hat{\mathcal{U}}[\![l : \textbf{while } b \textbf{ do } \{\varsigma\}]\!] &= \mathit{lift}_u(\{l\}, \hat{\mathcal{U}}[\![\varsigma]\!]).
\end{aligned}$$

Note that function $\hat{\mathcal{U}}$ is not collecting labels in called procedures.

A.3 Renaming Functions

The following sections presents the set of functions used for renaming tags and locations in the case of relational arithmetic expression and relational boolean expression.

A.3.1 Location

The following sections presents the set of functions used for renaming a location into a given location for a given tag, in the case of relational arithmetic expression and relational boolean expression.

Definition A.23. Function $\tilde{\mathcal{R}}_{v_a} : \tilde{\mathbb{E}}_a \to (\mathbb{T} \times \mathbb{X} \times \mathbb{X} \to \tilde{\mathbb{E}}_a)$, renaming a location in a relational arithmetic expression $\tilde{\mathbb{E}}_a$ into a given location for a given tag, is defined by structural induction on relational arithmetic expressions:

$$\begin{aligned}
\tilde{\mathcal{R}}_{v_a}[\![n]\!](t', x', x'') &= [\![n]\!] \\
\tilde{\mathcal{R}}_{v_a}[\![x\langle t\rangle]\!](t', x', x'') &= \begin{cases} [\![x\langle t\rangle]\!] & \text{if } x \neq x' \text{ or } t \neq t' \\ [\![x''\langle t\rangle]\!] & \text{if } x = x' \text{ and } t = t' \end{cases} \\
\tilde{\mathcal{R}}_{v_a}[\![\tilde{a}_0 \ \mathit{opa} \ \tilde{a}_1]\!](t', x', x'') &= [\![\tilde{\mathcal{R}}_{v_a}[\![\tilde{a}_0]\!](t', x', x'') \ \mathit{opa} \ \tilde{\mathcal{R}}_{v_a}[\![\tilde{a}_1]\!](t', x', x'')]\!].
\end{aligned}$$

Definition A.24. Function $\tilde{\mathcal{R}}_{v_b} : \tilde{\mathbb{E}}_b \to (\mathbb{T} \times \mathbb{X} \times \mathbb{X} \to \tilde{\mathbb{E}}_b)$, renaming a location in a relational boolean expression $\tilde{\mathbb{E}}_b$ into a given location for a given tag, is defined by structural induction on relational boolean expressions:

$$\begin{aligned}
\tilde{\mathcal{R}}_{v_b}[\![true]\!](t', x', x'') &= [\![true]\!] \\
\tilde{\mathcal{R}}_{v_b}[\![false]\!](t', x', x'') &= [\![false]\!] \\
\tilde{\mathcal{R}}_{v_b}[\![\tilde{a}_0 \ opb \ \tilde{a}_1]\!](t', x', x'') &= [\![\tilde{\mathcal{R}}_{v_a}[\![\tilde{a}_0]\!](t', x', x'') \ opb \ \tilde{\mathcal{R}}_{v_a}[\![\tilde{a}_1]\!](t', x', x'')]\!] \\
\tilde{\mathcal{R}}_{v_b}[\![\tilde{b}_0 \ opl \ \tilde{b}_1]\!](t', x', x'') &= [\![\tilde{\mathcal{R}}_{v_b}[\![\tilde{b}_0]\!](t', x', x'') \ opl \ \tilde{\mathcal{R}}_{v_b}[\![\tilde{b}_1]\!](t', x', x'')]\!] \\
\tilde{\mathcal{R}}_{v_b}[\![\neg \tilde{b}]\!]t' &= [\![\neg \tilde{\mathcal{R}}_{v_b}[\![\tilde{b}]\!]t']\!].
\end{aligned}$$

A.3.2 Tags

The following sections presents the set of functions used for renaming all tags or a tag into a given tag, in the case of relational arithmetic expression and relational boolean expression.

Definition A.25. Function $\tilde{\mathcal{R}}_{tt_a} : \tilde{\mathbb{E}}_a \to (\mathbb{T} \to \tilde{\mathbb{E}}_a)$, renaming the set of tags used in a relational arithmetic expression $\tilde{\mathbb{E}}_a$ into a given tag, is defined by structural induction on relational arithmetic expressions:

$$\begin{aligned}
\tilde{\mathcal{R}}_{tt_a}[\![n]\!]t' &= [\![n]\!] \\
\tilde{\mathcal{R}}_{tt_a}[\![x\langle t \rangle]\!]t' &= [\![x\langle t' \rangle]\!] \\
\tilde{\mathcal{R}}_{tt_a}[\![\tilde{a}_0 \ opa \ \tilde{a}_1]\!]t' &= [\![\tilde{\mathcal{R}}_{tt_a}[\![\tilde{a}_0]\!]t' \ opa \ \tilde{\mathcal{R}}_{tt_a}[\![\tilde{a}_1]\!]t']\!].
\end{aligned}$$

Definition A.26. Function $\tilde{\mathcal{R}}_{tt_b} : \tilde{\mathbb{E}}_b \to (\mathbb{T} \to \tilde{\mathbb{E}}_b)$, renaming the set of tags used in a relational boolean expression $\tilde{\mathbb{E}}_b$ into a given tag, is defined by structural induction on relational boolean expressions:

$$\begin{aligned}
\tilde{\mathcal{R}}_{tt_b}[\![true]\!]t' &= [\![true]\!] \\
\tilde{\mathcal{R}}_{tt_b}[\![false]\!]t' &= [\![false]\!] \\
\tilde{\mathcal{R}}_{tt_b}[\![\tilde{a}_0 \ opb \ \tilde{a}_1]\!]t' &= [\![\tilde{\mathcal{R}}_{tt_a}[\![\tilde{a}_0]\!]t' \ opb \ \tilde{\mathcal{R}}_{tt_a}[\![\tilde{a}_1]\!]t']\!] \\
\tilde{\mathcal{R}}_{tt_b}[\![\tilde{b}_0 \ opl \ \tilde{b}_1]\!]t' &= [\![\tilde{\mathcal{R}}_{tt_b}[\![\tilde{b}_0]\!]t' \ opl \ \tilde{\mathcal{R}}_{tt_b}[\![\tilde{b}_1]\!]t']\!] \\
\tilde{\mathcal{R}}_{tt_b}[\![\neg \tilde{b}]\!]t' &= [\![\neg \tilde{\mathcal{R}}_{tt_b}[\![\tilde{b}]\!]t']\!].
\end{aligned}$$

Definition A.27. Function $\tilde{\mathcal{R}}_{t_a} : \tilde{\mathbb{E}}_a \to (\mathbb{T} \times \mathbb{T} \to \tilde{\mathbb{E}}_a)$, renaming a tag in a relational arithmetic expression $\tilde{\mathbb{E}}_a$ into a given tag, is defined by structural induction on relational arithmetic expressions:

$$\begin{aligned}
\tilde{\mathcal{R}}_{t_a}[\![n]\!](t', t'') &= [\![n]\!] \\
\tilde{\mathcal{R}}_{t_a}[\![x\langle t\rangle]\!](t', t'') &= \begin{cases} [\![x\langle t\rangle]\!] & \text{if } t \neq t' \\ [\![x\langle t''\rangle]\!] & \text{if } t = t' \end{cases} \\
\tilde{\mathcal{R}}_{t_a}[\![\tilde{a}_0 \ opa \ \tilde{a}_1]\!](t', t'') &= [\![\tilde{\mathcal{R}}_{t_a}[\![\tilde{a}_0]\!](t', t'') \ opa \ \tilde{\mathcal{R}}_{t_a}[\![\tilde{a}_1]\!](t', t'')]\!].
\end{aligned}$$

Definition A.28. Function $\tilde{\mathcal{R}}_{t_b} : \tilde{\mathbb{E}}_b \to (\mathbb{T} \times \mathbb{T} \to \tilde{\mathbb{E}}_b)$, renaming a tag in a relational boolean expression $\tilde{\mathbb{E}}_b$ into a given tag, is defined by structural induction on relational boolean expressions:

$$\begin{aligned}
\tilde{\mathcal{R}}_{t_b}[\![true]\!](t', t'') &= [\![true]\!] \\
\tilde{\mathcal{R}}_{t_b}[\![false]\!](t', t'') &= [\![false]\!] \\
\tilde{\mathcal{R}}_{t_b}[\![\tilde{a}_0 \ opb \ \tilde{a}_1]\!]t' &= [\![\tilde{\mathcal{R}}_{t_a}[\![\tilde{a}_0]\!](t', t'') \ opb \ \tilde{\mathcal{R}}_{t_a}[\![\tilde{a}_1]\!](t', t'')]\!] \\
\tilde{\mathcal{R}}_{t_b}[\![\tilde{b}_0 \ opl \ \tilde{b}_1]\!]t' &= [\![\tilde{\mathcal{R}}_{t_b}[\![\tilde{b}_0]\!](t', t'') \ opl \ \tilde{\mathcal{R}}_{t_b}[\![\tilde{b}_1]\!](t', t'')]\!] \\
\tilde{\mathcal{R}}_{t_b}[\![\neg\tilde{b}]\!](t', t'') &= [\![\neg\tilde{\mathcal{R}}_{t_b}[\![\tilde{b}]\!](t', t'')]\!].
\end{aligned}$$

A.4 Delete Functions

The following sections present the set of functions used for removing all tags from relational (extended) arithmetic expression and relational (extended) boolean expression.

A.4.1 Tags

Definition A.29. Function $\tilde{\mathcal{D}}_{t_a} : \tilde{\mathbb{E}}_a \to \mathbb{E}_a$, deleting all tags used in a relational arithmetic expression $\tilde{\mathbb{E}}_a$, is defined by structural induction on relational arithmetic expressions:

$$\begin{aligned}
\tilde{\mathcal{D}}_{t_a}[\![n]\!] &= [\![n]\!] \\
\tilde{\mathcal{D}}_{t_a}[\![x\langle t\rangle]\!] &= [\![x]\!] \\
\tilde{\mathcal{D}}_{t_a}[\![\tilde{a}_0 \ opa \ \tilde{a}_1]\!] &= [\![\tilde{\mathcal{D}}_{t_a}[\![\tilde{a}_0]\!] \ opa \ \tilde{\mathcal{D}}_{t_a}[\![\tilde{a}_1]\!]]\!].
\end{aligned}$$

Definition A.30. Function $\tilde{\mathcal{D}}_{t_b} : \tilde{\mathbb{E}}_b \rightarrow \mathbb{E}_b$, deleting all tags used in a relational boolean expression $\tilde{\mathbb{E}}_b$, is defined by structural induction on relational boolean expressions:

$$\begin{aligned}
\tilde{\mathcal{D}}_{t_b}[\![true]\!] &= [\![true]\!] \\
\tilde{\mathcal{D}}_{t_b}[\![false]\!] &= [\![false]\!] \\
\tilde{\mathcal{D}}_{t_b}[\![\tilde{a}_0 \ opb \ \tilde{a}_1]\!] &= [\![\tilde{\mathcal{D}}_{t_a}[\![\tilde{a}_0]\!] \ opb \ \tilde{\mathcal{D}}_{t_a}[\![\tilde{a}_1]\!]]\!] \\
\tilde{\mathcal{D}}_{t_b}[\![\tilde{b}_0 \ opl \ \tilde{b}_1]\!] &= [\![\tilde{\mathcal{D}}_{t_b}[\![\tilde{b}_0]\!] \ opl \ \tilde{\mathcal{D}}_{t_b}[\![\tilde{b}_1]\!]]\!] \\
\tilde{\mathcal{D}}_{t_b}[\![\neg\tilde{b}]\!] &= [\![\neg\tilde{\mathcal{D}}_{t_b}[\![\tilde{b}]\!]]\!].
\end{aligned}$$

Definition A.31. Function $\tilde{\hat{\mathcal{D}}}_{t_a} : \tilde{\hat{\mathbb{E}}}_a \rightarrow \hat{\mathbb{E}}_a$, deleting all tags used in an relational extended arithmetic expressions $\hat{\mathbb{E}}_a$, is defined by structural induction on relational extended arithmetic expressions:

$$\begin{aligned}
\tilde{\hat{\mathcal{D}}}_{t_a}[\![n]\!] &= [\![n]\!] \\
\tilde{\hat{\mathcal{D}}}_{t_a}[\![at(x,l)\langle t\rangle]\!] &= [\![at(x,l)]\!] \\
\tilde{\hat{\mathcal{D}}}_{t_a}[\![\tilde{\alpha}_0 \ opa \ \tilde{\alpha}_1]\!] &= [\![\tilde{\hat{\mathcal{D}}}_{t_a}[\![\tilde{\alpha}_0]\!] \ opa \ \tilde{\hat{\mathcal{D}}}_{t_a}[\![\tilde{\alpha}_1]\!]]\!].
\end{aligned}$$

Definition A.32. Function $\tilde{\hat{\mathcal{D}}}_{t_b} : \tilde{\hat{\mathbb{E}}}_b \rightarrow \hat{\mathbb{E}}_b$, deleting all tags used in a relationl extended boolean expressions $\hat{\mathbb{E}}_b$, is defined by structural induction on relational extended boolean expressions:

$$\begin{aligned}
\tilde{\hat{\mathcal{D}}}_{t_b}[\![true]\!] &= [\![true]\!] \\
\tilde{\hat{\mathcal{D}}}_{t_b}[\![false]\!] &= [\![false]\!] \\
\tilde{\hat{\mathcal{D}}}_{t_b}[\![\tilde{\alpha}_0 \ opb \ \tilde{\alpha}_1]\!] &= [\![\tilde{\hat{\mathcal{D}}}_{t_a}[\![\tilde{\alpha}_0]\!] \ opb \ \tilde{\hat{\mathcal{D}}}_{t_a}[\![\tilde{\alpha}_1]\!]]\!] \\
\tilde{\hat{\mathcal{D}}}_{t_b}[\![\tilde{\beta}_0 \ opl \ \tilde{\beta}_1]\!] &= [\![\tilde{\hat{\mathcal{D}}}_{t_b}[\![\tilde{\beta}_0]\!] \ opl \ \tilde{\hat{\mathcal{D}}}_{t_b}[\![\tilde{\beta}_1]\!]]\!] \\
\tilde{\hat{\mathcal{D}}}_{t_b}[\![\neg\tilde{\beta}]\!] &= [\![\neg\tilde{\hat{\mathcal{D}}}_{t_b}[\![\tilde{\beta}]\!]]\!] \\
\tilde{\hat{\mathcal{D}}}_{t_b}[\![p^n(\tilde{\alpha}_1,..,\tilde{\alpha}_n)]\!] &= [\![p^n(\tilde{\hat{\mathcal{D}}}_{t_a}[\![\tilde{\alpha}_0]\!],..,\tilde{\hat{\mathcal{D}}}_{t_a}[\![\tilde{\alpha}_n]\!])]\!].
\end{aligned}$$

A.5 Add Functions

The following sections present the set of functions used for adding tags $\mathbb{T}$ to arithmetic expression and boolean expression.

A.5.1 Tags

Definition A.33. Function $\tilde{\mathcal{A}}_{t_a} : \mathbb{E}_a \to \tilde{\mathbb{E}}_a$, adding a tag in an arithmetic expression $\mathbb{E}_a$ to get a relational arithmetic expression $\tilde{\mathbb{E}}_a$, is defined by structural induction on arithmetic expressions:

$$\begin{aligned}
\tilde{\mathcal{A}}_{t_a}[\![n]\!] &= [\![n]\!] \\
\tilde{\mathcal{A}}_{t_a}[\![x]\!]t &= [\![x\langle t\rangle]\!] \\
\tilde{\mathcal{A}}_{t_a}[\![a_0 \ opa \ a_1]\!]t &= [\![\tilde{\mathcal{A}}_{t_a}[\![a_0]\!]t \ opa \ \tilde{\mathcal{A}}_{t_a}[\![a_1]\!]t]\!].
\end{aligned}$$

Definition A.34. Function $\tilde{\mathcal{A}}_{t_b} : \mathbb{E}_b \to \tilde{\mathbb{E}}_b$, adding a tag in a boolean expression $\mathbb{E}_b$ to get a relational boolean expression $\tilde{\mathbb{E}}_b$, is defined by structural induction on boolean expressions:

$$\begin{aligned}
\tilde{\mathcal{A}}_{t_b}[\![true]\!]t &= [\![true]\!] \\
\tilde{\mathcal{A}}_{t_b}[\![false]\!]t &= [\![false]\!] \\
\tilde{\mathcal{A}}_{t_b}[\![a_0 \ opb \ a_1]\!]t &= [\![\tilde{\mathcal{A}}_{t_a}[\![a_0]\!]t \ opb \ \tilde{\mathcal{A}}_{t_a}[\![a_1]\!]t]\!] \\
\tilde{\mathcal{A}}_{t_b}[\![b_0 \ opl \ b_1]\!]t &= [\![\tilde{\mathcal{A}}_{t_b}[\![b_0]\!]t \ opl \ \tilde{\mathcal{A}}_{t_b}[\![b_1]\!]t]\!] \\
\tilde{\mathcal{A}}_{t_b}[\![\neg b]\!]t &= [\![\neg\tilde{\mathcal{A}}_{t_b}[\![b]\!]t]\!].
\end{aligned}$$

Appendix B

Translation Function $\hat{\mathcal{T}}_C$

In the following sections, we define function $\hat{\mathcal{T}}_c$ for the translation of commands **if** and **while** into a formula q.

B.1 Command if

In this section we focus on the definition of function $\hat{\mathcal{T}}_c$ for command **if**.

Definition B.1. Function $\hat{\mathcal{T}}_c : \hat{\mathbb{C}} \to (\hat{\Xi} \times V \to V)$, translating a command $\hat{\mathbb{C}}$ (case of condition) into an formula of $\mathbb{Q}$, is defined by:

$$\hat{\mathcal{T}}_c[\![l : \textbf{if } b \textbf{ then } \{\varsigma_0\} \textbf{ else } \{\varsigma_1\}]\!](q, u_{\mathbb{Q}}, m, (\theta, \delta)) =$$
$$([\![q \wedge (q' \Rightarrow (q_{\varsigma_1} \wedge m' = m_{\varsigma_1})) \wedge (\neg q' \Rightarrow (q_{\varsigma_2} \wedge m' = m_{\varsigma_2}))]\!], u_{\mathbb{Q}}{}', m', (\theta[l \leftarrow m], \delta_{\varsigma_2}))$$

where

(i) $(q', \delta') = \mathcal{T}_b[\![b]\!](m, \delta)$,

(ii) $m' = \mathcal{N}_{v_m}$,

(iii) $(q_{\varsigma_1}, u_{\mathbb{Q}_{\varsigma_1}}, m_{\varsigma_1}, (\_, \delta_{\varsigma_1})) = \hat{\mathcal{T}}_c[\![\varsigma_0]\!](T, \emptyset, m, (\theta[l \leftarrow m], \delta'))$,

(iv) $(q_{\varsigma_2}, u_{\mathbb{Q}_{\varsigma_2}}, m_{\varsigma_2}, (\_, \delta_{\varsigma_2})) = \hat{\mathcal{T}}_c[\![\varsigma_1]\!](T, \emptyset, m, (\theta[l \leftarrow m], \delta_{\varsigma_1}))$,

(v) $u_{\mathbb{Q}}{}' = \begin{array}{l} \{q_u \mid q_u = [\![q \wedge q' \Rightarrow q_t]\!] \text{ and } q_t \in u_{\mathbb{Q}_{\varsigma_1}}\} \cup \\ \{q_u \mid q_u = [\![q \wedge \neg q' \Rightarrow q_t]\!] \text{ and } q_t \in u_{\mathbb{Q}_{\varsigma_2}}\} \cup u_{\mathbb{Q}}. \end{array}$

(i) First, we translate the condition b into a formula q'.

(ii) Then, we define a new array m', using function $\mathcal{N}_{v_m}$, which models the state after the command.

(iii) Then, we translate the command for the case where b is true. Notice that the translation function $\hat{\mathcal{T}}_c$ is called with the true formula as parameter. We get a formula q_{ς_1} and a set of sub-formulas $u_{\mathbb{Q}_{\varsigma_1}}$.

(iv) Then, we translate the command for the case where b is false. Notice that the translation function $\hat{\mathcal{T}}_c$ is called with the true formula as parameter. We get a formula q_{ς_2} and a set of sub-formulas $u_{\mathbb{Q}_{\varsigma_2}}$.

(v) Then, we add q and respectively q' and $\neg q'$ to the sub proof. Since the verification conditions for each branch are generated only from the fact true (T), we add the strongest postcondition generated up to this point (q) and that b is true or false (q' and $\neg q'$) to get the complete strongest postcondition.

Finally, we add to q the fact that we have either q_{ς_1} or q_{ς_2}. We also add the fact that the array variable (m'), which models the memory state after the evaluation of the command, is either equal to the array variable which models the state after the command evaluation when the condition is true (m_{ς_1}), or the array variable which models the state after the command evaluation when the condition is false (m_{ς_2}).

Example B.1. We consider the following program composed of assignments and a condition:

$$
\begin{array}{ll}
l_1: & \textbf{if } x_1 > 1 \textbf{ then } \{ \\
 & \quad l_2 : x_2 := 2 \\
 & \} \textbf{ else } \{ \\
 & \quad l_3 : x_2 := 3 \\
 & \}; \\
l_4: & x_2 := x_2 + 5
\end{array}
$$

Using function $\hat{\mathcal{T}}_c$, we get the following formula:

$$
\begin{array}{ll}
l_1: & \textbf{if } x_1 > 1 \textbf{ then } \{ \\
 & \quad l_2 : x_2 := 2 \\
 & \} \textbf{ else } \{ \\
 & \quad l_3 : x_2 := 3 \\
 & \}; \\
l_4: & x_2 := x_2 + 5
\end{array}
\qquad
\begin{array}{c}
(m_{l_1}[1] > 1 \Rightarrow \\
(m_{next\ l_2} = m_{l_1}[2 \leftarrow 2] \wedge m_{l_4} = m_{next\ l_2})) \\
\wedge \\
(\neg(m_{l_1}[1] > 1) \Rightarrow \\
(m_{next\ l_3} = m_{l_1}[2 \leftarrow 3] \wedge m_{l_4} = m_{next\ l_3})) \\
\wedge m_{next} = m_{l_4}[2 \leftarrow m_{l_4}[2] + 5]
\end{array}
$$

We can recognize the two parts of the formula corresponding to the two branches of the condition (one with $m_{l_1}[1] > 1$, and the other with $\neg m_{l_1}[1] > 1$).

B.2 Command while

In this section we focus on the definition of function $\hat{\mathcal{T}}_c$ for command **while**. As mentioned in Section 3.3, loop invariants are used to summarize the behavior of a loop. Moreover, frame rules can also be defined for loops, as shown in Section 2.1. Thus, we define a new syntax for loop **while** b **inv** $(\beta, u_{\mathbb{X}})$ **do** $\{\varsigma\}$, where we can define the loop invariant and the frame rule. As the loop invariant must hold at the beginning of each loop iteration, we define a reserved label $Here$, like label Pre and $Post$, to denote this state.

$$
\begin{array}{ll}
l_1: & \textbf{while } x_1 > 0 \textbf{ inv } (at(x_1, Here) > 0 \vee at(x_1, Here) = 0, x_1) \textbf{ do } \{ \\
 & \quad l_2 : x_1 := x_1 - 1 \\
 & \}
\end{array}
$$

We now define function $\hat{\mathcal{T}}_c$ for the case of loop.

Definition B.2. Function $\hat{\mathcal{T}}_c : \hat{\mathbb{C}} \to (\hat{\Xi} \times V \to V)$, translating a command $\hat{\mathbb{C}}$ (case of loop) into a formula of $\mathbb{Q}$, is defined by:

$$\hat{\mathcal{T}}_c[\![l : \textbf{while } b \textbf{ inv } \beta_I, \{x_1, ..., x_n\} \textbf{ do } \{\varsigma\}]\!](q, u_{\mathbb{Q}}, m, (\theta, \delta)) =$$

$$([\![q \wedge q_e \wedge q_i \wedge \neg q_b \wedge m' = m[e_1 \leftarrow v_1]...[e_n \leftarrow v_n]]\!], u_{\mathbb{Q}} \cup u_{\mathbb{Q}_e} \cup u_{\mathbb{Q}_p} \cup u_{\mathbb{Q}_f}, m', (\theta', \delta_n))$$

where

I.

(i) $(q_e, (\_, \delta_e)) = \hat{\mathcal{T}}_b[\![\beta_I]\!](\theta[l \leftarrow m][Here \leftarrow m], \delta)$

(ii) $u_{\mathbb{Q}_e} = \{q \Rightarrow q_e\}$,

II.

(i) $m' = \mathcal{N}_{v_m}$

(ii) $(q_b, \delta') = \mathcal{T}_b[\![b]\!](m', \delta_e)$,

(iii) $(q_i, (\theta', \delta'')) = \hat{\mathcal{T}}_b[\![\beta_I]\!](\theta[l \leftarrow m][Here \leftarrow m'], \delta')$,

(iv) $v_1 = \mathcal{N}_{v_n}, ..., v_n = \mathcal{N}_{v_n}$,

(v) $(\delta_1, e_1) = \mathit{lift}(\delta'', x_1), ..., (\delta_n, e_n) = \mathit{lift}(\delta_{n-1}, x_n)$,

III.

(i) $m_p = \mathcal{N}_{v_m}$

(ii) $(q'_b, \delta_p) = \mathcal{T}_b[\![b]\!](m_p, \delta_n)$,

(iii) $(q'_i, (\theta_p, \delta'_p)) = \hat{\mathcal{T}}_b[\![\beta_I]\!](\theta[l \leftarrow m][Here \leftarrow m_p], \delta_p)$,

(iv) $(q_\varsigma, u_{\mathbb{Q}_\varsigma}, m'_p, (\_, \delta''_p)) =$
$\hat{\mathcal{T}}_c[\![\varsigma]\!]([\![q \wedge q_e \wedge q'_i \wedge q'_b \wedge m_p = m[e_1 \leftarrow v_1]...[e_n \leftarrow v_n]]\!], \emptyset, m_p, (\theta_p, \delta'_p))$,

(v) $(q_p, \_)) = \hat{\mathcal{T}}_b[\![\beta_I]\!](\theta[l \leftarrow m][Here \leftarrow m'_p], \delta''_p)$,

(vi) $u_{\mathbb{Q}_p} = \{q_\varsigma \Rightarrow q_p\} \cup u_{\mathbb{Q}_\varsigma}$,

IV. $u_{\mathbb{Q}_f} = \hat{\mathcal{VC}}_{fl}(\varsigma_c, b_I, b, q, \{x_1, ..., x_n\})$

$$\hat{\mathcal{T}}_c[\![l : \textbf{while } b \textbf{ do } \{\varsigma\}]\!](q, u_{\mathbb{Q}}, m, (\theta, \delta)) = (q, u_{\mathbb{Q}}, m', (\theta[l \leftarrow m], \delta))$$
$$\text{where } m' = \mathcal{N}_{v_m}.$$

I. We verify that the invariant is established at loop entry:

(i) First, we translate the loop invariant β_I into a formula q_e.

(ii) Then, we define the verification condition $u_{\mathbb{Q}_e}$ stating that the strongest postcondition generated up to this point implies the loop invariant q_e.

We add $u_{\mathbb{Q}_e}$ to the set of verification conditions .

II. We assume the invariant, the negation of the loop condition and the frame rule after the loop:

- ***(i)*** First, we define a new array m' which models the state after the loop.
- ***(ii)*** Then, we translate the loop condition b into a formula q_b.
- ***(iii)*** Then, we translate the invariant β_I into a formula q_i, with the label $Here$ associated to m'.
- ***(iv)*** Then, for each assigned location, we define a new natural variable.
- ***(iv)*** Then, for each assigned location, we get the corresponding index in the array representing the memory state.

Finally, we add to q the invariant q_i, the negation of the condition $\neg q_b$ and the frame rule.

III. We verify that the invariant is preserved by the loop body:

- ***(i)*** First, we define a new array m_p which models the state for an arbitrary loop iteration.
- ***(ii)*** Then, we translate the loop condition b into a formula q_b.
- ***(iii)*** Then, we translate the invariant β_I into a formula q_i', with the label $Here$ associated to m_p.
- ***(iv)*** Then, we translate the body of the loop into a formula. Function $\hat{\mathcal{T}}_c$ is called with as strongest postcondition formula q, the formula corresponding to the invariant at loop entry, the formula corresponding to the invariant and the loop condition at state m_p and the frame rule. We get a strongest postcondition q_ς and a set of verification conditions $u_{\mathbb{Q}_\varsigma}$.
- ***(v)*** Then, we translate the invariant β_I into a formula q_p, with the label $Here$ associated to m_p'.
- ***(vi)*** Then, we define the verification condition $u_{\mathbb{Q}_p}$ stating that the strongest postcondition q_ς implies the loop invariant at the end of the loop body q_p.

We add to the set of verification conditions $u_{\mathbb{Q}_p}$ and $u_{\mathbb{Q}_c}$.

III. We verify the frame rule: As for the definition of function $\hat{\mathcal{T}}_c$, we assume the existence of a function $\hat{\mathcal{VC}}_{fl} : \hat{\mathbb{C}} \times \hat{\mathbb{E}}_b \times \mathbb{E}_b \times \mathbb{Q} \times \mathcal{P}(\mathbb{X}) \to \mathcal{P}(\mathbb{Q})$ that returns the set of verification conditions that must be valid in order for the frame rule of the loop $\mathcal{P}(\mathbb{X})$ to be valid.

Example B.2. We consider the example of function `loop` shown on Figure 2.3 returning always 10. We can define the equivalent program using the R-WHILE* syntax:

$\{\ true\ \}$

l_1 : **if** $x >= 10$ **then** {
  $l_2 : x := 10;$
} **else** {
  l_3 : **while** $x < 10$ **inv** $(at(x, Here) <= 10, x)$ **do** { $\{\ at(x, Post) = 10\ \}$
    $l_4 : x := x + 1;$
  }
}

Using function $\hat{\mathcal{VC}}_h$ (defined in Section 5.3.4), for generating the verification condition for a Hoare Triple, we get the following formulas (the formulas for the frame rule excluded):

- The main formula corresponding to the triple:

$$
\begin{gathered}
T \wedge \\
((m_{l_1}[1] >= 10 \Rightarrow (T \wedge m_{next\ l_2} = m_{l_1}[1 \leftarrow 10]) \wedge m_{Post} = m_{next\ l_2})) \\
\wedge \\
(\neg(m_{l_1}[1] >= 10) \Rightarrow \\
(T \wedge m_{l_1}[1] <= 10 \wedge m_{next\ l_3}[1] <= 10 \wedge \neg(m_{next\ l_3}[1] < 10) \wedge m_{next\ l_3} = m_{l_1}[1 \leftarrow v] \\
\wedge m_{Post} = m_{next\ l_3})) \\
\Rightarrow m_{Post}[1] = 10
\end{gathered}
$$

  We can recognize the two parts of the formula corresponding to the two branches of the condition (one with $m_{l_1}[1] >= 10$, and the other with $\neg m_{l_1}[1] >= 10$).

- A sub formula corresponding to the fact that the invariant is holding at loop entry:

$$T \wedge \neg(m_{l_1}[1] >= 10) \Rightarrow T \Rightarrow m_{l_1}[1] <= 10$$

- A sub formula corresponding to the fact that the loop invariant is preserved by the loop body:

$$
\begin{gathered}
\text{(* Pre-condition *)} \\
T \wedge \\
\text{(* Else *)} \\
\neg(m_{l_1}[1] >= 10) \Rightarrow \\
(T \wedge \\
\text{(*Loop invariant holds at loop entry *)} \\
m_{l_1}[1] <= 10 \wedge \\
\text{(* Loop invariant holds*)} \\
m_{l\_begin}[1] <= 10 \wedge \\
\text{(* Loop condition holds *)} \\
m_{l\_begin}[1] < 10 \wedge \\
\text{(* Frame rule*)} \\
m_{l\_begin}[1] = m_{l_1}[1 \leftarrow v] \wedge \\
\text{(* Assignment *)} \\
m_{l\_end} = m_{l\_begin}[1 \leftarrow m_{l\_begin}[1] + 1] \\
\Rightarrow m_{l\_end}[1] <= 10)
\end{gathered}
$$

Some similarity can be noted with the results, shown in Section 2.2, of WP for the same objective. We recognize the precondition ($0 <= x$ which is always true in our case), the fact that the boolean condition of the **if** is false (in green), the loop invariant holds at the beginning of the current loop step (in orange), the loop condition holds at the beginning of the current loop step (in violet) and the loop body, corresponding to an assignment (in purple), and finally the loop invariant holds at the end of the loop body (in blue). Note that the frame rule and the fact that the loop invariant holds at loop entry are not present in the results shown in Section 2.2 due to WP simplification reason. The initial assignment to local variable is not present in the above formula since we have no local variables.

Bibliography

[ABB+16] Wolfgang Ahrendt, Bernhard Beckert, Richard Bubel, Reiner Hähnle, Peter H. Schmitt, and Mattias Ulbrich, editors. *Deductive Software Verification - The KeY Book - From Theory to Practice*, volume 10001 of *Lecture Notes in Computer Science*. Springer, 2016.

[AdO09] Krzysztof R. Apt, Frank S. de Boer, and Ernst-Rüdiger Olderog. *Verification of Sequential and Concurrent Programs*. Texts in Computer Science. Springer, 2009.

[AGH+17] Timos Antonopoulos, Paul Gazzillo, Michael Hicks, Eric Koskinen, Tachio Terauchi, and Shiyi Wei. Decomposition instead of self-composition for proving the absence of timing channels. In *Proc. of the 38th Conference on Programming Language Design and Implementation (PLDI 2017)*, pages 362–375, 2017.

[AGM94] Samson Abramsky, Dov M. Gabbay, and T. S. E. Maibaum, editors. *Handbook of Logic in Computer Science (Vol. 3): Semantic Structures*. Oxford University Press, Inc., 1994.

[Bar11] Romain Bardou. *Vérification de programmes avec pointeurs à l'aide de régions et de permissions*. PhD thesis, Université Paris Sud, 2011.

[BBC13] Peter G. Bishop, Robin E. Bloomfield, and Lukasz Cyra. 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)*, pages 248–257, 2013.

[BBK+16] Bernhard Beckert, Thorsten Bormer, Michael Kirsten, Till Neuber, and Mattias Ulbrich. Automated verification for functional and relational properties of voting rules. In *Proc. of the 6th International Workshop on Computational Social Choice (COMSOC 2016)*, 2016.

[BBPS16] Jasmin Christian Blanchette, Sascha Böhme, Andrei Popescu, and Nicholas Smallbone. Encoding monomorphic and polymorphic types. *Logical Methods in Computer Science*, 12(4), 2016.

[BC04] Yves Bertot and Pierre Castéran. *Interactive Theorem Proving and Program Development - Coq'Art: The Calculus of Inductive Constructions*. Texts in Theoretical Computer Science. An EATCS Series. Springer, 2004.

[BCD+05] Michael Barnett, Bor-Yuh Evan Chang, Robert DeLine, Bart Jacobs, and K. Rustan M. Leino. Boogie: A modular reusable verifier for object-oriented programs. In *Proc. of the 4th International Symposium on Formal Methods for Components and Objects (FMCO 2005)*, pages 364–387, 2005.

[BCF+13] Patrick Baudin, Pascal Cuoq, Jean-Christophe Filliâtre, Claude Marché, Benjamin Monate, Yannick Moy, and Virgile Prevosto. *ACSL: ANSI/ISO C Specification Language*, 2013. `http://frama-c.com/acsl.html`.

[BCK11] Gilles Barthe, Juan Manuel Crespo, and César Kunz. Relational verification using product programs. In *Proc. of the 17th International Symposium on Formal Methods (FM 2011)*, pages 200–214, 2011.

[BCK16] Gilles Barthe, Juan Manuel Crespo, and César Kunz. Product programs and relational program logics. *Journal of Logical and Algebraic Methods in Programming*, 85(5):847–859, 2016.

[BDR11] Gilles Barthe, Pedro R. D'Argenio, and Tamara Rezk. Secure information flow by self-composition. *J. of Mathematical Structures in Computer Science*, 21(6):1207–1252, 2011.

[Ben04] Nick Benton. Simple relational correctness proofs for static analyses and program transformations. In *Proc. of the 41st Symposium on Principles of Programming Languages (POPL 2004)*, pages 14–25, 2004.

[BKG+18] Lionel Blatter, Nikolai Kosmatov, Pascale Le Gall, Virgile Prevosto, and Guillaume Petiot. 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)*, pages 44–62, 2018.

[BKGP17] Lionel Blatter, Nikolai Kosmatov, Pascale Le Gall, and Virgile Prevosto. 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)*, pages 391–397, 2017.

[BLS05] Mike Barnett, K. Rustan M. Leino, and Wolfram Schulte. The Spec# programming system: An overview. In *Proc. of the 2th International Workshop on Construction and Analysis of Safe, Secure, and Interoperable Smart Devices (CASSIS 2004)*, pages 49–69, 2005.

[BP13] Jasmin Christian Blanchette and Andrei Paskevich. TFF1: the TPTP typed first-order form with rank-1 polymorphism. In *Proc. of the 24th International Conference on Automated Deduction (CADE 2013)*, pages 414–420, 2013.

[Car94] Morgan Carroll. *Programming from Specifications (2Nd Ed.)*. Prentice Hall International, 1994.

[CC77] Patrick Cousot and Radhia Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Porc. of the Fourth Symposium on Principles of Programming Languages, (POPL 1977)*, pages 238–252, 1977.

[CES86] Edmund M. Clarke, E. Allen Emerson, and A. Prasad Sistla. Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 8(2):244–263, 1986.

[CMPP11] Pascal Cuoq, Benjamin Monate, Anne Pacalet, and Virgile Prevosto. Functional dependencies of C functions via weakest pre-conditions. *International Journal on Software Tools for Technology Transfer (STTT 2011)*, 13(5):405–417, 2011.

[Cok14] David R. Cok. OpenJML Software verification for Java 7 using JML, OpenJDK, and Eclipse. In *Proc. of the 1st Workshop on Formal Integrated Development Environment (F-IDE 2014)*, pages 79–92, 2014.

[CPR11] Byron Cook, Andreas Podelski, and Andrey Rybalchenko. Proving program termination. *Communications of the ACM*, 54(5):88–98, 2011.

[Dij68] Edsger Wybe Dijkstra. A constructive approach to the problem of program correctness. *BIT Numerical Mathematics*, 8(3):174–186, 1968.

[DKS13] Mickaël Delahaye, Nikolai Kosmatov, and Julien Signoles. Common specification language for static and dynamic analysis of C programs. In *Proc. of the 28th Annual ACM Symposium on Applied Computing (SAC 2013)*, pages 1230–1235, 2013.

[DM06] Adám Darvas and Peter Müller. Reasoning about method calls in JML specifications. *Journal of Object Technology*, 5(5):59–85, 2006.

[DS90] Edsger W. Dijkstra and Carel S. Scholten. *Predicate Calculus and Program Semantics*. Springer-Verlag, 1990.

[EMH18] Marco Eilers, Peter Müller, and Samuel Hitz. Modular product programs. In *Proc. of the 27th European Symposium on Programming (ESOP 2018)*, pages 502–529, 2018.

[FCG17] Gian Pietro Farina, Stephen Chong, and Marco Gaboardi. Relational symbolic execution. *Computing Research Repository*, abs/1711.08349, 2017.

[Flo67] Robert W. Floyd. Assigning meanings to programs. In *Proc. of Symposia in Applied Mathematics*, volume 19 (Mathematical Aspects of Computer Science), page 19–32, 1967.

[FM07] Jean-Christophe Filliâtre and Claude Marché. The why/krakatoa/caduceus platform for deductive program verification. In *Proc. of the 19th International Conference on Computer Aided Verification (CAV 2007)*, pages 173–177, 2007.

[FP13] Jean-Christophe Filliâtre and Andrei Paskevich. Why3 - where programs meet provers. In *Proc. of the 22nd European Symposium on Programming (ESOP 2013)*, pages 125–128, 2013.

[FS01] Cormac Flanagan and James B. Saxe. Avoiding exponential explosion: generating compact verification conditions. In *Proc. of the 28th Symposium on Principles of Programming Languages POPL 2001*, pages 193–205, 2001.

[Gor88] Michael J. C. Gordon. *Programming language theory and its implementation - applicative and imperative paradigms*. Prentice Hall International series in Computer Science. Prentice Hall, 1988.

[HKLR13] Chris Hawblitzel, Ming Kawaguchi, Shuvendu K. Lahiri, and Henrique Rebêlo. Towards modularly comparing programs using automated theorem provers. In *Proc. of the 24th International Conference on Automated Deduction (CADE 2013)*, pages 282–299, 2013.

[Hoa69] C. A. R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–580, 1969.

[JSP10] Bart Jacobs, Jan Smans, and Frank Piessens. A quick tour of the verifast program verifier. In *Proc. of the 8th Asian Symposium on Programming Languages and Systems (APLAS 2010)*, pages 304–311, 2010.

[K+14] Lauri Kasanen et al. *Into the Core: A look at Tiny Core Linux*. Electronic textbook, 2014. `http://tinycorelinux.net/corebook.pdf`.

[KCC+14] Johannes Kanig, Roderick Chapman, Cyrille Comar, Jérôme Guitton, Yannick Moy, and Emyr Rees. Explicit assumptions - A prenup for marrying static and dynamic program verification. In *Proc. of the 8th International Conferenceon Tests and Proofs (TAP 2014)*, pages 142–157, 2014.

[Kin76] James C. King. Symbolic execution and program testing. *Communications of the ACM*, 19(7):385–394, 1976.

[Kip14] Irvine Kip. *Assembly Language for x86 Processors*. Prentice Hall Press, 7th edition, 2014.

[KKP+15] Florent Kirchner, Nikolai Kosmatov, Virgile Prevosto, Julien Signoles, and Boris Yakobowski. Frama-C: A software analysis perspective. *Formal Aspects of Computing*, 27(3):573–609, 2015.

[KKU18] Moritz Kiefer, Vladimir Klebanov, and Mattias Ulbrich. Relational program reasoning using compiler IR - combining static verification and dynamic analysis. *J. of Automated Reasoning*, 60(3):337–363, 2018.

[Lio96] Jacques-Louis Lions. ARIANE 5 Flight 501 Failure, Report by the Inquiry Board. Technical report, European Space Agency, 1996.

[LM08] K. Rustan M. Leino and Peter Müller. Verification of equivalent-results methods. In *Proc. of the 17th European Symposium on Programming, (ESOP 2008)*, pages 307–321, 2008.

[LP13] K. Rustan M. Leino and Nadia Polikarpova. Verified calculations. In *Proc. of the 5th International Conference on Verified Software: Theories, Tools, Experiments (VSTTE 2013), Revised Selected Papers*, volume 8164, pages 170–190. Springer, 2013.

[LRL+00] Gary T. Leavens, Clyde Ruby, K. Rustan M. Leino, Erik Poll, and Bart Jacobs. JML: notations and tools supporting detailed design in Java. In *Proc. of the Conference on Object Oriented Programming Systems Languages and Applications (OOPSLA 2000)*, pages 105–106, 2000.

[LW14] K. Rustan M. Leino and Valentin Wüstholz. The dafny integrated development environment. In *Proc. of the 1st Workshop on Formal Integrated Development Environment (F-IDE 2014)*, pages 3–15, 2014.

[Mey97] Bertrand Meyer. *Object-Oriented Software Construction, 2nd Edition*. Prentice-Hall, 1997.

[Min15] Antoine Miné. École Normale Supérieure Paris, semantics and application to program verification, lecture notes: Denotational semantics, 2015. `https://www.di.ens.fr/~rival/semverif-2017/sem-05-denotational.pdf`. Last visited on 2019/18/01.

[Moy09] Yannick Moy. *Automatic Modular Static Safety Checking for C Programs*. PhD thesis, Université Paris Sud, 2009.

[PAC+18] Benjamin C. Pierce, Arthur Azevedo de Amorim, Chris Casinghino, Marco Gaboardi, Michael Greenberg, Cătălin Hriţcu, Vilhelm Sjöberg, and Brent Yorgey. *Logical Foundations*. Software Foundations series, volume 1. Electronic textbook, 2018. `http://www.cis.upenn.edu/~bcpierce/sf`.

[Pie02] Benjamin C. Pierce. *Types and programming languages*. MIT Press, 2002.

[PKB+16] Guillaume Petiot, Nikolai Kosmatov, Bernard Botella, Alain Giorgetti, and Jacques Julliand. Your proof fails? testing helps to find the reason. In *Proc. of the 10th International Conference on Tests and Proofs (TAP 2016)*, volume 9762, pages 130–150. Springer, 2016.

[Sch86] David A. Schmidt. *Denotational Semantics: A Methodology for Language Development*. William C. Brown Publishers, 1986.

[SD16] Marcelo Sousa and Isil Dillig. Cartesian Hoare Logic for Verifying k-safety Properties. In *Proc. of the 37th Conference on Programming Language Design and Implementation (PLDI 2016)*, pages 57–69. ACM, 2016.

[SS14] Christoph Scheben and Peter H. Schmitt. Efficient self-composition for weakest precondition calculi. In *Proc. of the 19th International Symposium on Formal Methods (FM 2014)*, pages 579–594, 2014.

[SSCB12] Geoff Sutcliffe, Stephan Schulz, Koen Claessen, and Peter Baumgartner. The TPTP typed first-order form with arithmetic. In *Proc. of the 18th International Conference on Logic for Programming, Artificial Intelligence, and Reasoning (LPAR 2012)*, pages 406–419, 2012.

[Tea17] The Coq Development Team. The Coq Proof Assistant, 2017. `https://coq.inria.fr/`.

[U+17] Mattias Ulbrich et al. Deductive verification of relational properties, 2017. JML Workshop.

[VSK17] Kostyantyn Vorobyov, Julien Signoles, and Nikolai Kosmatov. Shadow state encoding for efficient monitoring of block-level properties. In *Proc. of the International Symposium on Memory Management (ISMM 2017)*, pages 47–58, 2017.

[Win93] Glynn Winskel. *The formal semantics of programming languages - an introduction*. Foundation of computing series. MIT Press, 1993.

[Yan07] Hongseok Yang. Relational separation logic. *Theoretical Computer Science*, 375(1-3):308–334, 2007.

[YVS+18] Weikun Yang, Yakir Vizel, Pramod Subramanyan, Aarti Gupta, and Sharad Malik. Lazy self-composition for security verification. In *Proc. of the 30th International Conference Computer Aided Verification (CAV 2018)*, pages 136–156, 2018.

[ZS13] Hui Zhan-Wei and Huang Song. A formal model for metamorphic relation decomposition. In *Proc. of the 4th World Congress on Software Engineering (WCSE 2013)*, pages 64–68, 2013.

Index

université PARIS-SACLAY
ÉCOLE DOCTORALE
INTERFACES
Approches Interdisciplinaires : Fondements, Applications et Innovation

Titre: Propriétés relationnnelles pour la spécification et la vérification de programmes C avec Frama-C

Mots-clefs: Vérification déductive, propriétés relationnelles, Frama-C

Les techniques de vérification déductive fournissent des méthodes puissantes pour la vérification formelle des propriétés exprimées dans la Logique de Hoare. Dans cette formalisation, également connue sous le nom de sémantique axiomatique, un programme est considéré comme un transformateur de prédicat, où chaque programme c exécuté sur un état vérifiant une propriété P conduit à un état vérifiant une autre propriété Q.

Les propriétés relationnelles, de leur côté, lient un ensemble de programmes à deux propriétés. Plus précisément, une propriété relationnelle est une propriété concernant n programmes $c_1, ..., c_n$, indiquant que si chaque programme c_i commence dans un état s_i et termine dans un état s'_i tel que $P(s_1, ..., s_n)$ soit vérifié, alors $Q(s'_1, ..., s'_n)$ est vérifié. Ainsi, les propriétés relationnelles invoquent tout nombre fini d'exécutions de programmes éventuellement dissemblables.

De telles propriétés ne peuvent pas être exprimées directement dans le cadre traditionnel de la vérification déductive modulaire, car la sémantique axiomatique ne peut se référer à deux exécutions distinctes d'un programme c, ou à des programmes différents c_1 et c_2.

Cette thèse apporte deux solutions à la vérification déductive des propriétés relationnelles. Les deux approches permettent de prouver une propriété relationnelle et de l'utiliser comme hypothèse dans des vérifications ultérieures. Nous modélisons ces solutions à l'aide d'un mini-langage impératif contenant des appels de procédures.

Les deux solutions sont implémentées dans le contexte du langage de programmation C, de la plateforme FRAMA-C, du langage de spécification ACSL et du plugin de vérification déductive WP. Le nouvel outil, appelé RPP, permet de spécifier une propriété relationnelle, de la prouver en utilisant la vérification déductive classique, et de l'utiliser comme hypothèse dans la preuve d'autres propriétés. L'outil est évalué sur une série d'exemples illustratifs.

Des expériences ont également été faites sur la vérification à l'exécution de propriétés relationnelles et la génération de contre-exemples lorsqu'une propriété ne peut être prouvée.

Title: Relational properties for specification and verification of C programs in Frama-C

Keywords: Deductive verification, relational properties, Frama-C

Deductive verification techniques provide powerful methods for formal verification of properties expressed in Hoare Logic. In this formalization, also known as axiomatic semantics, a program is seen as a predicate transformer, where each program c executed on a state verifying a property P leads to a state verifying another property Q.

Relational properties, on the other hand, link n program to two properties. More precisely, a relational property is a property about n programs $c_1, ..., c_n$ stating that if each program c_i starts in a state s_i and ends in a state s'_i such that $P(s_1, ..., s_n)$ holds, then $Q(s'_1, ..., s'_n)$ holds. Thus, relational properties invoke any finite number of executions of possibly dissimilar programs.

Such properties cannot be expressed directly in the traditional setting of modular deductive verification, as axiomatic semantics cannot refer to two distinct executions of a program c, or different programs c_1 and c_2.

This thesis brings two solutions to the deductive verification of relational properties. Both of them make it possible to prove a relational property and to use it as a hypothesis in the subsequent verifications. We model our solutions using a small imperative language containing procedure calls.

Both solutions are implemented in the context of the C programming language, the FRAMA-C platform, the ACSL specification language and the deductive verification plugin WP. The new tool, called RPP, allows one to specify a relational property, to prove it using classic deductive verification, and to use it as hypothesis in the proof of other properties. The tool is evaluated over a set of illustrative examples.

Experiments have also been made on runtime checking of relational properties and counterexample generation when a property cannot be proved.

Université Paris-Saclay
Espace Technologique / Immeuble Discovery
Route de l'Orme aux Merisiers RD 128 / 91190 Saint-Aubin, France