Generalization of counterexamples for inductive invariant synthesis

Mickaël Laurent

Generalization of counterexamples for inductive invariant synthesis

Mickaël LAURENT

Carnegie Mellon University
Ecole Normale Supérieure Paris-Saclay

Supervisor: Bryan PARNO

March-July 2018

Contents

Chapter 1

Presentation of IVy

1.1 Context

As distributed programs become more and more widespread, their verification is a major challenge. The general purpose of my internship is to make the certification of those programs easier, by providing new methods and semi-automated tools that will assist the user in this process. Certifying a program consists in writing some specifications (in our case, some safety properties), and then proving than the program satisfy them. Several languages and tools already exist for that:

- Proof-assistants (Coq, F*, etc.): Powerful logics can be used for the specification, however the user has to manually write the proofs (in a interactive way).
- SMT-solver based tools (dafny, F*, etc.): Logics and constructions are often restricted, but specifications can automatically be checked using a SMT-solver. Most of time, this process is undecidable, so there are 3 possible outputs:
  - Yes, the program matches the specifications
  - No, the program doesn't match the specifications (sometimes a counterexample can be provided)
  - I don't know whether the program matches the specifications or not.

  When this last case occurs, the user has to reformulate the specifications differently or to explicitely write some intermediate properties (like inductive invariants) in order to help the SMT-solver.

IVy[1] is a language and a set of tools that allows the user to certify its program using a SMT-solver based approach. However, unlike most of its concurrents, IVy restricts the language and the logic used for specifications in order to be able to check the program in a decidable way.

This approach has many advantages:

- Once the program is written and specified in an accepted logic, it can be checked more easily: the checker can always decide whether the program matches the invariants or not, and give a counterexample if it does not.
- It does not rely on any heuristic/specificity of the SMT-solver used. When a program cannot be checked, a precise reason is provided.

- A program can be checked with any version of any SMT-solver (because it does not depend on any heuristic).

The main disadvantage is that the user is forced to specify its program in a decidable fragment of first order logic: it can force him to rethink the architecture of its code, to fragment its code by creating intermediate abstract modules, to add some 'ghost' variables, etc.

1.2 The RML language

IVy is inspired by a modeling language called RML (Relational Modeling Language)[1]. It is restricted such that some decidability properties are guaranteed (see section 1.4). A RML program is composed of:

- Some uninterpreted types called 'sorts'. 'Uninterpreted' means that these types are not interpreted in any existing theory like arithmetic: they are just set of values without any specific structure, though we can add axioms to our system (see below). A sort has an unbounded number of elements. Some enumerated types can also be defined (= types with fixed finite number of elements).
- Some variables, relations and functions over these types (a variable can be considered like a function of arity 0, and a relation can be considered as a function which returns a boolean). These elements are mutable (their valuation can be modified by the transitions described below).
- Some axioms over these types, variables, functions and relations. Axioms are $\exists^*\forall^*$ formulas (ϕ_{EA} in the grammar below).
- Some transitions that we call 'actions'. An action is characterized by a name and a $statement$ (see the grammar below).
- A special 'init' action that is only executed once at the beginning (before any other action).
- A set of safety properties, that are first order formulas that should be satisfied after the execution of any action in a valid run (starting with the 'init' action). Only a restricted fragment of first order logic can be used, for decidability reasons (see sections 1.3 and 1.4).

The transitions can be written using the following grammar:

$$
\begin{aligned}
\langle \textit{statement} \rangle ::=\ & \text{skip} \\
\mid\ & \text{abort} \\
\mid\ & \mathbf{r}(\bar{x}) := \phi_{QF}(\bar{x}) \\
\mid\ & \mathbf{f}(\bar{x}) := t(\bar{x}) \\
\mid\ & \mathbf{v} := * \\
\mid\ & \text{assume } \phi_{EA} \\
\mid\ & \textit{statement} \ ; \ \textit{statement} \\
\mid\ & \textit{statement} \mid \textit{statement}
\end{aligned}
$$

$$
\begin{aligned}
\langle t \rangle ::=\ & x \\
\mid\ & \mathbf{v} \\
\mid\ & \mathbf{f}(t, \ldots t) \\
\mid\ & \text{ite}(\phi_{QF}, t, t)
\end{aligned}
$$

$$
\begin{aligned}
\langle \phi_{QF} \rangle ::=\ & \mathbf{r}(t, \ldots t) \\
\mid\ & t = t \\
\mid\ & \phi_{QF} \wedge \phi_{QF} \quad \mid \quad \phi_{QF} \vee \phi_{QF} \quad \mid \quad \neg \phi_{QF}
\end{aligned}
$$

$$
\langle \phi_{EA} \rangle ::= \exists x_1, \ldots, x_n \forall x_{n+1}, \ldots, x_{n+m} \phi_{QF}
$$

The semantic of terms and formulas is as usual. The semantic of statements will be described later in the section 1.4.

One important thing to note is that types can only be uninterpreted sorts or enumerated types (like booleans). It means that the structure of these types and the operations over them are only constrained by the axioms of the RML program, and the form of these axioms is quite restrictive. In particular, we can't define arithmetic operations (see section 1.3).

Also, unlike some other languages like F*, we can't have refinement types or dependent types. As a consequence, we can't encode any property in the types. Instead, the specifications of the program are expressed with assertions (that can be built with the primitives 'I', 'assume' and 'abort') and safety properties.

1.3 Decidable logics

In this section, we will describe the Bernays-Schönfinkel class of formulas. It is a decidable fragment of the first order logic that is used in RML. We will refer to it as EPR (for Effectively Propositional) because this logic can be effectively translated into propositional logic (see the subsection 1.3.2).

1.3.1 The Bernays-Schönfinkel class (EPR)

The Bernays-Schönfinkel class is the set of first-order formulas such that:

- Under the prenex normal form, they have an $\exists^*\forall^*$ quantifier prefix (a formula in prenex normal form is composed of a string of quantifiers called the prefix, followed by a quantifier-free part).
- They do not use any function symbol (only a finite number of constants and relations).

For instance, the following formulas define a total relation order and are in the Bernays-Schönfinkel class:

$$\begin{aligned} \forall x.\ R(x,x) &\quad \text{(reflexivity of R)} \\ \forall x,y,z.\ R(x,y) \wedge R(y,z) \rightarrow R(x,z) &\quad \text{(transitivity of R)} \\ \forall x,y.\ R(x,y) \wedge R(y,x) \rightarrow x = y &\quad \text{(antisymetry of R)} \\ \forall x,y.\ R(x,y) \vee R(y,x) &\quad \text{(totality of R)} \end{aligned}$$

1.3.2 Deciding the satisfiability

We want to decide whether an EPR formula is satisfiable or not. Let's consider for instance the following formula:

$$F : \exists x.\forall y.\ R(y,y) \wedge \neg R(x,y)$$

This formula is not satisfiable, because for any value of x, the formula $R(y,y) \wedge \neg R(x,y)$ is false when we take $y = x$. However, there is infinitely many possible interpretations for the relation R (because the set of possible values for x and y is not bounded). So how to decide the satisfiabilty of such formulas?

First, we can get rid of the existential quantifier by introducing a new constant A:

$$\forall y.\ R(y,y) \wedge \neg R(A,y)$$

This is called the skolem normal form of F. In the general case, we can eliminate any existential quantifier of a formula in prenex normal form by introducing a new function that depends on all previously declared universally quantified variables. This process is called 'skolemization' and it preserves the satisfiability of the formula.

Property (Skolemization)**.** *Let F be a formula and F' be the formula obtained after solemization of F. Then F is satisfiable iff F' is satisfiable.*

Now, we want to restrict the domain of the possible interpretations in order to be able to decide the satisfiability of our formula.

Definition (Herbrand domain)**.** *The Herbrand domain of a formula is the set of every term that can be written with constants and functions symbols that are used in the formula (if there is no constant symbol in the formula, a new fresh constant symbol can be used).*

Our formula has a unique constant A and no function symbol (only a relation symbol), so its Herbrand domain is the following: $\{A\}$. More generally, we have the following property:

Property. *The Herbrand domain of an EPR formula is finite.*

Thus, we can decide whether or not our formula is satisfiable by an interpretation over it's Herbrand domain. For that, we just have to test every interpretation of R over the domain $\{A\}$:

- With $R(A, A) =$ false: $\forall y.\ R(y, y) \wedge \neg R(A, y)$ is false (with $y \in \{A\}$)
- With $R(A, A) =$ true: $\forall y.\ R(y, y) \wedge \neg R(A, y)$ is false (with $y \in \{A\}$)

So our formula is not satisfiable for any interpretation over it's Herbrand domain.

Finally, by applying the following theorem, we can conclude that our formula is not satisfiable by any interpretation (over any domain):

Theorem (Herbrand)**.** *A universally quantified first-order formula F is satisfiable iff it is satisfied by an interpretation over its Herbrand domain.*

This process can be used to decide the satisfiability of any EPR formula. Moreover, when a formula is satisfiable, it gives us an interpretation over the Herbrand domain that satisfy the formula. The Herbrand domain being finite, we have the following property:

Property (Finite model property)**.** *Every satisfiable EPR formula has a finite model.*

1.3.3 EPR with types

In RML, relations, functions and constants (that are preferably called 'variables' since they are mutable) have types. In particular, the arguments and the output of a function have types. We can easily add these typing constraints to the logical symbols we used in the previous subsection. Moreover, it allows us to extend the domain of EPR formulas by still keeping a finite Herbrand domain.

Indeed, we can allow function symbols and $\forall\exists$ quantifier alternation as long as, once the formula is skolemized, we have the following property (we say that function symbols are **stratified**): there exists a

strict order '<' over the unbounded sorts such that, for every function symbol $f : t_1, \ldots, t_n \to t$, we have $t_i < t$ for all $i \in [1..n]$ such that t_i and t are unbounded sorts.

However, the EPR fragment can still be too restrictive: for instance, we can't introduce linear arithmetic using only EPR axioms. Indeed, the standard way to define integers is to introduce a successor function symbol, which is not allowed because it can't be stratified. More generally, we can't characterize any infinite set with only EPR axioms: it would contradict the finite model property.

Actually, some strict extensions of EPR are still decidable, for instance the Finite Almost Uninterpreted fragment[2] which allows some basic linear arithmetic. However, we will not use such extensions in this report because it breaks the finite model property.

1.4 Checking a RML program

Definition (Structure & State)**.** *For a given RML program, a structure (or interpretation) is a pair* $((D_s)_{s \in sorts}, (V_s)_{s \in symbols})$ *where:*

- *For all* $s \in$ *sorts,* D_s *is a domain associated to the sort* s*.*
- *For all* $s \in$ *symbols (variable, function or relation),* V_s *is a valuation for the symbol* s *(using the domains* $(D_s)_{s \in sorts}$*).*

A state is a structure that satisfies every axiom of the RML program.

In order to certify that a RML program never aborts abnormally and satisfies its safety properties, we must provide IVy with an invariant I such that:

1. I is initially satisfied (after any execution of the 'init' action starting from any state).
2. Any state satisfying I also satisfy all the safety properties.
3. I is **inductive**, that is: from any state satisfying I, any execution of one step (= one action) will not fail and will lead to a new state that also satisfy I.

To formalize these properties, we introduce the notion of weakest precondition wp:

Definition (Weakest precondition)**.** *For any statement* S *and formula* Q*, the weakest precondition of* Q *by* S *(noted* $wp(S, Q)$*) is the weakest formula* P *such that every execution of* S *starting from a state satisfying* P *will lead to a state that satisfies* Q*.*

Weakest preconditions can be computed as follows[1]:

Statement	Semantic	wp(statement, Q)
skip	Do nothing	Q
abort	Terminate abnormally (fail)	false
$\mathbf{r}(\bar{x}) := \phi_{QF}(\bar{x})$	Quantifier-free update of relation **r**	$(A_r \to Q)[\phi_{QF}(\bar{s}) \,/\, \mathbf{r}(\bar{s})]$
$\mathbf{f}(\bar{x}) := t(\bar{x})$	Update of function **f** to term $t(\bar{x})$	$(A_f \to Q)[t(\bar{s}) \,/\, \mathbf{f}(\bar{s})]$
$\mathbf{v} := *$	Non-deterministic assignment of variable **v**	$\forall x.(A_v \to Q)[x \,/\, \mathbf{v}]$
assume ϕ_{EA}	Assume a $\exists^*\forall^*$ formula holds	$\phi_{EA} \to Q$
S_1 ; S_2	Sequential composition	$wp(S_1, wp(S_2, Q))$
S_1 | S_2	Non-deterministic choice	$wp(S_1, Q) \wedge wp(S_2, Q)$

$\phi(\beta, \alpha)$ denotes ϕ with occurences of α substituted by β, and A_s denotes the axioms of the RML program that involve the symbol s.

Now, the properties above can be rewritten using weakest preconditions:

1. $A \Rightarrow wp(\text{init}, I)$
2. For any safety property P, $I \Rightarrow P$
3. $A \wedge I \Rightarrow wp(\text{action}_1 | \ldots | \text{action}_n, I)$

Lemma. *Let S be a RML statement and Q a $\forall^*\exists^*$ formula (= a formula whose normal prenex form has a $\forall^*\exists^*$ prefix followed by a quantifier-free part), then $wp(S, Q)$ is also a $\forall^*\exists^*$ formula.*

This lemma can easily be proved by induction on S (we remind that axioms A are $\exists^*\forall^*$ formulas).

Moreover, according to the section 1.3, satisfiability of $\exists^*\forall^*$ formulas can be checked if function symbols are stratified. We can deduce the following theorem:

Theorem. *Let S be a RML statement, P a $\exists^*\forall^*$ formula and Q a $\forall^*\exists^*$ formula. Then, if all function symbols are stratified, checking the validity of the three formulas above is decidable (it can be done by checking the satisfiability of their negation).*

In particular, checking whether a formula with no quantifier alternation satisfies these three properties is decidable. In practice, since satisfiability of some formulas with $\forall\exists$ quantifier alternation can still be decidable, some invariants with quantifier alternation are allowed in RML. However, if an invariant can't be checked in a decidable way, IVy will reject it and will indicate to the user the faulty functions or quantifier alternations.

1.5 Challenges and current research

As we have just seen, some inductive invariants can be checked in a decidable way. However, it does not mean that they can be automatically produced.

Indeed, when writing a protocol using IVy, the two main challenges are:

- The implementation and specification of programs, in accordance with the restrictions of RML (stratification of functions, restriction on formulas, no recursion... though some additional primitives can be added to RML). A lot of research is in progress in order to help the user to make such programs. In particular, two majors methods are studied: the fragmentation of the code into many independent modules that can use different fragments of the first-order logic[2], and the insertion of some mutable 'ghost' relations in the program that aim to capture the values of some formulas[3].
- The finding of an inductive invariant (once the program is implemented and specified). As we will see in the next part, IVy can help the user to find an inductive invariant by providing a counterexample if the current invariant is not inductive. From this counterexample, a new conjecture can be automatically generated in order to strengthen the current invariant. My work has consisted in improving this synthesis.

Some other challenges are also studied, like the verification of liveness properties[4].

Chapter 2

Invariant synthesis

2.1 Introduction

The synthesis of an inductive invariant is a common problem in verification, and even with the restrictions of RML, it is still undecidable (though it can be decidable for some very restricted classes of invariants)[5]. Nevertheless, even an partial resolution of this problem can still be very helpful for the user.

In the context of RML programs, most of invariants can be expressed with universal formulas ($\forall^*$), but sometimes a $\forall^*\exists^*$ formula can be required (under some restrictions in order to stay in a decidable fragment of logic). Thus, we will restrain our search to these two types of invariant. In particular, when possible, we will try to synthetize universal invariants because they are easier to generate and will never cause any decidability issue.

As described in this article[6], a common way to find an inductive invariant (see section 1.4) is to start from the safety properties of the program and try to strengthen them by adding new invariants (using one of the methods described in the next sections) until the resulting set of invariants form an inductive invariant (thereafter, we will use the terms 'invariant' or 'set of invariants' indistinguishably: a set of invariants can be seen as a single invariant that is the conjunction of all the invariants of the set). This iterative process may not terminate, but it offers to the user an interactive way to find an inductive invariant: if the current invariant is not inductive, a new conjecture is proposed to the user (in order to strengthen the current invariant). If this conjecture is simple enough, the user can understand it and adjust it as he wants (making it stronger, resolving decidabilities issues, etc.), and we repeat this process.

Thus, the problem we want to solve is the following: given an invariant that is not inductive, we want to strenghten it by generating a new (correct) invariant that is as strong as possible, if possible without breaking the decidability of the system. To be correct, an invariant must be satisfied by every state that can be obtained with a valid execution (beginning with the 'init' action). The following criteria can be used to evaluate an algorithm that tries to solve this problem:

1. **Reliability**: Can the algorithm fail, not terminate or generate incorrect invariants?

2. **Decidability**: Is the new set of invariants generated checkable in a decidable way (in our case, we want it to be EPR)?

3. **Efficiency**: How strong is the new invariant generated? Is it easily understandable by the user (in order to be adjusted if needed)?

2.2 Weakest precondition as a new invariant

One possible way to strengthen an invariant I is to make the conjunction with its weakest precondition $wp(\text{action}_1|\ldots|\text{action}_n, I)$. If the initial invariant is not inductive, the result will be strictly stronger.

Let's evaluate this method with the criteria above.

Reliability: This method always terminates and never fails. Moreover, it produces correct invariants. Here is a proof:

We assume that the initial invariant I is correct. Let $P = wp(\text{action}_1|\ldots|\text{action}_n, I)$.

By contradiction, let's suppose that we have a state S, issued from a valid execution E, that doesn't satisfy P. Two cases are possible:

- There exists an execution E' of one step (= one action) starting in S such that the resulting state S' doesn't satisfy the initial invariant. E followed by E' is a valid execution that results in the state S'. As S' doesn't satisfy I, we have a contradiction.

- Every possible execution of one step starting in S will result in a state that satisfy I. In this case, we can generate a precondition for our initial invariant that is stricly weaker than P: let F be a formula that fully characterize S modulo isomorphism (it can be done by constraining the cardinality of the domains and the valuations of the different symbols in these domains), then $P \vee F$ is still a precondition for I and is strictly weaker than P (contradiction).

So P is a correct invariant.

Decidability: Even if the initial invariant is EPR, the weakest precondition is not guaranteed to be EPR. Indeed, when computing the weakest precondition, stratification can be broken on several cases: assumptions, non-deterministic operations, etc.

Nevertheless, we can have the guarantee that the weakest precondition is EPR by adding these restrictions:

- All function symbols used must be stratified.

- The initial invariant must be universally quantified.

- Assumptions must be existentially quantified.

- Axioms that involve a mutable variable, function or relation must be existentially quantified.

We can easily verify that under these conditions, the weakest precondition will be an universally quantified formula and thus will be EPR. However, these conditions are very restrictive and so they can't be reasonably imposed.

Efficiency: Weakest preconditions aim to be 'weakest', while we want an invariant as strong and simple as possible. They are usually very long, even after simplification (they can grow exponentially). Moreover, they are not focused on one aspect in particular (they mix many different elements together), and so they are even more complicated to understand.

2.3 Naive counterexample generalization

In order to produce simpler invariants, we can focus our search on a concrete counterexample.

Definition (Counterexample)**.** *A counterexample is a pair (S, E) where S is a state that satisfies the current invariant and E is a deterministic statement that describes the execution of an action, starting from S and leading to a new state S' that doesn't satisfy the current invariant.*

Thanks to the finite model property of EPR (section 1.3.2), we are always able to find a finite counterexample (i.e. a counterexample with finite domains) when the current invariant is not inductive (section 1.4 explains how to check the inductiveness of an invariant).

Let (S, E) be a finite counterexample.

Definition. *Let S be a state, then $constr(S)$ is the set of all constraints of the following forms that are satisfied by S:*

- $symbol(concrete\_value_1, \ldots, concrete\_value_n) = concrete\_value$ *with 'symbol' refering to any variable, function or relation of S and 'concrete values' being elements of the domains of S*
- $concrete\_value_1 \neq concrete\_value_2$ *(where '$concrete\_value_1$' and '$concrete\_value_2$' have the same type)*

We can generate a formula as follows:

1. We associate a fresh variable X_v to each concrete value v in S.
2. In each constraint of $constr(S)$, we replace each concrete value v by X_v.
3. We make the conjunction of all these constraints and we quantify existentially on each X_v. Optionaly, we can simplify the resulting formula to get rid of some variables.

The resulting formula F is an existentially quantified formula that characterizes all the structures containing a substructure that is isomorphic to the structure of the counterexample. In order to prevent this counterexample from happening again, we just have to negate F (it will become universally quantified) and add it to the current set of invariants.

This technique always generate an universal invariant, and so there is no decidability issue (if the function symbols are stratified). However, there are three major issues:

- As we will see in the next section, the new invariant generated is not always correct.
- The formula generated can be very long, because it contains all the constraints characterizing our counterexample.
- The formula generated is very specific: it will not prevent other *similar* counterexamples from happening, as long as at least one constraint differs, even if this difference has no impact.

2.4 A problematic example with non-monotonicity

Let's try the naive counterexample generalization on this IVy code:

```
type a
type b

relation f(X:a,Y:b)
relation dom(X:a)
individual elt : a
individual img : b

after init
{
    dom(X) := false;
    f(X,Y) := false;
}
```

```
action unset(x:a) =
{
    dom(x) := false;
    f(x,B) := false;
}

action set(x:a,y:b) =
{
    dom(x) := true;
    f(x,Y) := Y=y;
    if elt=x {
        img := y;
    };
}
```

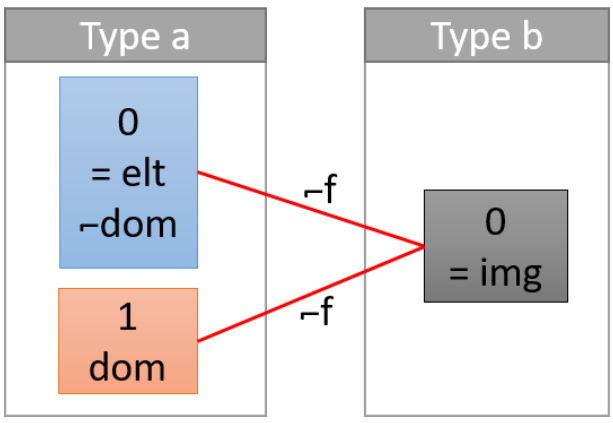
```
action change_elt (e:a) =
{
    elt := e;
    if some r:b. f(elt,r)
    {
        img := r;
    };
}

conjecture dom(elt) -> f(elt,img)
```

The keyword *individual* is used to declare a variable, and the construction *if some* x. $\phi(x)$ *{ statement }* assigns to the variable x a value v such that $\phi(v)$ is true and executes *statement*. If no such value v exists, *statement* is not executed.

In a nutshell, this example defines the following:

- Two uninterpreted sorts a and b.
- A relation f from a to b that will represent a partial function, and dom its domain. It might look curious to not directly define a function instead, but it can sometimes be useful for decidability reasons, as functions are subject to stratification constraints.
- A variable elt in a and a variable img in b. If elt is in the domain of f, then img should always correspond to the image of elt by f (this specification is defined at the end with the keyword *conjecture*).
- The init action, and some other actions to set/unset a value of f and to change the value of elt.

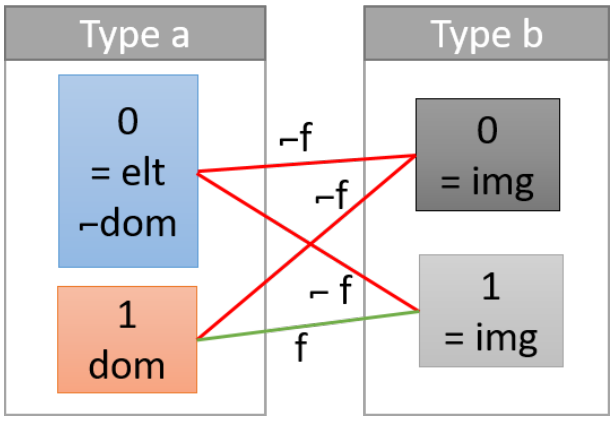
This code always satisfies the conjecture, however the conjecture is not an inductive invariant. Here is a counterexample:



Domain for a: $\{a_0, a_1\}$
Domain for b: $\{b_0\}$
Constraints on valuations:
$elt = a_0$
$\neg dom(a_0)$
$dom(a_1)$
$img = b_0$
$\neg f(a_0, b_0)$
$\neg f(a_1, b_0)$
Execution: change_elt(a_1)

This state satisfies the conjecture because we have $\neg dom(elt)$, but after executing change_elt(a_1) the conjecture will be broken. Applying the method described in the previous section will give the following invariant: $\forall A : a.\neg(\neg dom(elt) \wedge dom(A) \wedge A \neq elt \wedge \neg f(elt, res) \wedge \neg f(A, res))$

However, this invariant is not correct. For instance, the following state doesn't satisfy it and can be obtained through a valid execution:



One element b_1 has been added to b
with the following new valuations:
$\neg f(a_0, b_1)$
$f(a_1, b_1)$
This time, the conjecture remains satisfied after executing any action.
In particular, executing change_elt(a_1) will not break the conjecture anymore.

Indeed, the course of the execution of change_elt(a_1) can be altered just by adding a new element to b (without changing anything else). In this example, it is due to the use of the *if some* statement. More generally, it can be caused by non-determinism, by the use of quantified formulas, or anything that explores all the elements of a sort.

Definition (Monotonicity)**.** *A statement s is monotonic iff, for any two states S_1 and S_2 such that $S_1 \sqsubseteq S_2$ (S_1 is a substructure of S_2), any states S_1' and S_2' that can respectively be obtained after executing s on S_1 and S_2 are such that $S_1' \sqsubseteq S_2'$.*

An invariant P is monotonic iff, for any two states S_1 and S_2 such that $S_1 \sqsubseteq S_2$, if S_1 ***doesn't*** *satisfy P then S_2 doesn't satisfy P.*

Property. *For a counterexample (S, E), if E is monotonic* ***and*** *the broken invariant is monotonic (and correct), then there exists a correct universal invariant that is not satisfied by S (we say that the invariant 'captures' the counterexample).*

Property. *If there exists a correct universal invariant that is not satisfied by S, then the naive counterexample generalization method will always generate such an invariant. More precisely, it will generate the weakest universal invariant that is not satisfied by S.*

If there is no such invariant, the invariant generated will be incorrect.

A proof for this last property can be found in this[6] article (lemma 1).

2.5 Improving the result with symbolic bounded verification

Symbolic bounded verification is a method to check whether a formula is a k-invariant or not:

Definition (k-invariant)**.** *A formula is a k-invariant if it is satisfied after any valid execution of k steps or less (that is, any execution of the init action followed by at most k other actions).*

In order to check whether a formula Q is a k-invariant or not, we can check the validity of the following formula:

$$A \Rightarrow \bigwedge_{j=0}^{k} wp(\text{init}; (\text{action}_1 | \dots | \text{action}_n)^j, Q)$$

where A refers to the axioms and 'statement^j' is the sequential composition of j copies of 'statement'.

In order to check the validity of this formula (for a given k), we can just check the satisfiability of its negation. According to the section 1.4, it is decidable (for A of the form $\exists^*\forall^*$ and Q of the form $\forall^*\exists^*$). Note that symbolic bounded verification does not allow us to find k-invariants, it can only be used to check whether a formula is a k-invariant or not.

The naive counterexample generalization method can be improved by using symbolic bounded verification[1]:

- It could allow us to know when a generated invariant is not correct. Actually, we cannot be sure that an invariant is correct using this method (because it does not analyze executions of length $> k$), but in practice taking $k = 5$ is sufficient to detect incorrect invariants almost everytime.
- It allows us to generate simpler and more general invariants. Indeed, instead of generating a formula that aggregates all constraints of $constr(S)$, we are able to only keep a minimal number of constraints such that the resulting invariant is still correct. It can be encoded as a minimal unSAT core problem and solved using a SMT-solver.
- If the program that we want to check is not correct (it doesn't match the specifications), symbolic bounded verification can sometimes allow us to find it out, while a naive counterexample generalization would just generate incorrect invariants.

Reliability: This method can generate an incorrect invariant if the boundary k for the symbolic bounded verification is too small (sometimes we can't go beyond $k = 3$ in a reasonable time). This method can only generate universal invariants, so it will fail if there is no universal invariant that can capture the counterexample (case of non-monotonicity, see section 2.4).

Decidability: All invariants generated are universal, so it never causes any decidability issue.

Efficiency: Invariants generated are quite small and simple. They are more focused than weakest preconditions (because they are issued from one precise counterexample) so they can be understood more easily. They do not contain any useless litteral, so we can expect them to be quite strong.

This method seems less reliable than just computing the weakest precondition, however it generates invariants that are much more interesting. This is the method that is currently implemented by IVy in order to help the user to synthesize inductive invariants interactively.

Chapter 3

Contributions

3.1 Course of my internship

I spent the first half of my 5-months internship studying the differents aspects of IVy. I wrote some basic protocols in order to get familiar with it, I read many articles in order to be aware of the state-of-the-art, etc.

Finally, I decided to work on the problem of invariant synthesis, and more specifically the counterexample generalization:

- I have proposed a new method to select the important constraints of the counterexample, based on an analysis of the execution of the counterexample. This approach is quite complementary with the preexisting IVy approach (symbolic bounded verification): both approaches can be combined.
- I have extended this method with an additional process that can generate correct invariants of the form $\forall^*\exists^*$ when no correct universal invariant exists.
- I have implemented a proof of concept for these methods, using the languages F# and OCaml and the SMT-Solver Z3[7].
- I have analyzed and compared the guarantees of all these different approaches.

3.2 Filtering constraints by analyzing the execution

The problem we want to solve is as follows: given a counterexample (S, E) (S being the initial state and E a deterministic statement describing the execution of an action), we want to highlight a subset of $constr(S)$ (see section 2.3) as small as possible, such that there is no valid state (= state that can be reached after a valid run) that can satisfy these constraints (for non-monotonic counterexamples it is not always possible, but this case will be treated in section 3.3).

To solve this problem, the IVy approach is to use symbolic bounded verification (see section 2.5) to minimize the number of constraints (it can be encoded as a minimal unSAT core problem). My approach is very different:

1. We compute S' the state obtained after the execution of E from S.

2. We try to compute a subset C' of $constr(S')$ (as small as possible) such that all states satisfying C' (modulo renaming of the concrete values) evaluate the current invariant to the same value (here, to false). More precisely, we compute a set of 'marks', a 'mark' being an element of the form $symbol(concrete\_value_1, \ldots, concrete\_value_n)$ or $concrete\_value_1 \neq concrete\_value_2$. At a given state, a mark is directly associated to a constraint (it *points* to a constraint).

3. We backpropagate these marks through the execution of E in order to obtain a subset C of $constr(S)$ such that all states satisfying C (modulo renaming of the concrete values) can reproduce the execution E and satisfy C' at the end (and so, break the invariant).

To perform the step 2, we can use the following algorithm (formulas and terms are assimilated):

Function `Compute_marks(S,V)`
- **Data:** a state S, a formula or a term V
- **Result:** a set of marks M pointing to some important constraints in $constr(S)$
- **match** V
  - **case** *symbol*$(t_1, \ldots, t_n)$
    - Compute $M_1 =$ `Compute_marks(`S, t_1`)`$, \ldots, M_n =$ `Compute_marks(`S, t_n`)`;
    - Compute $v_1, \ldots, v_n$ the evaluations of $t_1, \ldots, t_n$ in S;
    - **return** $\{symbol(v_1, \ldots, v_n)\} \cup \bigcup M_k$;
  - **case** $t_1 = t_2$
    - Compute v_1, v_2 the evaluations of t_1, t_2 in S;
    - **if** $v_1 = v_2$ **then**
      - **return** `Compute_marks(`S, t_1`)` $\cup$ `Compute_marks(`S, t_2`)`;
    - **else**
      - **return** `Compute_marks(`S, t_1`)` $\cup$ `Compute_marks(`S, t_2`)` $\cup \{v_1 \neq v_2\}$;
  - **case** $\neg F$
    - **return** `Compute_marks(`S, F`)`;
  - **case** $F_1 \vee F_2$
    - Compute b_1, b_2 the evaluations of F_1, F_2 in S;
    - **if** b_1 **then**
      - **return** `Compute_marks(`S, F_1`)`;
    - **else if** b_2 **then**
      - **return** `Compute_marks(`S, F_2`)`;
    - **else**
      - **return** `Compute_marks(`S, F_1`)` $\cup$ `Compute_marks(`S, F_2`)`;
  - **case** $\forall x : a.F(x)$
    - `/* for a given execution, type` a `has a fixed domain */`
    - **if** *there exists* $x_f \in a$ *such that* $F(x_f)$ *evaluates to false* **then**
      - **return** `Compute_marks(`$S, F(x_f)$`)`;
    - **else**
      - `/* potential non-monotonicity, see section 3.3 */`
      - **return** $\bigcup_{x_t \in a}$ `Compute_marks(`$S, F(x_t)$`)`
- **end**

end

For instance, at the end of the counterexample of the section 2.4, the formula $dom(elt) \rightarrow f(elt, img)$ is not satisfied and the marked constraints would be:
$\{elt = a_1 \ ; \ dom(a_1) = true \ ; \ img = b_0 \ ; \ f(a_1, b_0) = false\}$

To perform the step 3, we can use the following algorithm:

Function `Backpropagate` (***M',E,S***)

> **Data:** a set of marks M' computed after the execution of the deterministic statement E from the state S
>
> **Result:** a new set of marks M pointing to some important constraints in $constr(S)$
>
> **match** E
>
> > **case** *skip*
> >
> > > **return** M';
> >
> > **case** ***f***$(\bar{x})$:= $t(\bar{x})$
> >
> > > $M \leftarrow M'$;
> > >
> > > **for** $\bar{x_0}$ *such that* ***f***$(\bar{x_0}) \in M'$ **do**
> > >
> > > > $M \leftarrow (M \backslash \{\mathbf{f}(\bar{x_0})\}) \cup$ `Compute_marks`$(S, t(\bar{x_0}))$;
> > >
> > > **end**
> > >
> > > **return** M;
> >
> > **case** *assume* ϕ_{EA}
> >
> > > **return** $M' \cup$ `Compute_marks`(S, ϕ_{EA});
> >
> > **case** E_1 *;* E_2
> >
> > > Compute S' the state obtained after executing E_1 from S;
> > >
> > > **return** `Backpropagate`(`Backpropagate`$(M', E_2, S'), E_1, S)$;
>
> **end**

end

For instance, after having applied this algorithm to the example of the section 2.4 (all primitives used in this example can be translated to RML), we can detect that the execution is not monotonic, and we obtain the following marked constraints:
$\{img = b_0; dom(a_1); \neg f(a_1, b_0)\} \quad \longrightarrow \quad \forall A : a.\ \neg(dom(A) \wedge \neg f(A, img))$
Note that this invariant is much simpler than the one produced by keeping all the constraints (naive generalization, see section 2.3). Though it is still incorrect in this case (because of non-monotonicity), it will be used to generate another valid invariant (see next sections). For monotonic counterexamples, we have the guarantee that the invariant generated is correct.

This approach has some advantages compared to the preexisting method used by IVy:

- It doesn't need to use symbolic bounded verification, which can return an incorrect output if k is too small (and it can take too much time for bigger k).
- It can detect potential non-monotonicity in the code.
- For a non-monotonic counterexample, it can still gather some important constraints, even if there is no correct universal invariant that can capture the counterexample. From these constraints, a correct invariant of the form $\forall^*\exists^*$ can sometimes be generated, as we will see in the section 3.3.

The major drawback is that, sometimes, some useless constraints are not eliminated (because the computation of the marks is not optimal). As a consequence, resulting invariants can be more complicated and weaker than invariants generated using symbolic bounded verification. However, the two approaches can be used together: symbolic bounded verification can be used after having already filtered the constraints with this method. In this way, the advantages of the two methods are combinated. Moreover, the unSAT core to minimize will be smaller, so it will be done faster.

3.3 Case of non-monotonicity

As we have seen in section 2.4, some non-monotonic counterexamples can't be captured by any correct universal invariant. For this reason, if some potential non-monotonicity is detected during the computation of marks, a second process is performed in order to know whether the generated conjecture is correct or not, and to fix it if it is not correct.

In order to do that, we can try to find some valid states that satisfy all the constraints C. Such states would be the witnesses of the non-correctness of the generated universal invariant. Moreover, if we find such a state, we can use it to weaken the generated conjecture. Otherwise, we know that the universal invariant generated is already correct. In this section, we will see how to find such states, and in the next section we will see how to use them to weaken the conjecture.

Let (S, E) be the initial counterexample, let C be the constraints of S that have been marked. We want to find a state S_v such that:

1. S_v satisfies all the constraints C. It implies that the domains of S_v contain all concrete values that appear in C. However, the domains of S_v can also contain some other elements.

2. S_v is a valid state (it can be reached after a valid run).

The first condition is easy to express with a formula, but the second condition is more complicated (it would probably need some model checking). However, we can approximate it with this other condition (which is weaker, if we assume that the current invariant is correct): S_v satisfies the current invariant and the axioms, and doesn't break any of them after one step (in particular after executing the action that is executed in the counterexample). We can express this new condition with the following formula:

$A \wedge I \wedge wp(\text{action}_1 | \ldots | \text{action}_n, I)$

Note that checking the satisfiability of this formula is not always decidable (so it may not terminate), because the weakest precondition is not always EPR (see section 2.2). However, it was not an issue for the examples that I have tested.

If the conditions 1 and 2 are satisfiable, we can compute a finite model: it gives us a state S_v as desired. We will see in the next section how to produce a formula F_v that 'generalizes' this state and use it to weaken the generated invariant. By adding as a third condition the negation of F_v, an other valid state can be searched. This process can potentially not terminate if a new valid state is always found. Indeed, it could be compared to the whole process of finding an inductive invariant:

- To find an inductive invariant, we start from the safety properties and we strengthen them until there are no more counterexamples.

- To generate a valid invariant from a non-monotonic counterexample, we first generate an universal invariant and we weaken it until there are no more valid states satisfying it.

However, this process of weakening is much more likely to terminate than the whole process of finding an inductive invariant, because the search space is more restricted: we are only looking for valid states that satisfy the constraints C.

3.4 Weakening the conjecture

Once we have a valid state S_v that satisfies all the marked constraints C of the counterexample, we will generalize it and we will build a new invariant that will allow valid states that are similar to S_v.

The process of generalization of S_v is quite similar to the process of generalization of the counterexample (S, E). First, we can compute a subset C_v of $constr(S_v)$ by computing marks for some relevant executions starting in S_v (it will not be detailled here, but the idea is to find an execution that is as close to E as possible). Note that symbolic bounded verification can't be used here: we don't want our formula to be a k-invariant (= to be satisfied by all valid k-executions), but only to be satisfied by at least one valid execution. Then, we remove from C_v every constraint that is already in C in order to keep only constraints that are specific to S_v.

Next, we replace each different concrete value in C_v by a fresh existentially quantified variable, but only for concrete values that are specific to S_v: concrete values that also appear in C are kept as they are. This will give us a semi-generalized formula F_v that characterize, among the states satisfying C, those that also contains a substructure isomorphic to S_v. Note that it may exist some states that satisfy F_v but that are not valid (again, due to a potential non-monotonicity of the negation of the current invariant). However, it is not a major issue since it will not compromise the correctness of the invariant that will be generated.

Now, we can build a new invariant that will not exclude states similar to S_v. We still want to focus on states that satisfy C, but since we only want to characterize invalid states, we don't want to include states that satisfy F_v. Thus, we use the following formula: $C \wedge (\neg F_v)$. This formula still contains concrete values (those that appear in C), so we must replace them by existentially quantified variables. The resulting formula has the form $\exists^* \ldots \wedge \neg(\exists^* \ldots)$. After negating it, we obtain a new invariant of the form $\forall^* \ldots \rightarrow (\exists^* \ldots)$, that is an invariant with a $\forall^*\exists^*$ prefix.

Here is a table that illustrates this process on the example of the section 2.4:

	Counterexample (S, E)	Valid state S_v
Shared concrete constraints (after reduction and simplification)	$dom(a_1)$ $\neg f(a_1, img)$	
Shared concrete values	$\{a_1\}$	
Constraints specific to S_v (after reduction)	-	$f(a_1, b_1)$
Concrete values specific to S_v	-	$\{b_1\}$
Semi-generalized formula F_v	-	$\exists B : b.\ f(a_1, B)$
New generated invariant	$\forall A : a.\ dom(A) \wedge \neg f(A, img) \rightarrow \exists B : b.\ f(A, B)$	

3.5 Comparison with other methods

Let's evaluate the method presented in this chapter using the same criteria as before.

Reliability: If we assume that the initial invariant is correct, then this algorithm always generates correct invariants. However, if there is no correct universal invariant that can capture the counterexample, this algorithm might not terminate (see section 3.3).

Decidability: If there exists a correct universal invariant that can capture the counterexample, then such an invariant will be produced. Otherwise, an invariant of the form $\forall^*\exists^*$ will be produced. It may cause some decidability issues that the user will have to solve (it is a challenge that the user will often have to face when using IVy, see section 1.5).

Efficiency: Invariants generated are quite small and simple because they only contains elements that have played an important role in the execution of the counterexample. Symbolic bounded verification can be used to improve the result.

	Weakest precondition (section 2.2)	Counterexample generalization	
		Symbolic bounded verification (section 2.5)	Execution analysis + Weakening (chapter 3)
Reliability	Always terminate, always correct	Can only generate universal invariants (no guarantee for non-monotonic counterexamples), incorrect if k is too small	Can generate universal invariants when possible, otherwise can generate $\forall^*\exists^*$ invariants (but may not terminate in this case)
Decidability	Not always EPR. Additional restrictions can be imposed to always have EPR weakest preconditions	Generated invariants are always universal (so they always can be checked in a decidable way)	Generated invariants are universal when possible (in particular, for monotonic counterexamples)
Efficiency	Generated invariants are very long, unreadable, and often too weak	Generated invariants are simple and don't contain any useless litteral	Generated invariants are quite simple, symbolic bounded verification can be used to improve the result

3.6 Conclusion

This internship has been a very fulfilling experience for me. I have experimented an aspect of research that I had not experienced during my last year's internship: during the first half of the internship, I had to find a precise problem to focus on and some ideas to solve it. My comprehension of the subject has really evolved during this time. I read many articles and explored many different problematics before focusing on the invariant synthesis.

For instance, one of my attempts was to try to add some more advanced typing to IVy, without loosing decidability. In particular, I have studied *session types* (it was even more interesting since there is a lot of session types experts in Carnegie Mellon University). Finally, I gave up this idea since the linearity of session types seemed hardly reconciliable with the distributed and decentralized aspects of the protocols that we wanted to prove. My ideas concerning the invariant synthesis came just after having implemented a queue and a lock protocol in IVy and having tried to prove them using the preexisting tools.

Bibliography

[1] Oded Padon, Kenneth L. McMillan, Aurojit Panda, Mooly Sagiv, and Sharon Shoham. Ivy: Safety verification by interactive generalization. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '16, pages 614–630, New York, NY, USA, 2016. ACM.

[2] Marcelo Taube, Giuliano Losa, Kenneth L. McMillan, Oded Padon, Mooly Sagiv, Sharon Shoham, James R. Wilcox, and Doug Woos. Modularity for decidability of deductive verification with applications to distributed systems. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2018, pages 662–677, New York, NY, USA, 2018. ACM.

[3] Oded Padon, Giuliano Losa, Mooly Sagiv, and Sharon Shoham. Paxos made epr: Decidable reasoning about distributed protocols. *Proc. ACM Program. Lang.*, 1(OOPSLA):108:1–108:31, October 2017.

[4] Oded Padon, Jochen Hoenicke, Giuliano Losa, Andreas Podelski, Mooly Sagiv, and Sharon Shoham. Reducing liveness to safety in first-order logic. *Proc. ACM Program. Lang.*, 2(POPL):26:1–26:33, December 2017.

[5] Oded Padon, Neil Immerman, Sharon Shoham, Aleksandr Karbyshev, and Mooly Sagiv. Decidability of inferring inductive invariants. *SIGPLAN Not.*, 51(1):217–231, January 2016.

[6] Aleksandr Karbyshev, Nikolaj Bjørner, Shachar Itzhaky, Noam Rinetzky, and Sharon Shoham. Property-directed inference of universal invariants or proving their absence. *J. ACM*, 64(1):7:1–7:33, March 2017.

[7] Leonardo de Moura and Nikolaj Bjørner. Z3: An efficient smt solver. In C. R. Ramakrishnan and Jakob Rehof, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, pages 337–340, Berlin, Heidelberg, 2008. Springer Berlin Heidelberg.

Appendix A

Experimental results

A.1 A queue

Here is an implementation and a specification of a queue. The initial safety properties are the two first conjectures (free capital variables are quantified universally by default).

```
#lang ivy1.7

# Implementation and specification of a queue

module total_order(t,r) = {
    axiom r(X:t,X)                          # Reflexivity
    axiom r(X:t, Y) & r(Y, Z) -> r(X, Z)    # Transitivity
    axiom r(X:t, Y) & r(Y, X) -> X = Y      # Anti-symmetry
    axiom r(X:t, Y) | r(Y, X)               # Totality
}

object incrementable = {

    type t
    relation (X:t <= Y:t)

    relation succ(X:t,Y:t)

    action next(input:t) returns (output:t) # No implementation given: it is an abstract module

    # Specifications of this abstract module
    object spec = {

        property succ(X,Z) -> (X < Z & ~(X < Y & Y < Z))

        instantiate total_order(t, <=)

        after next {
            ensure input < output ;
            ensure succ(input, output);
        }
    }

    # Provide an implementation for this abstract module (implementation is not EPR).
    object impl = {

        interpret t -> int

        definition succ(X,Y) = (Y = X + 1)

        implement next {
            output := input + 1;
        }
    }
}
```

```
module unbounded_queue(data) =
{
    individual next_e : incrementable.t
    relation content(D:data, E:incrementable.t)
    individual first_e : incrementable.t
    individual first : data

    after init
    {
        content(D,E) := false ;
        first_e := 0 ;
        next_e := 0;
    }

    derived empty = first_e >= next_e

    action push (da:data) =
    {
        if empty
        {
            first := da;
        } ;
        content(da, next_e) := true ;
        next_e := incrementable.next(next_e);
    }

    action pop returns (res:data) =
    {
        require ~empty ;
        res := first ;
        content(first, first_e) := false ;
        first_e := incrementable.next(first_e) ;
        if some nf:data. content(nf,first_e)
        {
            first := nf;
        };
    }

    object spec =
    {
        conjecture ~empty -> content(first,first_e)
        conjecture content(A,first_e) -> A=first

        # Invariants found
        conjecture (A < q.next_e) & (q.first_e < A) -> (exists B:data. q.content(B, A))
        conjecture ~q.content(B, q.next_e)
        conjecture ~(B >= q.next_e & q.content(A, B))
        conjecture ~(q.content(A, D) & q.content(B, D) & A ~= B)
    }

    export pop
    export push
}

type data
instance q: unbounded_queue(data)

isolate iso_incr = incrementable
isolate iso_queue = q with incrementable
```

The four last conjectures have been found using my proof of concept (without any modification from the user). The combination of all these invariants is inductive.

As we can see, one of these invariants has a $\forall^*\exists^*$ prefix: it would not have been found using the preexisting invariant synthesis tool of IVy. We can also notice that the second invariant that has been found is strictly weaker than the third (it can be removed). It is due to the fact that the second counterexample found was too specific: its structure has imposed some (useless) equality constraints. However, generating the associated invariant and adding it to the set has permitted to find an other counterexample that has given a more general invariant (the third).