Mechanised Verification Patterns for Dafny

Mechanised Verification Patterns for Dafny*

Gudmund Grov, Yuhui Lin and Vytautas Tumas
Heriot-Watt University, Edinburgh, UK, {G.Grov,Y.Lin,vt50}@hw.ac.uk

Abstract. In Dafny, the program text is used to both specify and implement programs in the same language [24]. It then uses a fully automated theorem prover to verify that the implementation satisfies the specification. However, the prover often needs further guidance from the user, and another role of the language is to provide such necessary hints and guidance. In this paper, we present a set of *verification patterns* to support this process. In previous work, we have developed a *tactic language* for Dafny, where users can encode their verification patterns and re-apply them for several proof tasks [16]. We extend this language with new features, implement our patterns in this tactic language and show, through experiments, generality of the patterns, and applicability of the tactic language.

1 Introduction

Dafny [24] is a program verifier and programming language where the specification of desired properties is intertwined with their implementation in the program text. It uses an automated theorem prover to *prove* that the specification is satisfied by the program. A specification serves two purposes: (1) it *specifies* the properties to be proven and acts as a *documentation* of the program, which is desirable to include in the program text; (2) it is used to *guide the prover* if a property cannot be verified without help. This is a necessary evil, which is not desirable and may obfuscate the readability of the program text. We will call this type of specification elements for *proofs*.

The process of creating a proof typically involves changing, and in most cases adding, auxiliary annotations such as assertions and loop (in)variants, as well as manipulation of a *ghost state*: a state that can be updated and used as normal, but is only used for verification purposes and will not be compiled. In addition to increasing the size of the program text, the process of generating proofs can be very time consuming.

In this paper, we investigate and document a set of *verification patterns* that captures common proofs of Dafny programs (§3). We have previously developed *Tacny*, a *tactic language* for Dafny, which enable users to encode and apply verification patterns [16]. In §4 we extend this with new features, before we mechanise the patterns in Tacny and evaluate them on a set of examples (§5). We conclude and discuss relevant and future work in §6.

* This work has been supported by EPSRC grants EP/M018407/1 and EP/N014758/1. Special thanks to Rustan Leino and his colleagues at MSR.

2 Background on Dafny & Tacny

Dafny combines imperative, object-oriented and functional programming language paradigms. It support features such as inductive [25], co-inductive [26] and higher-order [23] types. It uses familiar notations for assignment (`x := e`), declarations (**var** `x := e;`), conditionals (**if** and **if**−**else**) and loops (e.g. **while**). It also supports pattern matching (**match**) and a 'such as' operator, where `x :| p` means that `x` is assigned a value such that `p` holds.

Dafny has been designed for verification. Properties are specified by *contracts* for methods/functions in terms of preconditions (**requires**) and postconditions (**ensures**). To verify a program, Dafny translates it into an *intermediate verification language* called Boogie [4]. From Boogie a set of VCs is generated and sent to the Z3 SMT solver [28]. If it fails, then the failure is translated back to the Dafny code, via Boogie.

In the case of failure, a user must provide guidance in the program text in terms of proof details. The simplest form is to add assertions (**assert**) of true properties in the program text. In the case of loops, we might also provide loop invariants (**invariant**). Loops and recursion have to be shown to terminate and for advanced cases a user needs to provide a variant (**decreases**) to help Dafny prove this.

For more advanced verification tasks, one can make use of the *ghost state*. A ghost variable (**ghost var**) or ghost method can be introduced and used by the verifier. A lemma (**lemma**) is a type of ghost method that can be used to express richer properties, where assumptions are preconditions, and the conclusion becomes the postcondition. The proof is a method body that satisfies the postcondition, given the precondition. We will see examples of this below, but note that standard programming language elements are used in the body of the lemma, which illustrates the close correspondence between proofs and programs.

Tacny is a conservative extension of Dafny with features to implement verification patterns as *tactics* [16]. This tactic language is a *meta-language* for Dafny, where evaluation of a tactic works at the Dafny level: it takes a Dafny program with tactics and tactic applications, evaluates the applications and produces a new valid Dafny program, where tactic calls are replaced by Dafny constructs which tactics have generated.

A *tactic* is a special Dafny ghost method, recognised by the **tactic** keyword. It contains many features to talk *about* a program, and features to *generate* proofs in terms of Dafny by *transforming* the program. A crucial property is that neither the program, nor the actual (non-proof) specification, can be changed – which we call *contract-preserving transformations* [16].

The application of a tactic will transform a tactic call into the Dafny code generated. To illustrate, the following tactic

```
tactic nat_assert()
{ tactic var n :| n in variables();
  assert n ≥ 0; }
```

will first bind n to a local variable where it was called. This binding is in the tactic world. If there are more than one variables then a search branch will be created for each variable (and it will fail if there are none). The result of applying this tactic is that an assertion, which asserts that the variable is positive, replaces the tactic call. A tactic will either be evaluated until the tactic reach the end of the code, or a proof is found. The top-level tactic application (a tactic that is not applied by another tactic) will only succeed if a proof is found on termination. A formal evaluation semantics is given in [16][1].

A design goal for Tacny is to make Tacny intuitive for Dafny users. It therefore makes use of many Dafny constructs, and follows standard Dafny conventions otherwise. As far as possible, the language supports *declarative* (or *schematic*) tactics, i.e. a schematic representation of a proof patterns is given and Tacny is used to fill in the details. A more detailed account of the Tacny features is given in §4, where we describe the new features developed here. Next, we outline some more informal verification patterns that are independent of Tacny.

3 Verification patterns for Dafny

Although Dafny relies on an automatic prover, Dafny proofs should not be seen as automatic. Instead they are called *auto-active* as the proof guidance is abstracted from the underlying prover to the program text. This has had positive usability effects on a *syntactic* level, as one do not need to learn an additional language to conduct proofs; and *conceptually*, as one can think of proofs in terms of programming, rather than the prover. The verification patterns introduced in this section capture proofs at the Dafny level. They are a result of analysing programs in the Dafny repository [1], analysing programs we have developed, and discussion with the developers of IronFleet [19], a large Dafny development consisting of $40K$ lines of proofs [18]. The source code and additional details of the examples can be found on a dedicated web-page [2].

3.1 Patterns as macros

A very simple pattern is to capture repetitive code, possibly with slight variations. This can be seen as a *macro*, as found in some programming languages (e.g. C). In our sense, we see a macro as a named entity of some repetitive code, as illustrated by the following lemmas:

```
lemma minus_dist() ensures ∀ m, n • minus(add(m, n), n) = m;
{ assert ∀ m,n • add(Suc(m),n) = Suc(add(m,n));
  assert ∀ m, n • add(m, n) = add(n, m); }

lemma geq_dist() ensures ∀ m, n • geq(add(Suc(m),n),n) = True;
{ assert ∀ m,n • add(Suc(m),n) = Suc(add(m,n));
  assert ∀ m, n • add(m, n) = add(n, m); }
```

Here the proofs (i.e. body) of these lemmas are identical, and can thus be turned into a macro (see §5). Possible reasons for using macros is to hide details in

[1] The requirement that a tactic has to find a proof is a result of user feedback, and is not required in the semantics described in [16].

code, or to reuse code across verifications tasks. Note that the macro pattern is common in the IronFleet proofs [19].

3.2 Proof by cases and induction

Two common and general proof patterns are *proof by cases* and *proof by induction*. We discuss these together as their representation are very similar in Dafny.

In its simplest form, a *proof by cases* step is achieved by an **if** statement, where the condition is the case to split on. For a proof by *natural induction*, the lemma being proven typically has a natural number `n` as an argument, and the condition is used to separate base from step cases (often `n =0`). In the step case, a recursive call is made to the lemma with `n` decremented.This will reveal the *induction hypothesis* (i.e. the postcondition of the lemma for `n−1`).

To illustrate, consider a mutually recursive definition of `even` and `odd`. The following lemma proves that all natural numbers are either even or odd:

```
lemma even_or_odd(n : nat)
 ensures even(n) ∨ odd(n)
{ if n = 0 ∨ n = 1{ }
  else{ even_or_odd(n−1); }}
```

Note that Dafny has a hard-coded tactic for this type of induction proofs [25]. It normally proves these simple cases where the step case only involves a call to itself. However, it did not work in this case, possibly due to the mutually recursive nature of `even` and `odd`. In cases where the step case needs more work, Dafny's induction tactic will not work (automatically), and more interaction is required (see e.g. [18] for examples). The pattern could also be written in different ways, such as:

```
if n = 0 { return; } ...          if n ≠ 0 { ... }
```

The dots (...) represents the step case. A proof by cases is similar, but without the recursive call. Multiple cases can be achieved by multiple **if** statements.

Another type of induction is when each case is a constructor in an inductively defined data type. In that case, a **match** statement is used, which will also perform a suitable binding of the variables in the constructor. For constructors with recursive arguments, a recursive call to the same lemma is made. This is a proof technique called *structural induction* [7]. To illustrate, consider the following inductive data type:

```
datatype aexp = N(n: int) | V(x: vname) | Plus(0: aexp ,1: aexp)
```

Here, the `Plus` constructor has two recursive arguments (i.e. they are of the same `aexp` type). Omitting irrelevant parts, a proof of a lemma by structural induction will then look as follows in Dafny:

```
lemma AsimpConst(a: aexp , s: state) ...
{  match a
    case N(n) ⇒
    case V(x) ⇒
    case Plus(a0 , a1) ⇒ AsimpConst(a0 , s); AsimpConst(a1 , s);}
```

This is a very common proof technique when working with inductive data types. In fact, most interactive theorem provers will automatically generate an induction principle when defining inductive data types such as aexp. As shown in [16], a similar pattern can be applied for *co-induction* [26].

3.3 Proof by contradiction

Another common proof pattern (for classical logics) is *proof by contradiction*. In order to prove a property P, this amounts to assuming $\neg P$ and derive *false* from it. Again, this technique is frequently used in [18]. To implement it in Dafny, the negated property $\neg P$ becomes the condition of an **if** statement, with **false** asserted at the of the body of the **if** statement. The following example illustrates this pattern:

```
lemma set_inter_empty_contr(A: set<int>, B: set<int>, x: int)
 requires x in A ∧ A * B = {}
 ensures ¬(x in B)
{ if x in B {
    assert x in A * B;
    set_eq_simple(A*B,{},x);
    assert x in {};
    assert false; }}
```

Here, set_eq_simple states that if x is in A∗B (the intersection of A and B) then x is in {}. This (rather trivial) lemma application was required for the proof.

3.4 Loop invariants

The discovery of sufficiently strong loop invariants is one of the most important parts of verifying imperative code. A substantial amount of work has been conducted to automate such discovery. Techniques include abstract interpretation [11], constraint-based techniques [10,17], inductive logic programming [13], symbol elimination [20] and predicate abstraction [29]. Dafny uses abstract interpretation (at the Boogie-level) [4]. Still, there are many cases where the user has to provide loop invariants manually in order to verify code. Below we outline three patterns for "manual" loop invariant discovery.

The Gries & van de Snepscheut approach In their systematic approaches to program development, Gries [15] and van de Snepscheut [32] developed several heuristics for verified program construction. Here, we adapt their heuristics for loop invariant discovery (assuming the actual code has been provided), resulting in the following patterns where an invariant is created by: (*i*) deleting a conjunction in a postcondition; (*ii*) replacing a constant of a postcondition with a local variable; and (*iii*) enlarging the range of a variable of a loop guard. The following example illustrates all of these loop patterns:

```
method FindMax(a: array<int>) returns (i: int)
   requires a ≠ null ∧ a.Length > 0
   ensures (0 ≤ i < a.Length)
   ensures (∀ k • 0 ≤ k < a.Length ⟹ a[i] ≥ a[k])
```

```
{  var idx, j, i := 0, 0, 0;
   while (idx < a.Length)
     invariant idx ≤ a.Length //(iii)
     invariant 0 ≤ i < a.Length //(i)
     invariant ∀ k • 0 ≤ k < idx ⟹ a[i] ≥ a[k] //(ii)
   { if (a[idx] > a [i]) { i := idx; }
     idx := idx + 1; }}
```

Use of guards Another pattern seen (albeit not as commonly) combines (negated) loop guards and guards of conditionals in the invariant. This is illustrated in the following example:

```
method Main() {
     var a,b,c,i := 0,−1,0,100;
     while a ≠ b
      invariant ¬(c < i) ⟹ ¬(a ≠ b)
      decreases i−c
     {   b, c := a, c + 1;
         if (c < i) {
          a := a + 1;}}}
```

Use of recursive functions One may argue that it is easier to reason in the functional fragment of Dafny compared with imperative code. A pattern exploring this generates a recursive function that is defined in the same way as the loop, and proves that this function satisfies the desired postcondition (of the method). This is typically proven by induction. A loop invariant is required to relate the function to the loop body. To illustrate, the following code has a loop invariant that relates the code to a function called find_max_aux:

```
method find_max_idx (a : seq<int>) returns (x : int)
 requires a ≠ []
 ensures  0 ≤ x ≤ |a| − 1
 ensures ∀ i • 0 ≤ i ≤ |a| − 1 ⟹ a[i] ≤ a[x]
{ var x,y,N,A := 0,|a|−1,0,|a|−1;
  while (x ≠ y)...
   invariant find_max_aux(a,x,y) = find_max_aux(a,N,A)
  {  if (a[x] ≤ a[y]){x := x + 1;}
     else {y := y − 1;}}
  proof_find_max_aux(a, N, A);}
```

The function is defined as follows:

```
function find_max_aux(a:seq<int>, x:int, y:int):int ... {
  if |a[x .. y+1]| = 1 then x
  else if (a[x] ≤ a[y]) then find_max_aux(a, x + 1, y)
  else find_max_aux(a, x, y − 1) }
```

As can be seen, the code of find_max_aux captures the body of the **while** loop[2]. The proof_find_max_aux lemma relates the function to the postcondition:

[2] The generation of this function happens to be the inverse of the well-known *tail-recursion to loop* compiler optimisation [8].

```
lemma proof_find_max_aux(a: seq<int>,x: int, y: int) ...
  ensures ∀ i • x ≤ i ≤ y ⟹ a[i] ≤ a[find_max_aux(a, x, y)]{}
```

4 Tactics for Dafny (Tacny)

To mechanise the verification patterns as Dafny tactics (§5), new features of the Tacny language [16] are required. Here we describe these features, and outline some existing important language properties used in the tactics of §5. It will be clear which parts are from [16] – the rest, which is summarised in Appendix A, should be considered as a contribution.

A design goal of Tacny is to make the language as familiar as possible to users by exploiting known Dafny constructs and conventions. As far as possible, we try to support *declarative* features in the tactics, where *schematic* representations of proof patterns are given as opposed to a set of procedures. Consequently, a tactic should look like Dafny code, which we believe will be more familiar and intuitive for users. This has been inspired by declarative tactic languages for interactive theorem provers (e.g. [3]).

A tactic is a ghost method, identified by the **tactic** keyword, for example:

```
tactic ex_tac(v : Element, t : Term, tac : Tactic)
  requires P
  ensures Q
{ ... }
```

Types As this is a meta-level language, constructs to talk *about* a program are required. To achieve this, two new types were introduced in [16]: Element captures a named element of the Dafny program text, such as a variable, method or lemma; while Term refers to the term representation of a formula (which can then be manipulated). Here, we introduce a third type Tactic, which makes a tactic a first class value. A (fully instantiated) tactic application can be passed to another tactic and used therein. A limitation is that it has to be fully instantiated, meaning that proper higher-order programming, where tactics can take arguments, is not (yet) supported. We use the Tactic type extensively in §5.

Statements [3] When used within Tacny, Dafny constructs have two different uses: in a declarative tactic they are part of an outline of code to be generated by Tacny, and we call this the *object-level*; they can also be used to control evaluation of tactics, and in this case they are at the *tactic-level*. It is a design decision if these should be separated syntactically, i.e. separate constructs for each level (meaning additional syntax) vs. the same constructs for both levels (meaning different semantics for the same syntax). We are using a combinations of these approaches.

Both **if** and **while** statements are used across the object-level and the tactic-level: they belong to the tactic-level if Tacny can evaluate the condition (to either

[3] All the statements, including the atomic tactics (modular some name changes) were introduced in [16].

true or **false**); and to the object-level if not[4]. The justification for this is that such constructs are familiar for users. Variable declarations, on the other hand, have been syntactically separated as the distinction is less clear. This is achieved by preceding a tactic-level declaration by **tactic**[5]:

```
tactic var x := e;
```

If **tactic** is omitted, then variable x will be in the object-level and thus part of the code to be generated. One can shorten **tactic var** and just write **tvar**.

In addition to assignment (x := e), the 'such as' operator (x :| p) is supported, albeit in a restricted form. Here, we need to be able to enumerate all possible values that x can have, and Tacny will generate a branch in its search space for each possibility. The Tacny statement, s || t, will either apply statement s, or statement t. Tactic calls are supported, which become normal method calls. To develop new tactics, a set of hard-coded and low-level *atomic tactics* are provided by the Tacny system, while expressions are extended with a set of *lookup functions* about the program. These are discussed next.

Atomic tactics The simplest atomic tactic involves (generating code for) a lemma or ghost method application. Following our declarative approach, this is represented exactly like a method call. For example, assume **tvar** m,a := lem,v, where lem is a lemma and v is a variable. The statement m(a+1); within a tactic will result in code containing the method call lem(v+1)[6]. Assertions, invariants and variants are handled using existing Dafny constructs, as can be seen below:

```
assert a = 1;        invariant a = 1;        decreases a;
```

Tacny will instantiate the tactic-level constructs (a) to the object-level counterpart (v). Note that if **invariant** or **decreases** is used in the body of a loop (or method for the latter), then they will be added to the loop invariant (or method declaration for variants) and not at the point of the tactic call as normally happens. They can also be called where the invariant/contract is stated.

The explore(m: Element,args: Seq<Element>) tactic generates all possible application of ghost method/lemma m with arguments taken from args. The proof of AsimpConst (§3.2) illustrated the use of the **match** statement to do a case analysis of all constructors for a variable v of an inductively defined type. The tactic **tactic match** v { ...} will generate such a match. Here, v is of type Element, and its body (...) contains tactics to be applied for each constructor. **tmatch** is a shorthand notation for **tactic match**[7].

Lookup functions and expression-level atomic tactics One often need properties of the program in tactics, and Tacny keeps track of a *context* that contains

[4] Meaning, code such as **while true** {...} cannot be generated.

[5] This naming convention is used for ghost variables in Dafny, which in certain cases needs to be declared as **ghost var**.

[6] If a sequence is given as argument for a method that does not expect a sequence, then Tacny will automatically unroll the sequence into multiple arguments.

[7] In [16], explore was called perm and **tactic match** called cases.

such information. Several "look-up" functions from the context are provided. lemmas(), methods() and functions () return the name of available lemmas, methods and functions as sequences of Elements (Seq<Element>). caller () returns the name (type Element) of the method/lemma/function in which the tactic call was made[8]. The functions below works on the original caller. They also accept an optional argument (omitted below), allowing users to look up these properties on other methods and lemmas. preconditions () and postconditions () return sequences of Terms holding all the preconditions or postconditions; args () and variables () return the local arguments and variables of the element (type Seq<Element>); if_guards () and loop_guards() return sequences of Terms, holding the guard of all conditionals and loops, respectively, while loop_guard() returns the loop guard of the loop where the tactic call is made (and fails otherwise).

The predicates is_inductive (v: Element) and is_nat (v: Element) check if the given elements are variables of an inductively defined type or a natural number, respectively; is_inductive can also be applied to a constructor to check if any of its arguments are recursive. eq_type(x: Expr,y: Expr) checks if two expressions are of the same type. When applied within the body of a **match** or **tmatch**, get_constructor () will return a pair of the constructor name (Element) and its arguments (Seq<Element>).

consts(t: Term) returns all constants of t (as a sequence of Terms); split (t: Term,sep: Term) splits all occurrences of sep in t into (a sequence of) separate terms[9]; replace(x: Element,y: Element,z: **seq**<Element>) replaces all occurrences of x with y in z; subst(t: Term,m: **map**<T,U>) applies the substitutions of map m in t. The map is overloaded: it allows T and U to be of types Term, Element or string, where the latter two are treated as named constants. Finally, explore can also be applied as an expression, and returns a term with a function application.

Tactic calls within expressions A limitation of [16] was that tactics could only be used within statements. Here, we extend the framework with tactic applications within expressions. These have the syntax[10]:

```
function tactic expr_tac(..) {..}
```

Note that a function tactic will not necessarily generate any code; it will return a Term which may generate code depending on where the call is made (and possible generate multiple search branches). E.g. it will not generate code on the r.h.s of a **tvar**, but it will when called in a Tacny tactic such as:

```
assert expr_tac(..);
```

[8] If this is a nested tactic call, then it refers to the name of the method/lemma/function that called the parent tactic.

[9] For example, split (A ∧ B, ∧) will return [A,B].

[10] This syntax is inspired by the syntax for **function method** used in Dafny.

Tactic-level contracts and annotations A new feature added is to support annotations/contracts at the tactic-level. These are interpreted dynamically, and are used to cut-off invalid branches as early as possible: e.g. if a tactic-level assertion or precondition fail (returns **false**), then the tactic will fail. We can write an assertion P as

```
tactic assert P;
```

or just tassert, while ex_tac illustrates the tactic contracts. This is used in §5.

Runtime improvements We have made improvements in the runtime and memory usage of tactics as a result of improved static checking, lazy evaluation and improved support for different search strategies. On our test data [31], an average speed-up of 44% and memory usage reduction of 23% was achieved (and these increased with the size and complexity of tactics). The details are omitted for space reasons – see Tumas' honours thesis for details [31].

5 Verification patterns implemented as Dafny tactics

With the new extensions to the Tacny language, we can now implement the verification patterns from §3 as Dafny tactics, and apply them in the Tacny tool. The results from this application is summarised at the end of this section, while all the code is available from [2][11].

5.1 Tactics as macro expansions

In §3, we saw that a common pattern is to extract repeated code, possibly with slight variations, as a *macro*. It does not have to contain an underlying high-level pattern, so in many ways this is just syntactical. Instantiating macros is normally called *macro expansion*, and we therefore see tactic applications as macro expansions.

The proof of lemmas minus_dist and geq_dist are identical (see §3), so the macro becomes the code within their proofs:

```
tactic dist_macro()
{ assert ∀ m,n • add(Suc(m),n) = Suc(add(m,n));
  assert ∀ m, n • add(m, n) = add(n, m); }
```

The lemmas using this tactic will then only contain a tactic call:

```
lemma minus_dist() ensures ∀ m, n • minus(add(m, n), n) = m;
{ dist_macro(); }
lemma geq_dist() ensures ∀ m, n • geq(add(Suc(m),n),n) = True;
{ dist_macro(); }
```

When there are slight variations one can either provide the parts that varies as arguments, or introduce search into the tactic.

In Dafny, commonalities can often be captured as as a lemma or a method. However, due to modularity, they require that all assumptions are explicitly

[11] The supported tool syntax has some minor limitations and thus deviates slightly.

stated as preconditions, and that all the relevant outcomes are explicitly stated as postconditions. If the goal is to capture some repetitive code as a macro, then, in most cases, stating these assumptions and outcomes can be very tedious, making lemmas unsuitable for this task. As tactics replaces a call with the generated code, such explicit statements are not required, thus making it a more suitable representation.

5.2 Proof by cases and induction

As in §3, we treat induction and case-split together as the former needs a case-split first in Dafny. The following tactic is a generic tactic for *natural induction*:

```
tactic nat_ind(cond: Tactic, base: Tactic, step: Tactic)
{   if cond() {  base(); }
    else{ tactic var m := caller();
          tvar a :| a in args()
          tactic assert is_nat(a);
          m(a-1);
          step(); }}
```

The tactic takes three tactics as arguments: the first (cond) is used to generate the condition (e.g. n = 0 when n is the inductive argument); the second (base) is used to handle the base case; and the third (step) is used for the step case. The first four lines of the step case will generate a recursive call to reveal the induction hypothesis. This is the only difference with proof by cases, where these four lines are omitted.

For the even_or_odd lemma, the condition is defined using a function tactic:

```
function tactic nat_ind_cond()
{ tvar a :| a in args() ∧ is_nat(a);
  a=0 ∨ a=1 }
```

The lemma can then be proved by the call: nat_ind(nat_ind_cond(), id(), id()).

For *structural induction*, a **match** statement is generated using our **tactic match** tactic, with recursive calls for the recursive constructors:

```
tactic struct_ind(v: Element, t: Tactic)
 requires is_inductive(v);
{ tactic match v {
    tvar c, cargs := get_constructor();
    if is_inductive(c) {
       tvar m, args := caller(), args();
       tvar i := 0;
       while i < |cargs|
       { if eq_type(v, cargs[i])
         { m(replace(v, cargs[i], args) }
         i := i + 1; }
    t(); }}
```

The tactic takes as arguments: a variable v of an inductively defined type (ensured by the precondition); and a tactic t to be applied to each case. For the recursive constructors, a recursive call to the caller is made for each (constructor) argument of the same type of v, with v replaced by this argument.

5.3 Proof by contradiction

Proof by contradiction involves assuming the negation of the desired property and deriving false. For Dafny, the property is often (one of) the postcondition(s). The following contr tactic picks one postcondition, and shows, using an **if** condition, that its negation will result in a contradiction. The method takes a tactic as argument that is used to derive the contradiction:

```
tactic contr(tac : Tactic)
{  tactic var post :| post in postconditions();
   if ¬post {
    tac();
    assert false; }}
```

For our set_inter_empty_contr lemma, we can follow the macro expansion approach and give the code directly:

```
tactic tbody()
{    assert x in A * B;
     set_eq_simple(A*B,{},x);
     assert x in {}; }
```

The lemma is verified by the following call: cntr(tbody()).

5.4 Loop patterns

The Gries & van de Snepscheut approach In §3.4, we described an approach that we called the 'Gries & van de Snepscheut approach'. It contains three patterns, and we implement each of them as a tactic:

```
tactic delete_conj_post ()
{ tvar post :| post in postconditions();
  tvar inv :| inv in split(post,∧);
  invariant inv; }

tactic const_to_var()
{ tvar post := postconditions();
  tvar inv0 :| inv in split(post,∧);
  tvar cons :| const in consts(post');
  tvar v :| v in variables();
  invariant subst(inv0, map[c := v]); }

tactic strengthen_guard()
{ invariant subst(loop_guard(),map["<":="≤", ">":="≥"]);}
```

A simple implementation of an overall pattern applies them one after another:

```
tactic GvdS_approach()
{ delete_conj_post(); const_to_var(); strengthen_guard(); }
```

Note that this rules out multiple application of one pattern and would fail if either of them fail. For space reasons we have omitted more generic and complex versions. This tactic is able to discover the invariants for the FindMax lemma and thus verify it.

Use of guards The second loop pattern is a combination of (possibly negated) guards:

```
tactic inv_guard()
{  tvar xx :| xx in if_guard() + loop_guards();
   tvar yy :| yy in if_guard() + loop_guards();
   tvar x :| x = xx ∨ x = ¬xx;
   tvar y :| y = yy ∨ y = ¬yy;
   invariant x ⟹ y; }
```

The inv_guard tactic projects all the guards from **if** and **while** statements. It then creates an invariant, which is an implication where both the antecedent and consequent is a guard or a negated guard. This tactic generates the invariant and proves the Main method of §3.4:

Use of recursive functions Tacny only partly supports the 'use of recursive functions' pattern of §3.4. The pattern requires: generation of a function (from the loop body); generation of a lemma (to connect the function and postcondition); and a lemma call outside the loop body (i.e. on the loop exit). Currently, lemma and function generation is not (yet) supported. This is however planned future work (see §6). A limited version can be implemented if we assume the existence of such function, lemma and lemma call. Tacny can then generate the required loop invariant, which link the function with the loop body:

```
tactic rec_func (func: Element)
{ tvar args := variables() + args();
  tvar lhs := explore(func, args);
  tvar rhs := explore(func, args);
  invariant lhs = rhs; }
```

A tactic could have generated the lemma call too, however this requires the loop invariant to be generated within the loop, whilst the call has to be outside the loop body. Generated such code multiple places from a tactic is not currently supported and discussed further in §6.

5.5 Summary & results

Fig. 1 summarises the results from our experiments with the patterns as tactics. Further details and code can be found on a dedicated web-page [2]. The table on the l.h.s. shows the total number of pattern instances (**Inst**) and the number of different tactics implemented (**Tactics**). In order to get an idea of time and memory usage, the r.h.s. summarises the run-time (X-axis) and search space size in terms of the number of nodes/steps (Y-axis) using logarithmic scales. Many tactics were re-used across methods and programs, but in some cases slightly different implementations were required (e.g. multiple different macro expansions). In most cases, Tacny used less than 10 seconds to run on a standard laptop (Intel i7 with 8GB RAM). On average, Boogie accounted for around 95% of the execution time, highlighting the importance of improving the integration with Boogie and Dafny. This is the reasons for the two outliers in Fig. 1 (right), which has a considerable larger search space and runtime compared with the other examples.

Pattern	Insts Tactics
Cases	15 4
Macro	8 5
Cntr	3 1
GvdS	6 3
Guard	1 1
Recur	4 2

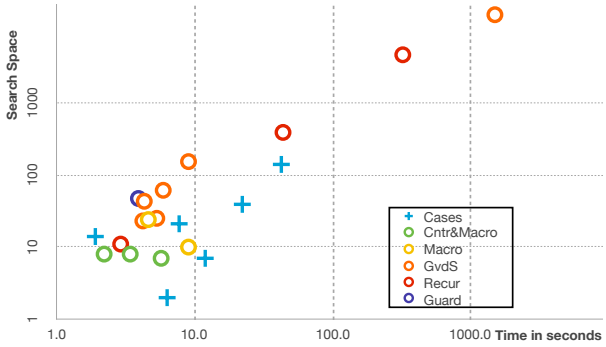

Fig. 1. Evaluation results

6 Related work, conclusion & future work

One contribution has been a set of (informal) *verification patterns*, extracted from various sources including [1,18,15,32], which we believe could support novices with their proofs. Surprisingly little work has been done in capturing and documenting verification patterns for mechanised systems[12]. Freitas and Whiteside [14] captures a set of proof patterns for formal methods conjectures in an interactive theorem proving (ITP) setting. Bundy's *proof plans* [5] is a more formal representation of proof patterns for meta-level reasoning, and includes work for algebraic equations in the PRESS system [30] and the rippling strategy for inductive proofs [6]. There is also a book by Joshi on proof patterns for mathematics [21], but this does not address mechanised proofs.

Although undocumented, patterns are still used within most theorem proving based system, implemented as heuristics or tactics in some cases. For example, Dafny has an "induction tactic" [25] to automate simple, yet common, cases of inductive proofs, while ITP systems such as Isabelle and Coq, will have a large collection of tactics to support users. These systems have also started to move the language where tactics are implemented from its implementation language (typically ML variants) to the proof language (e.g. LTac for Coq [12] and EisBach for Isabelle [27]). Autexier and Dietrich [3] has taken this even further and developed a *declarative tactic language* where tactics are written schematically. Inspired by declarative tactics, our work is analogous to [3,27,12], as users can encode proof patterns (tactics) in the program text of Dafny, as opposed to its implementation language (C#), as was the case in e.g. [25].

Building on our initial tactic language [16], our main contributions have been the encoding of the discovered proof patterns as Dafny tactics, together with the necessary extensions to the language. We have shown that the patterns and tactics are generic by applying them to multiple examples, with a reasonable running time. In addition to being an exercise in encoding Dafny tactics, we have shown feasibility and highlighted invaluable language features. Firstly, the language gives a tactic developer freedom to focus on encoding the patterns without concerns of soundness issues (which was the case in [25]), as the actual verification is still conducted by Dafny[13]. The ability to pass tactics as argu-

[12] Klein's FM 2014 keynote also addressed this limitations and its importance.
[13] Under the proviso of *contract preservation* as discussed in §2 and formalised in [16].

ments has enabled us to develop more generic tactics. However, in many cases it would have been useful to improve tactic composition by supporting tactics *with arguments* to be passed between tactics. Dafny's type system now supports higher-order features [23]. A next step is to improve the type system in Tacny, and incorporating such features would be beneficial.

The code fragment `x : | x in P` is used extensive in our tactics and a shorthand notation for this will be useful[14]. We are also considering automatically binding variables that occurs frequently (as in [27]) to reduce the code that users have to write, e.g. `vars` (or `Tacny.vars`) for `variables ()`. Instead of explicitly introduce branches through assigning a variable with `: |`, a similar notation could be used. For example, in cases where sequences are not expected, `split (P ∧ Q,∧)` could automatically be bound to `P` in one branch and `Q` in another branch.

We would also like to include features to generate new lemmas and functions, and investigate how to encode tactics that generate code at different places in a method. This will help us to encode the full 'recursive function' loop pattern. Dafny's (experimental) refinement feature uses a '...' notation to step over code [22], which would serve as a starting point. Another limitation is that we need to hard-code functions, such as `replace` in the `struct_ind` tactic, which could have been implemented in Dafny directly (user-defined Dafny functions are not supported at the tactic-level). This will require us to write an interpreter, or possible utilise Dafny's existing compiler into C# (our implementation language).

Following from user feedback, we have improved the language of [16], and *user evaluations* will also play crucial role to ensure a user friendly language in the future. We are now in the process of developing a tighter integration with Dafny, Boogie and the Dafny IDE, where failure-handling and features for debugging tactics are high on our agenda; we believe that these are crucial for adaptation. This will hopefully help us addressing the Boogie bottleneck (see §5.5).

Some of our tactics can be found in ITP systems: e.g. proof by contradiction, natural induction and structural induction are common; while Dafny can already automate simple inductive lemmas. The `explore` tactic is a simple form of *term synthesis* at the Dafny level, as used in e.g. HipSpec for Haskell [9]. We plan to implement tactics for richer explorations, supporting more than single statements and conditionals. We have already discussed automated approaches for loop invariant discovery in §3.4. A key distinction from these techniques is that tactics follows a more *human-oriented* approach, where the developer's (mental) pattern is encoded as a tactic.

Other important challenges include the discovery of new patterns and their corresponding tactic implementations, and to address *scalability* of the approach. For example, a common pattern used in IronFleet is to first unfold universal quantification, set up a proof by contradiction and then apply some lemmas afterwards [19]. We support some of these components, but would like to complete the circle and see if we can develop a tactic for the complete pattern, which also handles the size of this program.

[14] For example, Event-B has an operator $x :\in P$ to express this.

References

1. Dafny Website. `research.microsoft.com/dafny`.
2. The Tacny project: FM 2016 information. `https://sites.google.com/site/tacnyproject/fm-2016`. Accessed: 29.05.2016.
3. S. Autexier and D. Dietrich. A Tactic Language for Declarative Proofs. In *ITP'10*, volume 6172 of *LNCS*, pages 99–114. Springer, 7 2010.
4. M. Barnett, B-Y. E. Chang, R. DeLine, B. Jacobs, and K. R. M. Leino. Boogie: A modular reusable verifier for object-oriented programs. In *FMCO*, volume 4111 of *LNCS*, pages 364–387. Springer, 2006.
5. A. Bundy. A science of reasoning. In *Computational Logic - Essays in Honor of Alan Robinson*, pages 178–198, 1991.
6. A. Bundy, D. Basin, D. Hutter, and A. Ireland. *Rippling: Meta-level Guidance for Mathematical Reasoning*, volume 56 of *Cambridge Tracts in Theoretical Computer Science*. Cambridge University Press, 2005.
7. Rod M Burstall. Proving properties of programs by structural induction. *The Computer Journal*, 12(1):41–48, 1969.
8. Rod M Burstall and John Darlington. A transformation system for developing recursive programs. *Journal of the ACM (JACM)*, 24(1):44–67, 1977.
9. K. Claessen, M. Johansson, D. Rosén, and N. Smallbone. Automating inductive proofs using theory exploration. In *CADE-24*, pages 392–406. Springer, 2013.
10. M. A. Colón, S. Sankaranarayanan, and H. B. Sipma. Linear invariant generation using non-linear constraint solving. In *CAV*, pages 420–432. Springer, 2003.
11. P. Cousot and N. Halbwachs. Automatic discovery of linear restraints among variables of a program. In *Proceedings of the 5th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 84–96. ACM, 1978.
12. D. Delahaye. A Tactic Language for the System Coq. In *LPAR-7*, pages 85–95, 2000.
13. M. D. Ernst, J. H. Perkins, P. J. Guo, S. McCamant, C. Pacheco, A. S. Tschantz, and C. Xiao. The Daikon system for dynamic detection of likely invariants. *Science of Computer Programming*, 69(1):35–45, 2007.
14. Leo Freitas and Iain Whiteside. Proof Patterns for Formal Methods. In Cliff Jones, Pekka Pihlajasaari, and Jun Sun, editors, *Formal Methods*, pages 279–295. Springer, 2014.
15. David Gries. *The Science of Programming*. Springer, 1st edition, 1987.
16. Gudmund Grov and Vytautas Tumas. Tactics for the Dafny Program Verifier. In Marsha Chechik and Jean-François Raskin, editors, *22nd International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 36–53. Springer, 2016.
17. A. Gupta and A. Rybalchenko. Invgen: An efficient invariant generator. In *CAV*, volume 5643 of *LNCS*, pages 634–640. Springer, 2009.
18. Chris Hawblitzel, Jon Howell, Manos Kapritsos, Jacob R Lorch, Bryan Parno, Michael L Roberts, Srinath Setty, and Brian Zill. Ironfleet: Proving practical distributed systems correct. In *Proceedings of the 25th Symposium on Operating Systems Principles*, pages 1–17. ACM, 2015.
19. Chris Hawblitzel, Jay Lorch, and Bryan Parno. Personal discussions (December, 2015).
20. K. Hoder, L. Kovács, and A. Voronkov. Invariant generation in Vampire. In *Tools and Algorithms for the Construction and Analysis of Systems*, pages 60–64. Springer, 2011.

21. Mark Joshi. *Proof Patterns*. Springer, 2015.
22. Jason Koenig and K Rustan M Leino. Programming language features for refinement. 2015.
23. K. R. M. Leino. Types in Dafny. `http://research.microsoft.com/en-us/um/people/leino/papers/krml243.html`. Manuscript KRML 243. 27 February 2015.
24. K. R. M. Leino. Dafny: An automatic program verifier for functional correctness. In *LPAR*, volume 6355 of *LNCS*, pages 348–370. Springer-Verlag, 2010.
25. K. R. M. Leino. Automating induction with an SMT solver. In *VMCAI*, volume 7148 of *LNCS*, pages 315–331. Springer, 2012.
26. K. R. M. Leino and M. Moskal. Co-induction simply. In *FM 2014: Formal Methods*, pages 382–398. Springer, 2014.
27. D. Matichuk, M. Wenzel, and T. Murray. An Isabelle proof method language. In *Interactive Theorem Proving*, pages 390–405. Springer, 2014.
28. L. Moura and N. Bjørner. Z3: An efficient SMT solver. In *TACAS*, volume 4963 of *LNCS*, pages 337–340. Springer, 2008.
29. S. Srivastava and S. Gulwani. Program verification using templates over predicate abstraction. In *ACM Sigplan Notices*, volume 44, pages 223–234. ACM, 2009.
30. Leon Sterling, Alan Bundy, Lawrence Byrd, Richard O'Keefe, and Bernard Silver. Solving symbolic equations with press. *Computer Algebra*, pages 109–116, 1982.
31. V. Tumas. *Search space reduction for Tacny tactics*. Honours thesis, Heriot-Watt University, 2016. Available from `https://sites.google.com/site/tacnyproject/`.
32. Jan L. A. van de Snepscheut. *What Computing is All About*. Springer, 1993.

A Summary of new Tacny features

This paper has extended and improved Tacny from the version presented in [16] as follows:

- A new type Tactic that makes a tactic a first class value is introduced.
- **function tactic** and tactic applications within expressions are now supported.
- Contracts for tactics, and tactic-level assertions have been added.
- Several new atomic tactics and lookup functions are supported, including: caller (); preconditions (); postconditions (); if_guards (); loop_guards(); is_inductive (v: Element); is_nat (v: Element); eq_type(x: Expr,y: Expr); get_constructor (); consts(t: Term); split (t: Term,sep: Term); replace(x: Element,y: Element,z: **seq**<Element>); subst(t: Term,m: **map**<T,U>); and explore as an expression.
- Considerable runtime improvements have been achieved.
- The syntax is improved to align with Dafny conventions and declarative tactics. For example: cases has become **tactic match** (or **tmatch**); tactic-level variable declarations have changed from **var** to **tactic var** (or **tvar**); Dafny-level variable declarations have changed from **fresh var** to **var**.