Tactics for the Dafny Program Verifier

Gudmund Grov(✉) and Vytautas Tumas(✉)

School of Mathematical and Computer Sciences,
Heriot-Watt University, Edinburgh, UK
{G.Grov,vt50}@hw.ac.uk

Abstract. Many modern program verifiers are based on automated theorem provers, which enable full hiding of proof details and allow users to focus all their effort on the program text. This has the advantage that the additional expertise of theorem provers is not required, but has the drawback that when the prover fails to verify a valid program, the user has to annotate the program text with guidance for the verifier. This can be tedious, low-level and repetitive, and may impact on the annotation overhead, readability of the program text and overall development time. Inspired by proof tactics for interactive theorem provers [19], a notion of 'tactics' for the state-of-the-art Dafny program verifier, called *Tacny*, is developed. With only minor extensions to the Dafny syntax, a user can encode high-level proof patterns as *Dafny tactics*, liberating herself from low-level and repetitive search tasks, whilst still working with familiar Dafny programming constructs. Manual search and guidance can be replaced with calls to such tactics, which will automate this task. We provide syntax and semantics for Tacny, and show feasibility through a prototype implementation, applied to several examples.

1 Introduction

Properties that programs should satisfy are commonly expressed by *contracts*: given a precondition the program guarantees that a given postcondition holds. Many modern *program verifiers* can then be used to verify that a program satisfies its contract by automatically generating verification conditions (VCs), which are sent to an automated theorem prover. Failure to prove VCs will then be highlighted in the program text, and the user must then update the code with *auxiliary annotations* to guide the proof. A feature of this approach is that all interaction, including proof guidance, is conducted in the program text using a single programming language. Spec# [6], VCC [9], Verifast [24], Dafny [27], and SPARK 2014 [34] are program verifiers that follow this approach.

The ability of users to guide the prover for the cases where the underlying theorem prover fails to verify a *correct* program is crucial. Such guidance involves

This work has been supported by EPSRC grants EP/M018407/1 and EP/J001058/1. We would like to the anonymous reviewers for excellent suggestions, thank our collaborators Rustan Leino (who suggested the tactic and one case study), Rosemary Monahan, Altran and Escher.

M. Chechik and J.-F. Raskin (Eds.): TACAS 2016, LNCS 9636, pp. 36–53, 2016.
DOI: 10.1007/978-3-662-49674-9_3

changing, and in most cases adding, auxiliary annotations, and, in many cases, 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.

With the exception of generic *lemmas* that can be reused in e.g. Dafny, the support for reuse of previous verification tasks in order to reduce the required user interaction is limited. In particular, there is no support for users to encode knowledge of common "proof" steps used for verification tasks. Some trial-and-error is involved as several known "verification patterns" may be attempted, and there could be multiple options for each of them. For these cases, the verification guidance process can be unnecessarily tedious and cost-ineffective.

This paper presents a novel extension to program verifiers, by extending the Dafny program verifier with a *tactic* language where users can encode more abstract and reusable verification patterns. Our hypothesis is that

> *it is possible to abstract over low-level manual proof guidance by encoding high-level and reusable verification patterns in the program text of Dafny.*

The work is inspired by *proof tactics* for *interactive theorem provers* (ITPs), which allow users to encode re-usable proof patterns [19], and our main contribution is the introduction of such a tactic language on top of Dafny. A high-level introduction is given in Sect. 2 before giving the details of the syntactic extension and semantics in Sect. 3, and the implementation as the Tacny system in Sect. 4. We believe that Dafny tactics can be used to (*i*) *reduce the annotation overhead*[1]; (*ii*) *reduce development time*; and (*iii*) *increase readability* of the program text by abstracting low-level proof details. Based upon experiments in Tacny, Sect. 5 provides some evidence for (*i*), Sect. 6 contains related work; while we conclude and discuss future work, including how we plan to address (*ii*) and (*iii*), in Sect. 7.

2 Dafny Tactics by Example

Dafny [27] is a programming language and program verifier for the .NET platform, developed by Microsoft Research. The language is an imperative object-oriented language, containing both methods and proper functions (i.e. without side-effects). It also supports advanced features such as inductive [28] and co-inductive [29] datatypes and higher-order types [26]. 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[2].

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* (IVL)[3] called Boogie [5]. From Boogie a set of VCs is generated

[1] To illustrate, [37] reported on 4.8 Lines of Annotation for each Line of Code.
[2] Full details of all examples and results in the paper available from [2].
[3] An IVL can be seen as a layer to ease the process of generating new program verifiers.

and sent to the Z3 SMT solver [35]. 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[4]. 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.

To illustrate Dafny, consider a simple compiler for arithmetic expressions[5]. Here, an inductive data type is used to capture arithmetic expressions as numbers, variables or addition:

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

A state s is a Dafny map from vnames to integers; Total(s) states that the state s is total; aval(a,s) is the evaluation of the arithmetic expression a over the state s; while asimp_const(a) performs constant folding of arithmetic expression a: constants added together are recursively replaced by their sum. The following lemma proves that constant folding preserves the behaviour for a total state:

```
lemma AsimpConst(a: aexp, s: state)
  requires Total(s);
  ensures aval(asimp_const(a),s) = aval(a,s);
{  match a
    case N(n) ⇒
    case V(x) ⇒
    case Plus(a0,a1) ⇒ AsimpConst(a0,s); AsimpConst(a1,s); }
```

The proof follows by structural induction: using pattern matching, a case is introduced for each constructor and for the recursive case the lemma is applied recursively for each argument. Next, consider another example[6] where a list of expressions is defined as:

```
datatype List = Nil | Cons(Expr, List)
```

SubstL(l,v,val) is a function that replaces variable v with value val for each expression in list l. The following lemma proves that SubstL is idempotent:

[4] We assume correct programs: verification may also fail because the program is incorrect and in this case the program (or specification) needs to change.

[5] This example is taken from NipkowKlein-chapter3.dfy on the Dafny webpage.

[6] Substitution.dfy, also taken from the Dafny webpage; both examples available at [2].

```
lemma Lemma(l: List ,v: int , val: int )
 ensures SubstL(SubstL(l ,v, val),v, val) = SubstL(l ,v, val); {
{ match l
    case Nil ⇒
    case Cons(e, tail) ⇒Theorem(e,v, val); Lemma(tail ,v, val); }
```

This proof follows more or less the same pattern, with the difference that a separate lemma Theorem is applied for the first case, which shows idempotence for substitution of a single expression. Theorem is not discussed further here.

AsimpConst and Lemma illustrate two Dafny verification tasks that exhibit the same verification pattern: a case analysis for each constructor of an inductive data type, with two possible lemma applications. Still, a user has to spell out all the proof details. To abstract over such details, we introduce a *tactic* construct that allows the user to encode the *verification pattern*. The proof details of the lemmas can then be replaced by a single tactic application. This is achieved by extending Dafny with a new ghost method called **tactic**, without any contracts. The following tactic captures the proofs of AsimpConst and Lemma:

```
tactic CasePerm(v: Element)
{ solved{
    var lem1  :| lem1 in lemmas();
    var lem2  :| lem2 in lemmas();
    cases(v){ var vars := variables();
              perm(lem1, vars); perm(lem2, vars);}} }
```

The tactic takes an input v, expected to be a variable of an inductive data type. It then picks two lemma names, lem1 and lem2. Here, all possible combinations of lemma names will be generated as separate branches of the search space. Then for each branch, cases(v) will generate a match statement with a case for each constructor ofv. Within each case the "body" of cases is evaluated. Here, all variables in scope are found (vars). The invocation of perm(lem1,vars) generates all possible permutations of applying lemma lem1 with arguments found in any sub-set of vars in separate branches of the search space. The second application of perm does the same for lem2. The keyword solved{ body } states that the program has to verify when body has been evaluated. This is required here as a tactic may be used to progress a proof without completing it, which is desirable in certain cases. Note that the tactic will stop evaluating when a proof is found, thus the body will not be applied for cases such as Nil above. For that reason, CasePerm(b) will also work for the following lemma found in the first example:

```
lemma BsimpCorrect(b: bexp,  s: state)
 requires Total(s);
 ensures bval(bsimp(b),s) = bval(b,s);{
{ match b
   case Bc(v) ⇒
   case Not(b0) ⇒ BsimpCorrect(b0,s);
   case And(b0, b1) ⇒ BsimpCorrect(b0,s); BsimpCorrect(b1,s);
   case Less(a0, a1) ⇒AsimpCorrect(a0,s); AsimpCorrect(a1,s);}
```

This lemma states a similar property to AsimpConst, with the difference that it is applied to boolean expressions. It works for the Not(b0) case as it will stop evaluating when the case verifies. Thus, in the Not(b0) case the second call to perm will not be applied as BsimpCorrect(b0,s) is sufficient. To apply a tactic the body is replaced by a call to it, illustrated for Lemma:

```
lemma Lemma2(l: List, v: int, val: int)
  ensures SubstL(SubstL(l,v,val),v,val) = SubstL(l,v,val);
{  CasePerm(l); }
```

The tactic language is *metalanguage* for Dafny, where tactic evaluation works at the Dafny level: it takes a Dafny program with tactics and tactic applications, evaluates the tactics and produces a new valid Dafny program, where tactic calls are replaced by Dafny constructs generated by the tactics. The advantages of working on the Dafny level are: (*i*) *iterative development and debugging* is supported as a user can partly develop a tactic, inspect the result and then extend or modify it; (*ii*) *soundness* as users can inspect and validate the result and the tool is independent of its encoding into Boogie; (*iii*) *modularity* as it becomes easier to adapt to new versions of Dafny.

While we can rely on the soundness of Dafny for the actual verification, a program transformation could in principle make changes to both the program and its specification. A tactic should not make such changes to Dafny programs as this is changing what we are attempting to prove:

Definition 1 (Contract Preserving Transformation). *A* contract preserving transformation *is a program transformation that preserves the behaviour and the contract of a method (or lemma or function).*

The evaluation of a tactic call will transform the code by replacing the call with the code the tactic generates. To illustrate, the evaluation of Lemma2 should generate the same body as Lemma (or similar verifiable code that fits the pattern encoded by the tactic). As a lemma is a ghost construct, and the contract is unchanged, the transformation is contract preserving.

By reusing Dafny constructs, users can develop schematic and intuitive tactics, comparable to *declarative* or *schematic* tactics found in some modern ITP tactic languages, e.g. [4]. To illustrate, a proof by ‘mathematical induction’ over a variable n, where we assume existence of tactic base_tac(), which handles the base case (when n is 0), and tactic step_tac(), which is used for the step case, can be written as follows:

```
tactic InductTac(n: Element)
{ if n = 0 { base_tac(); }
  else { var curr := current();  curr(n−1); step_tac(); } }
```

Variable curr will point to the “current” method or lemma in which the tactic was called from (and the generated code will be added). curr(n-1) is therefore an application of the induction hypothesis before the step case is handled.

The **if** statement of InductTac is at the *object level*, i.e. it will be part of the generated Dafny code after applying the tactic. Statements such as **if** (and **while**) are also used at the *meta-level*, i.e. used by Tacny and will not be generated. These levels are distinguished by whether the tactic evaluator can evaluate the condition. If it cannot be evaluated then it is an object-level feature. In this case, we do not know the value of n hence we cannot resolve whether n = 0 is true or false.

Such schematic tactics provide a very elegant way of composing tactics, a well known problem for ITP tactics [3]. cases(v){ body } illustrated another example of composition where { body } is used to separate tactics applied within each case, from tactics that should follow the match statement.

Both CasePerm and InductTac have a parameter of type *Element*. This is a Tacny-specific type denoting the name of an element of the Dafny program text, such as a variable, method name or lemma name. For CasePerm this is assumed to be a variable of an inductively defined datatype while for InductTac it should be a variable that is a natural number. To simplify, we are using a single type to refer to this; *type safety* is handled by Dafny which will fail when we try to verify a wrongly typed program. This is another example of *modularity*, albeit at the expense of efficiency in this case. Note that for InductTac we cannot use **nat** as this refers to a number and not a variable of **nat** type.

3 The Tacny Language

The Tacny language is designed to be as familiar for Dafny users as possible. A tactic definition is a type of Dafny ghost method, with the following syntax:

tactic *Id(Params)*{ *TStmts* }

where a parameter *Param* has the syntax *Id* : *Type*. A type here is any Dafny type [26], with two additional types: *Element*, already discussed; and *Term*, the term representation of a Dafny expression. A Tacny statement *TStmt* is:

TStmt := *Atom* | *Id(TExprs);* | *var Id* := *TExpr;* | *Id* := *TExpr;*
| **var** *Id* :| *TExpr;* | *Id* :| *TExpr;* | { *TStmts* }
| **if** *TExpr* { *TStmts* } | **if** *TExpr* { *TStmts* } **else**{ *TStmts* }
| **while** *TExpr Invs* { *TStmts* } | *TStmt* || *TStmt;*

With the exception of || and *Atom*, these constructs are part of the Dafny language. However, within a tactic they have different semantics: e.g. a declared variable in a tactic will not appear in the Dafny program resulting from tactic evaluation. *Atom* refers to the *atomic tactics* of the Tacny language. These are the hard-coded building blocks of Tacny, and all tactics are compositions of them. The set of atomic tactics is expected to change and develop, but hopefully converge. So far, we have identified the following atomic tactics:

Atom := *id();* | *fail ();* | **invariant** *TExpr;* | **decreases** *TExpr;*
| **assert** *TExpr;* | **fresh var** *Id* := *TExpr;* | **fresh var** *Id* :| *TExpr;*
| *try*{ *TStmts* }*catch*{ *TStmts* } | *cases(Element)*{ *TStmts* }
| *solved*{ *TStmts* } | *perm(Element,***seq***<Element>);*

Dafny's contract and loop (in)variant *Inv* is extended with tactic calls *Id(TExprs)*, with the syntax definition omitted for space reasons. A *TExpr* is an extension of Dafny's expression *Expr*:

$$\mathit{TExpr} := \mathit{Expr} \mid \textsf{current()} \mid \textsf{variables ()} \mid \textsf{lemmas()} \mid \textsf{params()} \mid \cdots$$

Note that to understand the evaluation of Tacny expressions, the full details of Dafny expressions are not required and thus omitted. To evaluate a Tacny expression, a context C, containing relevant details of the program at the point a tactic call was made, and a state s, which holds a map from Tacny-specific variables to values, are given. These may be updated during evaluation. The evaluation of an expression is given by $[\![-]\!]_{\langle C,s\rangle}$, with the following semantics:

$$\begin{array}{rll}
[\![e_1 \; op \; e_2]\!]_{\langle C,s\rangle} := & & \text{when } op \in \{+,-,*,/\} \\
 & [\![e_1]\!]_{\langle C,s\rangle} \; op \; [\![e_2]\!]_{\langle C,s\rangle} & \text{and } [\![e_1]\!]_{\langle C,s\rangle} \text{ and } [\![e_2]\!]_{\langle C,s\rangle} \text{ are numbers.} \\
[\![!e]\!]_{\langle C,s\rangle} & := \neg[\![e]\!]_{\langle C,s\rangle} & \text{when } [\![e]\!]_{\langle C,s\rangle} \neq \bot \\
[\![|e|]\!]_{\langle C,s\rangle} & := \mathit{length}([\![e]\!]_{\langle C,s\rangle}) & \text{when } [\![e]\!]_{\langle C,s\rangle} \neq \bot \\
[\![e]\!]_{\langle C,s\rangle} & := \mathit{true} & \text{when } \mathit{tautology}(e) \\
[\![e]\!]_{\langle C,s\rangle} & := \mathit{false} & \text{when } \mathit{tautology}(!e) \\
[\![n]\!]_{\langle C,s\rangle} & := s(n) & \text{when } n \in \mathit{dom}(s) \\
[\![v]\!]_{\langle C,s\rangle} & := v & \text{when } v \text{ is a value} \\
[\![f\,()]\!]_{\langle C,s\rangle} & := C.f & \text{when } f \in \{\textsf{current}, \textsf{variables}, \textsf{lemmas}, \textsf{params}\} \\
[\![e]\!]_{\langle C,s\rangle} & := \bot & \text{otherwise}
\end{array}$$

tautology is a simple tautology checker for proposition logic with (in)-equality; *length* returns the length of a sequence, and a dot-notation is used to project values from the context. If an expression cannot be evaluated, then $\bot$ is returned. In that case, the expression is treated as *object* level. E.g. if we cannot evaluate the condition of an **if**-statement, then an **if**-statement will be generated, which enables us to write declarative/schematic tactics in Tacny. In such cases, Tacny-level variables need to be instantiated (using the state s) and Tacny-level expressions (current, variables, lemmas, params) unfolded (using the context C). This is achieved by $[-]_{\langle C,s\rangle}$, which we do not provide further details of. In the semantics we often try $[\![e]\!]_{\langle C,s\rangle}$ first, applying $[-]_{\langle C,s\rangle}$ if it fails (i.e. returns $\bot$):

$$[\![e]\!]^?_{\langle C,s\rangle} := \begin{cases} [\![e]\!]_{\langle C,s\rangle} & \text{when } [\![e]\!]_{\langle C,s\rangle} \neq \bot \\ [e]_{\langle C,s\rangle} & \text{otherwise} \end{cases}$$

We give Plotkin-style big-step operational semantics [38], using a nondeterministic relation $\longrightarrow$. $\langle C, s, c, \mathit{stmt}\rangle \longrightarrow \langle C', s', c'\rangle$ should be read as: given context C, state s, generated Dafny code c and Tacny statement *stmt*, evaluation will produce a new context C', state s' and Dafny code c'. The inference rules are given in Figs. 1 and 2. For space reasons we do not provide a full set of rules[7]. We focus on the interesting cases, omitting rules for generating context and evaluating methods and lemmas, where tactics are called from, which is only briefly discussed.

[7] For full details we refer to the 'Tacny system working document' [21].

$$\frac{\langle C, s, c, \mathit{stmt}_1\rangle \longrightarrow \langle C', s', c'\rangle}{\langle C, s, c, \mathit{stmt}_1 \mid\mid \mathit{stmt}_2\rangle \longrightarrow \langle C', s', c'\rangle} \qquad \frac{\langle C, s, c, \mathit{stmt}_2\rangle \longrightarrow \langle C', s', c'\rangle}{\langle C, s, c, \mathit{stmt}_1 \mid\mid \mathit{stmt}_2\rangle \longrightarrow \langle C', s', c'\rangle}$$

$$\frac{\langle C, s, c, \mathit{stmt}\rangle \longrightarrow \langle C'', s'', c''\rangle \qquad C''.\mathit{vcs} \neq \{\} \qquad \langle C'', s'', c'', \mathit{stmts}\rangle \longrightarrow \langle C', s', c'\rangle}{\langle C, s, c, \mathit{stmt}\ \mathit{stmts}\rangle \longrightarrow \langle C', s', c'\rangle}$$

$$\frac{\langle C, s, c, \mathit{stmt}\rangle \longrightarrow \langle C', s', c'\rangle \qquad C'.\mathit{vcs} = \{\}}{\langle C, s, c, \mathit{stmt}\ \mathit{stmts}\rangle \longrightarrow \langle C', s', c'\rangle}$$

$$\frac{\langle C, s, c, B\rangle \longrightarrow \langle C, s', c'\rangle}{\langle C, s, c, \{B\}\rangle \longrightarrow \langle C[\mathit{vcs} := C'.\mathit{vcs}], \mathit{dom}(s) \lhd s', c'\rangle}$$

$$\frac{\mathsf{x} \notin \mathit{dom}(s) \cup C.v \cup C.p \qquad [\![e]\!]^?_{\langle C,s\rangle} = v}{\langle C, s, c, \textsf{var}\ x := e;\rangle \longrightarrow \langle C, s[x := v], c\rangle} \qquad \frac{\mathsf{x} \in \mathit{dom}(s) \qquad [\![e]\!]^?_{\langle C,s\rangle} = v}{\langle C, s, c, x := e;\rangle \longrightarrow \langle C, s[x := v], c\rangle}$$

$$\frac{\mathsf{x} \notin \mathit{dom}(s) \cup C.v \cup C.p \qquad [\![P(n)]\!]_{\langle C,s\rangle} = \mathit{true}}{\langle C, s, c, \textsf{var}\ x :\mid P(x);\rangle \longrightarrow \langle C, s[x := n], c\rangle} \qquad \frac{C' = C[\mathit{vcs} := \mathit{verify}(C.M[c])]}{\langle C, s, c, \mathsf{id}();\rangle \longrightarrow \langle C', s, c\rangle}$$

$$\frac{\mathsf{x} \in \mathit{dom}(s) \qquad [\![P(n)]\!]_{\langle C,s\rangle} = \mathit{true}}{\langle C, s, c, x :\mid P(x);\rangle \longrightarrow \langle C, s[x := n], c\rangle} \qquad \frac{[\![b]\!]_{\langle C,s\rangle} = \mathit{false}}{\langle C, s, c, \textsf{while}\ b\ I\{B\}\rangle \longrightarrow \langle C, s, c\rangle}$$

$$\frac{[\![b]\!]_{\langle C,s\rangle} = \mathit{true} \qquad \langle C, s, c, \mathsf{id}();B\rangle \longrightarrow \langle C', s', c'\rangle \qquad C'.\mathit{vcs} = \{\}}{\langle C, s, c, \textsf{while}\ b\ I\{B\}\rangle \longrightarrow \langle C', \mathit{dom}(s) \lhd s', c'\rangle}$$

$$\frac{\begin{array}{c}[\![b]\!]_{\langle C,s\rangle} = \mathit{true} \qquad \langle C, s, c, \mathsf{id}();B\rangle \longrightarrow \langle C'', s'', c''\rangle \\ C''.\mathit{vcs} \neq \{\} \qquad \langle C'', s'', c'', \textsf{while}\ b\ I\{B\}\rangle \longrightarrow \langle C', s', c'\rangle\end{array}}{\langle C, s, c, \textsf{while}\ b\ I\{B\}\rangle \longrightarrow \langle C', \mathit{dom}(s) \lhd s', c'\rangle}$$

$$\frac{\begin{array}{c}\langle C[\mathit{mode} := \mathit{annot}], s, \epsilon, I\rangle \longrightarrow \langle C'', s'', I'\rangle \qquad M = \lambda x.\ C.M[\textsf{while}\ [b]_{\langle C,s\rangle}\ I'\{x\}] \\ [\![b]\!]_{\langle C,s\rangle} = \bot \qquad \langle C[M := M], s, \epsilon, B\rangle \longrightarrow \langle C', s', B'\rangle\end{array}}{\langle C, s, c, \textsf{while}\ b\ I\{B\}\rangle \longrightarrow \langle C[\mathit{vcs} := C'.\mathit{vcs}], s, c \cdot \textsf{while}\ [b]_{\langle C,s\rangle}\ I'\{B'\}\rangle}$$

Fig. 1. Operational semantics for Tacny statements [1/2]

To evaluate $||$, as shown in Fig. 1, either the statement on the left or on the right is evaluated. Sequential composition depends on whether the program verifies after the first statement is completed. If the set of verification conditions *vcs* is empty, the evaluation stops; if not, it continues to the next statement. When a block is evaluated then only changes to the given state s are kept and the context is only updated with the VCs[8]. A declaration or assignment will update s, and the expression e is evaluated by $[\![e]\!]^?_{\langle C,s\rangle}$, meaning it may not evaluate fully. The projections $C.v$ and $C.p$ are the set of declared variables and parameters, respectively, of the method the tactic was called from. To evaluate the $:\mid$ operator

[8] $S \lhd R$ restricts the domain of relation/map R to the set S.

$$\frac{C.tac[t] = t(a_1, \cdots, a_n)\{B\} \quad \langle C, [a_1, \cdots, a_n := [\![e_1]\!]^?_{\langle C,s\rangle}, \cdots, [\![e_n]\!]^?_{\langle C,s\rangle}], c, B\rangle \longrightarrow \langle C', s', c'\rangle}{\langle C, s, c, t(e_1, \cdots, e_n)\rangle \longrightarrow \langle C, s, c'\rangle}$$

$$\frac{\langle C, s, c, B\rangle \longrightarrow \langle C', s', c'\rangle \qquad C'.vcs = \{\}}{\langle C, s, c, \textsf{solved}\{B\}\rangle \longrightarrow \langle C[vcs := C'.vcs], dom(s) \lhd s', c'\rangle}$$

$$\frac{mode = annot}{\langle C, s, c, \textbf{decreases}\ e;\rangle \longrightarrow \langle C, s, c \cdot \textbf{decreases}\ [e]_{\langle C,s\rangle};\rangle}$$

$$\frac{mode = code \qquad C' = C[vcs := verify(C.M[c \cdot \textbf{assert}\ [e]_{\langle C,s\rangle};])]}{\langle C, s, c, \textbf{assert}\ e;\rangle \longrightarrow \langle C', s, c \cdot \textbf{assert}\ [e]_{\langle C,s\rangle};\rangle}$$

$$\frac{\begin{array}{c} a_1 \in as \quad \cdots \quad a_n \in xs \qquad m(a_1, \cdots, a_n)\_\{\_\} \in C.m \cup C.l \cup C.f \\ ghost(m) \qquad C' = C[vcs := verify(C.M[c \cdot m(a_1, \cdots, a_n);])] \qquad C.mode = code \end{array}}{\langle C, s, c, \textsf{perm}(m, as);\rangle \longrightarrow \langle C', s, c \cdot m(a_1, \cdots, a_n);\rangle}$$

$$\frac{\begin{array}{c} x : T \qquad T = C_1(x_{10}, \cdots, x_{1i}) \mid \cdots \mid C_n(x_{n0}, \cdots, x_{nj}) \qquad x_{ij} \notin C.v \cup C.p \\ M_1 = \lambda y.\ C.M[\textsf{match}\ x\ \textsf{case}\ C_1(x_{10}, \cdots, x_{1i}) \Rightarrow y \quad \textsf{case}\ \_ \Rightarrow \textsf{assume false;}] \ \cdots \\ M_n = \lambda y.\ C.M[\textsf{match}\ x\ \textsf{case}\ C_n(x_{n0}, \cdots, x_{nj}) \Rightarrow y \quad \textsf{case}\ \_ \Rightarrow \textsf{assume false;}] \\ \langle C[v := C.v \cup \{x_{10}, \cdots, x_{1i}\}, M = M_1], s, \epsilon, \textsf{id}();B\rangle \longrightarrow \langle C'_1, s_1, c_1\rangle \ \cdots \\ \langle C[v := C.v \cup \{x_{n0}, \cdots, x_{nj}\}, M = M_n], s, \epsilon, \textsf{id}();B\rangle \longrightarrow \langle C'_n, s_n, c_n\rangle \\ m_1 = \textsf{case}\ C_1(x_{10}, \cdots, x_{1i}) \Rightarrow c_1 \quad \cdots \quad m_n = \textsf{case}\ C_n(x_{n0}, \cdots, x_{nj}) \Rightarrow c_n \\ C' = C[vcs := verify(C.M[c \cdot \textsf{match}\ x\ m_1\ \cdots\ m_n])] \qquad C.mode = code \end{array}}{\langle C, s, c, \textsf{cases}(x)\{\ B\ \}\rangle \longrightarrow \langle C', s, c \cdot \textsf{match}\ x\ m_1\ \cdots\ m_n\rangle}$$

Fig. 2. Operational semantics for Tacny statements [2/2]

we find a value where the property holds. This has to be enumerable, and P has to have the syntactic form x in X, possibly followed by further constraints on x. X must be a collection that can be derived from s and/or C.

The identity atomic tactic id() only changes the context, by attempting to *verify* the program using Dafny. In order to apply Dafny to verify it, the code surrounding a tactic call (or other construct as seen below) must be given. This is provided by $C.M$ in the context, with a "hole" $[-]$ where the code generated by the tactic can be "plugged in", as illustrated by $C.M[c]$ in the identity tactic. *verify* is used to represent a call to Dafny, returning a set of open VCs. If Dafny fails to execute, e.g. due to type errors, then the rule will fail. fail() always fails and is therefore not given an inference rule following a closed world assumption.

We will only discuss the **while** control structure as this is the most interesting and complicated; conditionals (**if** and **if-else**) have comparable semantics but are omitted for space reasons. **while** is captured by 4 inference rules. The first is the trivial case where the condition is *false* and nothing is changed. The second case is when the condition is *true* and the resulting program verifies, in which case the loop is terminated. Note that the body is prefixed by a call to the identity tactic to enforce a call to the verifier before a tactic is applied. The third case is the "step case" where the program does not verify and the loop is recursively applied. The final case is the most interesting one. This is an example of a schematic tactic, where the **while** loop will be generated in the Dafny code. Here, it is not possible to evaluate the condition, meaning $[-]_{\langle C,s\rangle}$ is applied to the generated condition. As the loop annotations (loop invariants and decrease clauses) may have tactic calls, this is first evaluated in an *annot* mode we will return to below. We then evaluate the body. Note that $C.M$ is updated with the loop (using λ notation for the hole) as the verifier has to know about the loop when applied within the body; this change is local to the loop and is discarded afterwards. ϵ denotes empty code, and $\cdot$ concatenates code.

The rule to make a call to another tactic is shown in Fig. 2. The input state of the method only contains the parameters, meaning there is no shared state between tactics. These are evaluated as far as possible. A tactic call within a lemma or method has the same semantics, with c set to ϵ. $C.tac$ maps tactic names to their definition. The solved tactic is similar to a block, but requires that the program verifies on termination of the block; **decreases** statements are only valid in the *annot* mode, i.e. contract or loop annotations. Note that the expression is simplified by $[-]_{\langle C,s\rangle}$ which e.g. allows us to write more generic tactics by including Tacny level variables. Note that the verifier is not applied in the annotations, however it is after an **assert**ion is added. This requires a *code* mode, i.e. generation of Dafny statements.

The perm(m, as) tactic generates all possible ways of applying m with arguments taken from as. Here, m is first found in the context among the methods, lemmas and functions, and checked that it is a ghost construct. The rule allows all possible combinations, while Dafny is used to ensure type checking as part of the verification. The cases tactic has the most involved semantics. The given variable has to of an inductively-defined type and the constructors (with fresh argument names) are created. Each case is evaluated separately, and to control the verifier the other cases are assumed to be false[9].

Most state-of-the-art ITP systems follow the LCF-approach [19] which reduces soundness to a small "trusted kernel" of axioms and inference rules. The following proposition states a similar feature for Tacny without proof:

Proposition 1. $\longrightarrow$ *is a contract preserving transformation if the atomics are contract preserving.*

To increase soundness, our aim is to converge to a small "trusted kernel" of atomic tactics we can show are contract preserving. This is straightforward to show for the atomic tactics discussed here:

Proposition 2. *id(), fail(), perm, cases,* **decreases***,* **assert***, solved are contract preserving.*

[9] The underscore '_' is not valid Dafny syntax in pattern matching but used for brevity.

4 The Tacny System

The Tacny tool provides a proof-of-concept implementation of the semantics. The architecture of this tool is shown in Fig. 3, where the shaded boxes represent Dafny components. The tool accepts a Dafny program extended with tactics (`.tacny`) and the Dafny PARSER has therefore been updated with the grammar discussed in the previous section.

The parsed program is then sent to the INTERPRETER, which is discussed in detail below. It uses the GENERATOR, which removes all tactics and tactic calls from the source program, thus making it a valid Dafny (`.dfy`) program. This is used at the end, to generate "a proof", in terms of a valid Dafny program, and during interpretation. In the latter case, the DAFNY RESOLVER performs *type checking* and prepares the program for translation to BOOGIE, which is conducted by the VERIFIER. As with the PARSER, this is a minor update of the existing Dafny code, with some additional book-keeping. The result from BOOGIE is then sent back to the INTERPRETER.

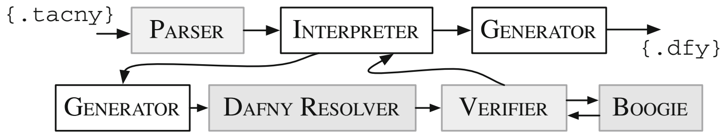

Fig. 3. Tacny tool architecture

```
procedure INTERPRETER($prog$ : Tacny)
    $r \leftarrow$ INITTASK($prog$)
    $r \leftarrow r[pc :=$ NEXTTAC($r$)$]$
    if ENDOFFILE($r.pc$) then
        return $[r]$
    $res, solutions \leftarrow [r], []$
    repeat
        for each $r \in$ MAPS(TAC, $res$) do
            $r \leftarrow r[pc :=$ NEXTTAC($r$)$]$
            if $r.vcs = \{\} \vee$ ENDOFFILE($r.pc$) then
                $solutions \leftarrow solutions + r$
            else $res \leftarrow res + r$
    until res = []
    return solutions

procedure TAC($t$ : Task)
    $res \leftarrow [t]$
    repeat
        for each $r \in$ MAPS(STEP, $res$) do
            if $r.vcs = \{\}$ then return $[r]$
            $res \leftarrow res + r'$
    until ENDOFTAC($t.pc$)
    return res

procedure STEP($t$ : Task)
    $res \leftarrow []$
    for each $r \in$ TACSTEP($t$) do
        $dfy \leftarrow$ GENERATE($r$)
        if $dfy =$ NULL then return []
        $bgy \leftarrow$ BOOGIE($dfy$)
        if PROVEN($bgy$) then
            return $[r[vcs := \{\}]]$
        if SUBGOALS($bgy$) then
            $res \leftarrow res + [r[vcs := bgy]]$
    return res
```

Fig. 4. Tacny interpreter

The main work of Tacny happens in the INTERPRETER, and the algorithm is given in Fig. 4. It first generates an initial task by INITTASK. A task is a record containing a state, context and Tacny program similar to the input of $\longrightarrow$ from Sect. 3. In addition, it contains a program counter. INITTASK will generate the context, and initialise the state to empty. NEXTTAC will then find the next tactic call, or reach the end of the file (ENDOFFILE) if there are no more calls.

In the main loop, the INTERPRETER keeps track of intermediate results *res* and completed *solutions*. It then applies a *breadth-first* search strategy by applying a single tactic application, represented by the undefined TACSTEP procedure, for each element of the result list. This continues until either there are no "open" VCs or there are no more tactic calls. Each tactic evaluation involves a step by step evaluation where after each step a Dafny program is generated. Tacny works on a method-by-method basis, and to focus verification on the current method, the body of all other methods is removed, and all tactics and tactic calls are removed to make it a valid Dafny program. The DAFNY RESOLVER may fail, e.g. if the program does not type check, returning NULL; in that case that particular task is aborted. If not, BOOGIE is applied and the result is returned. If BOOGIE can prove that the program is correct, then the task is completed. This modularity has the advantage that extensions to Dafny and Boogie can be easily integrated: Tacny is a layer on top of Dafny. Currently, some features of Sect. 3 are not supported, such as object level (schematic) loops.

5 Experiments

In Sect. 2 we introduced the CasePerm tactic and showed that it was applicable for multiple examples. Table 1 shows the results from our experiments[10], with results from running CasePerm above the double line (1). **Calls** refers to the number of tactic calls for each file; **TLoC** and **DLoC** are the LoC for the respective Tacny and (analysed) Dafny programs; **Vars** is the highest number of variables in a branch; **#B** is the number of branches; **#DB** is the number of discarded branches before a solution was found; **T** is the running time for Tacny in seconds.

A user must often provide a variant in form of a **decreases** clause to prove that a loop or recursive method terminates. This can often be a trial-and-error process, where the user may have some ideas in hand. Here, we give a generic tactic to generate either a single variable or a subtraction of two variables:

```
tactic VariantGen(){
  solved{ var x :| x in params() + variables();
          decreases x ||
           { var y :|  x in params() + variables() ∧ x ≠ y;
             decreases x−y }; }
}
```

[10] All examples are taken from the Dafny repo [1], with code used available from [2]. Experiments conducted running Windows 8.1 on Intel i7 2.4 GHz with 4GB RAM.

Table 1. Results from executing: CasePerm (1) and VariantGen (2).

Tac	Program	Calls	TLoC	DLoC	Vars	#B	#DB	T(sec)
(1)	`NipkowKlein-chapter3.dfy`	2	186	192	4	1473	506	113
	`Substitution.dfy`	2	52	84	5	21739	4758	190
	`InductionVsCoinduction.dfy`	1	74	70	5	1137	544	30
	`Streams.dfy`	7	221	228	3	62	0	37
	`CoqArt-InsertionSort.dfy`	2	201	198	2	58	0	15
(2)	`Dijkstra.dfy`	1	126	117		19	4	8
	`SchorrWaite.dfy`	1	204	195		34	7	11
	`Prime.dfy`	1	232	224		54	9	20
	`SetIterator.dfy`	1	82	63		48	6	20
	`SimpleInduction.dfy`	1	66	65		11	0	5

The results from applying this to a set of examples can be seen below the double line (2) in Table 1. The number of variables are omitted as they are not relevant.

The fact that a single tactic can be applied to several examples (14 and 5) and, for CasePerm several lemmas within each example, provides evidence for our hypothesis that tactics for Dafny are feasible. In most cases, CasePerm reduces the annotation overhead. For the other cases, the size has not reduced for 2 reasons: (i) the proofs replaced where short; and (ii) although the tactic is the same for all examples, it had to be copied into each example for technical reasons. This is the reason why VariantGen increased the LoC: each variant is a single line, thus replacing it with a tactic call will have the same LoC. For this tactic, we cannot argue reduced annotation overhead, however the manual search task is replaced by a tactic, thus development time should decrease.

For some examples CasePerm has a very high branching factor, in particular when there are 4 or more variables. This is due to the naivety of the perm tactic, and we are working on improvements to this. We believe that this has potential for a very generic tactic if we can extend it and, at the same time, improve on the branching factor. VariantGen is discussed further in Sect. 7.

6 Related Work

An alternative approach to Dafny tactics is the more traditional approach of proving the generated VCs in an interactive prover and developing tactics at this level. This has the following drawbacks: users have to (additionally) learn how to use and develop tactics in the interactive prover[11]; and certain tactics, such as adding an invariant, precede VC generation. Thus, a richer set of tactics can be developed in the program text. Most tactic languages for ITP systems,

[11] It commonly takes at least six months just to become a proficient user of an ITP system (see e.g. [30]), and even longer to have sufficient expertise to develop tactics.

dating back to the seminal LCF system [19], contain a *combinator* language to compose tactics into larger and more powerful ones. For example, the solved tactic is common. As far as possible, we have attempted to use Dafny's constructs to compose tactics to keep them as familiar as possible. Within ITP, there is also a trend towards building tactic languages at the *proof script* level, compared with the *implementation language* of the system. LTac for Coq [14] and EisBach for Isabelle [33] are examples of such languages. Non-trivial tactic compositions are hard to get right for tactic languages due to their procedural nature; see e.g. [18] for a discussion. Inspired by *declarative proof scripts*, with Mizar [36] and Isabelle/Isar [40] probably the most well-known, [4] develops a *declarative tactic language*, where a tactic is given a more schematic description. Such schematics are also supported in Tacny, and provide a more intuitive mechanism for tactic composition. Most of the popular ITPs follow the so-called LCF approach [19], where soundness is ensured by a small "trusted kernel" of axioms and inference rules. The type system ensures that all proof steps go through this kernel. We are following this approach by reducing our correctness property (contract preserving transformation) to the atomic tactics, where all the code is generated. We hope the set of atomics will converge into a small kernel. These resemblences to ITP tactics are the reason that we have adopted the 'tactic' name for our language. It is also considered good practice that each refactoring should only make small changes to the code as this is easier to analyse [17]. [41] applies refactoring to proof scripts to improve existing proofs; however, it is not used to support the proof process of open conjectures, as is the case for Tacny.

We are not familiar with any other work attempting to develop a tactic language at the program text level for program verifiers[12]. Chen [7] describes a simple imperative language that includes verification constructs. This may provide the foundations to encode a tactic language, similar to ours, but it is not clear how this can be done. Moreover, our goal is to work with existing program verifiers. The Aris project [32] uses case-based reasoning to re-use specifications from a large corpus of existing proofs in Spec# [6], and it would be interesting to see if the Tacny language could be used as a target language for the generalisation of such specifications. There has also been work at "lower-levels": Leino [28] has developed an "induction tactic" for Dafny. This is an optimisation of the encoding into Boogie, and requires deep understanding of the underlying Dafny implementation. Moreover, working at this level one has to be very careful not to introduce inconsistencies in the logic. At an even lower-level, a tactic language has been developed for the underlying Z3 prover [13] – again, this requires expertise in SMT solving and Z3.

To *improve readability* of the program text, Dafny supports Dijkstra-style calculational proofs (**calc**) [30], comparable to declaritive proofs in e.g. Mizar or Isabelle/Isar. A considerable amount of work has been done using *static analysis* techniques on the program text in order to *reduce the number of required annotations* by automatically generating these (in particular loop invariants) [16,22,23,39]. Techniques include abstract interpretation [12], constraint-based

[12] Unpublished early ideas for tactics by the first author is available on ArXiV [20].

techniques [10], inductive logic programming [16], and declarative machine learning [31]. Stretching our "ITP analogy", comparing this work to Tacny, is like comparing tactic languages to decision procedures: we are not proposing a new technique to improve automation, but a language in which users can encode patterns so *they* can improve automation. Note that Dafny uses abstract interpretation at the Boogie level [5].

The perm tactic can seen as a limited form of *term synthesis* at the Dafny level. This technique is used in theory exploration tools, such as IsaCosy [25] and HipSpec [8]. The perm tactic can also be seen as a form of a *brute-force* tactic that essentially tries various combinations without an overall proof pattern. The cases tactic introduces 'proof by cases' for inductive data types. This can easily be extended to support structural induction, by adding recursive calls. The CasePerm tactic is a generalisation of this and similar to how one would prove simple inductive lemmas in ITPs: apply induction followed by a powerful tactic. The VariantGen tactic is used to guide the search for a termination measure. There is a considerable amount of work on proving termination (see e.g. [11]), which is beyond the discussion here.

7 Conclusion and Future Work

We have extended the Dafny program verifier by adding support for users to encode reusable verification patterns using a novel *tactic language* in the program text. We have provided formal syntax and semantics for this extension, implemented as the Tacny tool. Our experiments have shown that *it is possible to encode Dafny tactics and reuse them accross verification tasks.* This has been illustrated by two tactics, used to automate 19 lemmas that required interaction. 10 different Dafny programs were used in order to illustrate generality.

We are continously developing, re-engineering and analysing Dafny programs, in order to extract common verification patterns, and use this to develop new (atomic) tactics, which we hope will converge as a result of this work. Based upon ITP kernels we hope that around $15-20$ will be sufficient. We are currently investigating better integration of the proof failure information into the language, e.g. the solved condition of VariantGen could be weakened to 'no termiation VCs'. We are also starting to incorporate (dynamic) contracts to rule out invalid branches earlier. We also plan to extend the language to: allow function definitions at the Tacny level for more readable tactics; allow "a tactic body" as an"argument" for user defined tactics (as is used for the cases atomic tactic); and use of the recently added higher order features in Dafny [26] to allow tactics as arguments, e.g. allow us to write **tactic** Maybe(t : Tactic){id() ‖ t(); }. Through user evaluations, we would like to validate if/when users find the program text more readable when low-level proofs are replaced by tactics.

The perm tactic is a starting point for a generic brute force tactic, and we are working on extending it to support richer parameters (e.g. constructors and operators which themselves have parameters), and support for assignments, control structures and use within calculations [30]. We would also like to create a more generic VariantGen tactic, with better control of the execution: e.g. only use variables that change, and determine in which cases one variant is more likely to work, thus replacing non-deterministic choice with a conditional.

The Tacny tool is only a proof-of-concept and not particularly fast and we have identified multiple improvements. Firstly, some of the tactics have an ad-hoc implementation and we plan to refactor the code into a more generic framework, where it is easy to extend it with new atomic tactics and expressions and the ability to explore different search strategies. Ideally, tactics should be able to tailor their search strategy. We also plan to explore the use of lazy lists and lazy evaluation to improve memory consumption, and data parallelisation to improve speed. We would also like to see how calls to Dafny/Boogie can be reduced, possibly adding such control to the user via e.g. a form of atomic statement.

Longer term we would like to have a closer integration with Dafny and added support for Tacny in the Visual Studio Dafny IDE. We would also like to investigate how general the approach is by exploring tactics for other program verifiers. "Dafny style proof" has been shown to be feasible in Spark 2014 [15], and we would like to try to develop tactics for Spark 2014 and other program verifiers.

References

1. Dafny. research.microsoft.com/dafny
2. The Tacny projectd: TACAS 2016 information. https://sites.google.com/site/tacnyproject/tacas-2016. Accessed 16 October 2015
3. Asperti, A., Ricciotti, W., Sacerdoti, C., Tassi, C.: A new type for tactics. In: PLMMS 2009, pp. 229–232 (2009)
4. Autexier, S., Dietrich, D.: A tactic language for declarative proofs. In: Kaufmann, M., Paulson, L.C. (eds.) ITP 2010. LNCS, vol. 6172, pp. 99–114. Springer, Heidelberg (2010)
5. Barnett, M., Chang, B.-Y.E., DeLine, R., Jacobs, B., M. Leino, K.R.: Boogie: a modular reusable verifier for object-oriented programs. In: Boer, F.S., Bonsangue, M.M., Graf, S., Roever, W.-P. (eds.) FMCO 2005. LNCS, vol. 4111, pp. 364–387. Springer, Heidelberg (2006)
6. Barnett, M., M. Leino, K.R., Schulte, W.: The Spec# programming system: an overview. In: Barthe, G., Burdy, L., Huisman, M., Lanet, J.-L., Muntean, T. (eds.) CASSIS 2004. LNCS, vol. 3362, pp. 49–69. Springer, Heidelberg (2005)
7. Chen, Y.: Programmable verifiers in imperative programming. In: Qin, S. (ed.) UTP 2010. LNCS, vol. 6445, pp. 172–187. Springer, Heidelberg (2010)
8. Claessen, K., Johansson, M., Rosén, D., Smallbone, N.: Automating inductive proofs using theory exploration. In: Bonacina, M.P. (ed.) CADE 2013. LNCS, vol. 7898, pp. 392–406. Springer, Heidelberg (2013)
9. Cohen, E., Dahlweid, M., Hillebrand, M., Leinenbach, D., Moskal, M., Santen, T., Schulte, W., Tobies, S.: VCC: a practical system for verifying concurrent C. In: Berghofer, S., Nipkow, T., Urban, C., Wenzel, M. (eds.) TPHOLs 2009. LNCS, vol. 5674, pp. 23–42. Springer, Heidelberg (2009)
10. Colón, M.A., Sankaranarayanan, S., Sipma, H.B.: Linear invariant generation using non-linear constraint solving. In: Hunt Jr., W.A., Somenzi, F. (eds.) CAV 2003. LNCS, vol. 2725, pp. 420–432. Springer, Heidelberg (2003)

11. Cook, B., Podelski, A., Rybalchenko, A.: Proving program termination. Commun. ACM **54**(5), 88–98 (2011)
12. Cousot, P., Halbwachs, N.: Automatic discovery of linear restraints among variables of a program. In: Proceedings of the 5th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages, pp. 84–96. ACM (1978)
13. de Moura, L., Passmore, G.O.: The strategy challenge in SMT solving. In: Bonacina, M.P., Stickel, M.E. (eds.) Automated Reasoning and Mathematics. LNCS, vol. 7788, pp. 15–44. Springer, Heidelberg (2013)
14. Delahaye, D.: A tactic language for the system Coq. In: Parigot, M., Voronkov, A. (eds.) LPAR 2000. LNCS (LNAI), vol. 1955, pp. 85–95. Springer, Heidelberg (2000)
15. Dross, C.: Manual Proof with Ghost Code in SPARK (2014). http://www.spark-2014.org/entries/detail/manual-proof-in-spark-2014. Accessed 01 October 2015
16. Ernst, M.D., Perkins, J.H., Guo, P.J., McCamant, S., Pacheco, C., Tschantz, A.S., Xiao, C.: The Daikon system for dynamic detection of likely invariants. Sci. Comput. Program. **69**(1), 35–45 (2007)
17. Fowler, M.: Refactoring: Improving the Design of Existing Code. Addison-Wesley, Menlo Park (1999)
18. Giero, M., Wiedijk, F.: MMode, a Mizar Mode for the proof assistant Coq. Technical report, 07 January 2004
19. Gordon, M.J., Milner, R., Wadsworth, C.P.: Edinburgh LCF. Springer, Heidelberg (1979)
20. Grov, G.: Some Ideas for Program Verifier Tactics. arxiv:1406.2824
21. Grov, G., Tumas, V.: The Tacny system (working document). Version generated, 16 October 2015. Available from [2]
22. Gupta, A., Rybalchenko, A.: InvGen: an efficient invariant generator. In: Bouajjani, A., Maler, O. (eds.) CAV 2009. LNCS, vol. 5643, pp. 634–640. Springer, Heidelberg (2009)
23. Hoder, K., Kovács, L., Voronkov, A.: Invariant generation in vampire. In: Abdulla, P.A., Leino, K.R.M. (eds.) TACAS 2011. LNCS, vol. 6605, pp. 60–64. Springer, Heidelberg (2011)
24. Jacobs, B., Smans, J., Piessens, F.: A quick tour of the verifast program verifier. In: Ueda, K. (ed.) APLAS 2010. LNCS, vol. 6461, pp. 304–311. Springer, Heidelberg (2010)
25. Johansson, M., Dixon, L., Bundy, A.: Conjecture synthesis for inductive theories. J. Autom. Reasoning **47**(3), 251–289 (2011)
26. Leino, K.R.M.: Types in Dafny. http://research.microsoft.com/en-us/um/people/leino/papers/krml243.html. Manuscript KRML 243, 27 February 2015
27. Leino, K.R.M.: Dafny: an automatic program verifier for functional correctness. In: Clarke, E.M., Voronkov, A. (eds.) LPAR-16 2010. LNCS, vol. 6355, pp. 348–370. Springer, Heidelberg (2010)
28. Leino, K.R.M.: Automating induction with an SMT solver. In: Kuncak, V., Rybalchenko, A. (eds.) VMCAI 2012. LNCS, vol. 7148, pp. 315–331. Springer, Heidelberg (2012)
29. Leino, K.R.M., Moskal, M.: Co-induction simply. In: Jones, C., Pihlajasaari, P., Sun, J. (eds.) FM 2014. LNCS, vol. 8442, pp. 382–398. Springer, Heidelberg (2014)
30. Leino, K.R.M., Polikarpova, N.: Verified calculations. In: Cohen, E., Rybalchenko, A. (eds.) VSTTE 2013. LNCS, vol. 8164, pp. 170–190. Springer, Heidelberg (2014)
31. Llano, M.T., Ireland, A., Pease, A.: Discovery of invariants through automated theory formation. FAoC **26**, 203–249 (2014)

32. Pitu, M., Grijincu, D., Li, P., Saleem, A., Monahan, R., O'Donoghue, D.P.: Aris : Analogical reasoning for reuse of implementation & specification. In: AI4FM 2013 (2013)
33. Matichuk, D., Wenzel, M., Murray, T.: An Isabelle proof method language. In: Klein, G., Gamboa, R. (eds.) ITP 2014. LNCS, vol. 8558, pp. 390–405. Springer, Heidelberg (2014)
34. McCormick, J.W., Chapin, P.C.: Building High Integrity Applications with SPARK. Cambridge University Press, Cambridge (2015)
35. de Moura, L., Bjørner, N.S.: Z3: an efficient SMT solver. In: Ramakrishnan, C.R., Rehof, J. (eds.) TACAS 2008. LNCS, vol. 4963, pp. 337–340. Springer, Heidelberg (2008)
36. Naumowicz, A., Korniłowicz, A.: A brief overview of MIZAR. In: Berghofer, S., Nipkow, T., Urban, C., Wenzel, M. (eds.) TPHOLs 2009. LNCS, vol. 5674, pp. 67–72. Springer, Heidelberg (2009)
37. Penninckx, W., Mühlberg, J.T., Smans, J., Jacobs, B., Piessens, F.: Sound formal verification of linux's USB BP keyboard driver. In: Goodloe, A.E., Person, S. (eds.) NFM 2012. LNCS, vol. 7226, pp. 210–215. Springer, Heidelberg (2012)
38. Plotkin, G.D.: The origins of structural operational semantics. J. Logic Algebraic Program. **60–61**, 3–15 (2004)
39. Srivastava, S., Gulwani, S.: Program verification using templates over predicate abstraction. In: ACM Sigplan Notices, vol. 44, pp. 223–234. ACM (2009)
40. Wenzel, M.: Structured induction proofs in Isabelle/Isar. In: Borwein, J.M., Farmer, W.M. (eds.) MKM 2006. LNCS (LNAI), vol. 4108, pp. 17–30. Springer, Heidelberg (2006)
41. Whiteside, I., Aspinall, D., Dixon, L., Grov, G.: Towards formal proof script refactoring. In: Farmer, W.M., Urban, J., Rabe, F., Davenport, J.H. (eds.) MKM 2011 and Calculemus 2011. LNCS, vol. 6824, pp. 260–275. Springer, Heidelberg (2011)