Tactics of Refinement

Marcel Oliveira and Ana Cavalcanti
Centro de Informática/UFPE
PO Box 7851 50740-540 Recife PE Brazil
{mvmo,alcc}@cin.ufpe.br

Abstract

The refinement calculus is a modern technique of formal program development. Its application, however, may lead to long and repetitive developments. In this paper we present a language to write refinement tactics, and present examples of useful tactics. They encompass the application of several refinement laws, but can be used as a single transformation rule. Using tactics is not a novel idea, but apparently, in the context of refinement the only existing work uses Prolog as a tactic language. Our language does not depend of any programming language or tool. Also, we are not aware of any presentation of refinement strategies written in the form of tactics as we present here.

Keywords: Formal methods, program development, refinement calculus, tactic language.

1 Introduction

Morgan's refinement calculus [10] is a successful technique to develop and implement software in a precise, complete, and consistent way. From a formal specification we produce a program which correctly implements the specification by repeatedly applying transformation rules, which are called refinement laws. Using the refinement calculus, however, can be a hard task, as program developments may prove to be long and repetitive.

Frequently used strategies of development are reflected in sequences of law applications that are over and over applied in different developments or even in different points of a single development. A lot is to be gained from identifying these tactics of development, documenting them, and using them in program developments as a single transformation rule.

In this paper we present RTL (Refinement Tactic Language), a language for the definition of refinement tactics based on Angel [9]. This is a general tactic language that is not tailored to any particular proof tool. It makes no assumption about the form of proof goals and the rules that are applied to them. Moreover, the semantics of Angel is well-defined and it has an associated algebraic theory that allows the proof of properties of tactics.

The proof of theorems is usually based on inference rules that involve only terms of the logic. Therefore, Angel assumes that rules transform proof goals into (a sequence of) proof goals. For refinement tactics we need a language that takes into account the fact that refinement laws transform programs into programs, but generate proof obligations. These are not affected by later applications of refinement laws, but the result of applying a sequence of laws (tactic) is a program and the list of all proof obligations generated by the individual law applications.

The constructs of RTL are similar to those of Angel, but are adapted to deal with refinement laws and programs. Moreover, RTL provides structural combinators that are suitable to apply

refinement laws to components of programs. Using RTL, we define refinement tactics that embody commom development and programming strategies [10, 8, 2].

The use of tactics to guide proofs is not a novel idea [9]. Many proof tools provide tactic languages to write programs that help in the construction of proofs [6, 11]. As far as we know, however, refinement tactics have only been considered in [7], where Prolog is used as a tactic language.

In Section 2 we give an overview of the refinement calculus. Section 3 introduces RTL. In Section 4 we present some tactics and examples of their applications. Finally in Section 5 we discuss related and future works.

2 Refinement Calculus

The refinement calculus is based on a unified language of specification, design and implementation. There is no difference between specification and programs. Developing programs consists of applying refinement laws to a specification repeatedly until an adequate program is obtained. Some of these laws are listed in Appendix A.

A specification has the form $w : [pre, post]$ where w, the frame, lists the variables whose values may change, *pre* is the precondition, and *post* is the postcondition. The language used to define the precondition and the postcondition is the predicate calculus. The execution of a specification statement in a state that satisfies the precondition changes the variables listed in the frame so that the final state satisfies the postcondition. If the initial state does not satisfy the precondition the result cannot be predicted. A precondition *true* can be omitted.

In the postcondition 0-subscripted variables can be used to represent the initial value of the corresponding variable. As an example of its use we have $x : [x = x_0 + 1]$. After the execution of this program the variable x has its value in the initial state incremented by one.

Besides the specification statement, the language of the refinement calculus includes all the constructors of Dijkstra's language [5]. There are also block constructs to declare local variables, logical constants, and procedures. Variable blocks have the form $[\![\mathbf{var}\ x : T \bullet p]\!]$, where x is a variable name of type T whose scope is restricted to p. Similarly, logical constants c are declared in blocks of the form $[\![\mathbf{con}\ c \bullet p]\!]$.

Procedure blocks are very simple: they begin with the keyword **proc**, the name of the procedure and a program fragment, called the body of the procedure. This body can have arguments declarations, if the procedure has arguments. Then, after a $\bullet$ the main program is declared. The general form is $[\![\mathbf{proc}\ name = body \bullet prog]\!]$. The arguments can be passed by value, using the keyword **val**, by result, keyword **res**, or by value-result, using the keyword **val_res**. An example of a parameterized procedure is $[\![\mathbf{proc}\ inc = (\mathbf{val\_res}\ n : \mathbb{N} \bullet n := n + 1) \bullet inc(n)]\!]$. The body of the procedure in this case is a parameterized command [1].

Variant blocks are used to develop recursive procedures. Besides declaring the procedure and the main program, a variant block declares a variant e named v used to develop a recursive implementation for the procedure. The general form is $[\![\mathbf{proc}\ name = body\ \mathbf{variant}\ v\ \mathbf{is}\ e \bullet prog]\!]$.

3 RTL

The simplest form of a tactic is a simple law application: **law** *Name*(*Arguments*). When applying a law to a program, there are two possible outcomes: if the law is applicable to the program the law is actually applied and changes the program, possibly generating proof obligations; if the law is not applicable to the program, the application of the law fails.

Sometimes it is very useful to apply a previously defined tactic in another tactic. In RTL we can do this using the construct **tactic** *tacticName*(*tacticArguments*). The behavior of this tactic is similar to that of a **law** tactic, except that it applies a whole tactic and not a single law. The distinction is useful for documentation purposes.

Two special tactics are introduced to help us to define some tactics; they are **skip** and **fail**. The first one always succeeds, does not change the program, and also does not generate proof obligations. The second one always fails.

3.1 Tacticals

In RTL tactics can be combined in sequence. The sequential composition of two tactics is written t_1 ; t_2. This tactics first applies t_1 to the program and then applies t_2 to the outcome of the application of t_1. If either t_1 or t_2 fails this tactic fails. The proof obligations generated by the application of this tactic are those resulting from the application of t_1 and t_2.

For example, if we apply the tactic **law strPost**$(x = 10)$; **law weakPre**$(true)$ to the program $x : [x < 10, x >= 10]$, first we have the application of **strPost**$(x = 10)$. This is the strengthen postcondition refinement law, which takes a new postcondition as argument. The result is $x : [x < 10, x = 10]$ and the proof obligation is $x = 10 \Rightarrow x \geq 10$. Then, **weakPre**$(true)$, the weaken precondition law with parameter $true$, is applied to $x : [x < 10, x = 10]$, and we get $x : [x = 10]$ with proof obligations $x = 10 \Rightarrow x \geq 10$ and $x < 10 \Rightarrow$ true.

Tactics can also be combined in alternation, which is written $t_1 \mid t_2$. This tactic applies t_1 to the program. If the application of t_1 succeeds, then this tactic succeeds, else this tactic applies t_2 to the program. If the application of t_2 succeeds then this tactic succeeds, else the whole tactic fails. When a tactic presents such a choice of next steps, the one that leads to success, if any, is chosen. This is implemented using backtracking.

Consider, by way of illustration, applying to the specification statement $x : [x < 10, x \geq 10]$ the tactic (**law assig**$(< x >, < 10 >) \mid$ **law strPost**$(x = 10)$); **law weakPre**$(true)$. First, this tactic applies **assig**$(< x >, < 10 >)$, the assignment introduction law, to the given program. This application succeeds and results in the assignment $x := 10$ with the proof obligation $x < 10 \Rightarrow 10 = 10$. Then, the tactic tries to apply **weakPre**$(true)$ to $x := 10$. This application fails because **weakPre** only applies to specification statements. Backtracking occurs to consider the second branch of the alternation: **strPost**$(x = 10)$. This application succeeds and we get $x : [x < 10, x = 10]$ with the proof obligation $x = 10 \Rightarrow x \geq 10$. Finally, the tactic tries to apply **law weakPre**$(true)$ to $x : [x < 10, x = 10]$. This application succeeds and we get the program $x : [x = 10]$ with proof obligations $x = 10 \Rightarrow x \geq 10$ and $x < 10 \Rightarrow$ true. If this last law application had failed then the whole tactic would have failed as well.

This kind of behavior may lead to problems of inefficient searches. So, RTL includes the cut operator (!). The tactic $!\,t$ behaves just like t: it returns the first successful tactic application;

if a subsequent tactic application fails, however, then the whole tactic fails. For example, the application of $!(\mathbf{law\ assig}(< x >, < 10 >) \mid \mathbf{law\ strPost}(x = 10))$; $\mathbf{law\ weakPre}(true)$ to the program $x : [x < 10, x \geq 10]$ fails since the application of **weakPre** to $x := 10$ fails.

We also have the recursion operator μ. As an example we have the tactic *exhaust*. When applied to a tactic t, this tactic applies t as many times as possible, terminating with success when the application of t fails. Its definition is $exhaust\ t = (\mu\ Y \bullet (t;\ Y \mid \mathbf{skip}))$.

Sometimes the recursive application of a tactic generates a situation where it neither fails nor succeeds, but keeps running indefinitely. To reason about this kind of problem, it is necessary to introduce a tactic which presents this behavior, this is **abort**.

When we want to define the programs to which the tactic can be applied we use the tactic **applies to** p **do** *tactic*, which introduces a meta-program p that characterizes the program to which this tactic is applicable. The meta-variables used in p can be used in *tactic*. Meta-programs are programs written in a general format. As an example we have the meta-program $w : [pre, post]$ which has as its meta-variables $w, pre, post$. By way of illustration, we consider **applies to** $w : [pre_1 \wedge pre_2, post]$ **do law weakPre**(pre_1); **law strPost**$(post \wedge pre_2)$, which applies only to specifications of the form $w : [pre_1 \wedge pre_2, post]$.

3.2 Structural Combinators

In some cases the program has subprograms and we want to apply tactics to each of them. This is made using *structural combinators*. For example the tactic t_1 [;] t_2 applies to programs of the form $p_1;\ p_2$. It returns the sequential composition of the programs obtained by applying t_1 to p_1 and t_2 to p_2. The proof obligations generated by the application of this tactic are those generated by the application of t_1 to p_1 and of t_2 to p_2.

In the case of an alternation, we use the combinator [**if**] $t_1, ..., t_n$ [**fi**]. When applied to an alternation $\mathbf{if}\ g_1 \rightarrow p_1\ []\ ...\ []\ g_n \rightarrow p_n\ \mathbf{fi}$, it applies each of the tactics to the corresponding program. For example, if we apply to $\mathbf{if}\ x \geq 0 \rightarrow x : [x \geq 0]\ []\ x < 0 \rightarrow x : [x < 0]\ \mathbf{fi}$ the tactic [**if**] $< \mathbf{law\ assig}(< x >, < 1 >), \mathbf{law\ assig}(< x >, < -1 >) >$ [**fi**], we get the program $\mathbf{if}\ x \geq 0 \rightarrow x := 1\ []\ x < 0 \rightarrow x := -1\ \mathbf{fi}$. The resulting proof obligations are true $\Rightarrow 1 \geq 0$ and true $\Rightarrow -1 < 0$. For iterations $\mathbf{do}\ g_1 \rightarrow p_1\ []\ ...\ []\ p_n \rightarrow p_n\ \mathbf{od}$ we have a similar structural combinator [**do**] $t_1, ..., t_n$ [**od**].

In the case we have a variable block we use the structural combinator [**var**] t [$[\![$], which applies the tactic t to the body of the block. For example, if we apply [**var**] $\mathbf{law\ strPost}(x > 0)$ [$[\![$] to $[\![\mathbf{var}\ x : \mathbb{N} \bullet x : [x \geq 0]]\!]$ we get $[\![\mathbf{var}\ x : \mathbb{N} \bullet x : [x > 0]]\!]$ and the proof obligation $x > 0 \Rightarrow x \geq 0$. Similarly, we have the structural combinator [**cons**] t [$[\![$], in the case we have a logical constant block. This structural combinator also applies the tactic t to the body of the block.

The structural combinators [**pmain**] t [$[\![$] and [**pmainvariant**] t [$[\![$] are applied to procedure blocks and variant blocks, respectively. They apply t to the main program of procedure and variant blocks, respectively. For example, if we have the tactic [**pmain**] $\mathbf{law\ strPost}(x > 0)$ [$[\![$] and we apply this tactic to the procedure block $[\![\mathbf{proc}\ nonNeg = x : [x > 0] \bullet x : [x \geq 0]]\!]$ we get $[\![\mathbf{proc}\ p = x : [x > 0] \bullet x : [x > 0]]\!]$ and the proof obligation $x > 0 \Rightarrow x \geq 0$.

When we want to apply a tactic to a procedure body we use the structural combinators [**pbody**] t [$[\![$] and [**pbodyvariant**] t [$[\![$], which apply to procedure blocks and variant blocks, respectively. For example, if we apply the tactic [**pbody**] $\mathbf{law\ assig}(< x >, < 10 >)$ [$[\![$] to the

program $[\![\mathbf{proc}\ nonNeg = x : [x > 0] \bullet nonNeg]\!]$, we get $[\![\mathbf{proc}\ nonNeg = x := 10 \bullet nonNeg]\!]$ and the proof obligation $true \Rightarrow 10 \geq 0$.

In the case we have arguments declaration, we use the structural combinators $\boxed{\mathbf{val}}\ t$, $\boxed{\mathbf{res}}\ t$, and $\boxed{\mathbf{val\_res}}\ t$ depending on whether the arguments are passed by value, result, or value-result, respectively. For example, if we apply the tactic $\boxed{\mathbf{pbody}}\ \boxed{\mathbf{val\_res}}\ \mathbf{law\ strPost}(x > 0)\boxed{[\![}$ to the procedure block $[\![\mathbf{proc}\ nonNegArg = (\mathbf{val\_res}\ x : \mathbb{N} \bullet x : [x \geq 0]) \bullet x : [x > 0]\ ]\!]$, we get the program $[\![\mathbf{proc}\ nonNegArg = (\mathbf{val\_res}\ x : \mathbb{N} \bullet x : [x > 0]) \bullet x : [x > 0]\ ]\!]$ and the proof obligation $x > 0 \Rightarrow x \geq 0$.

The declaration of a tactic has the form

> **Tactic** *name* (*args*) *tactic*
> [**proof obligations** {*predicate*}*] [**program generated** *program*] **end**

where *name* is the name of the tactic being defined, and *args* are its arguments. For documentation purposes, we include clauses **proof obligations** and **program generated**; the former lists the proof obligations generated by the application of *tactic*, and the latter presents the program generated. These two clauses are optional as this information can be infered from the tactic itself. In the next section we give several examples of tactics declarations.

4 Tactics and Examples

In this section we present some tactics and examples of refinements using these tactics. The tactics are based on the development strategies presented in [10, 2, 8].

In the design of a program involving an iteration, the main concern is to determine the iteration invariant. In [8] several strategies are presented, which we capture as tactics.

Tactic takeConjAsInv The first strategy consists of taking a conjunct of the postcondition of the program specification as the main part of the invariant. The tactic **takeConjAsInv** defined below follows this strategy to transform a specification $w : [pre, invconj \wedge notGuard]$ into an initialized iteration.

> **Tactic takeConjAsInv** $(invBound, (lstVar, lstVal), variant)$
> **applies to** $w : [pre, invConj \wedge notGuard]$
> **do** **law strPost**$(invConj \wedge notGuard \wedge invBound)$;
> **law seqCom**$(invConj \wedge invBound)$;
> (**law assig**$(lstVar, lstVal)\boxed{;}$**law iter**$(\langle \neg\ notGuard \rangle, invConj \wedge invBound, variant)$);
> **proof obligations**
> $1.(invConj \wedge notGuard \wedge invBound) \Rightarrow (invConj \wedge notGuard)$
> $2.pre \Rightarrow (invConj \wedge invBound)[lstVal/lstVar]$
> **program generated**
> $lstVar := lstVal$;
> **do** $\neg\ notGuard \rightarrow$
> $w : [invConj \wedge invBound \wedge \neg\ notGuard,$
> $invConj \wedge invBound \wedge 0 \leq variant < variant_0]$
> **od**
> **end**

This tactic has three arguments: a predicate *invBound*, a pair $(lstVar, lstVal)$, where *lstVar*

is a list of variables and *lstVal* is a list of values, and an integer expression *variant*. This tactic applies to a specification statement $w : [pre, invConj \wedge notGuard]$ to introduce an initialized iteration whose invariant is $invConj \wedge invBound$ and whose variant is *variant*. The initialization is $lstVar := lstVal$. Typically, the predicate *invBound* states the range limits of indexing variables of the iteration. The conjunction $invConj \wedge invBound$ is used as invariant of the iteration.

The tactic **takeConjAsInv** first strengthens the postcondition (law **strPost**) using the argument *invBound*, then it introduces a sequential composition (law **seqCom**). Afterwards, the law **assig** is applied to the first program of the composition to derive the initialization, and the law **iter** to the second program in order to introduce the iteration. The first proof obligation is generated by the application of law **strPost**, and the second by the application of law **assig**.

As an example consider the program $q, r : [a \geq 0 \wedge b > 0, q = a \operatorname{div} b \wedge r = a \bmod b]$ which makes q have the value of the division of a by b, and r have the reminder of this division. Using the law **strPost** we can refine it to $q, r : [a \geq 0 \wedge b > 0, a = q * b + r \wedge 0 \leq r \wedge r < b]$ because, this new postcondition implies the old one, based on the definition of div and mod.

Consider the algorithm that first initializes q and r with values 0 and a, respectively, and uses q to count how many times r can be reduced by b. To develop such an algorithm it is convenient to use $a = q * b + r \wedge 0 \leq r \wedge r < b \wedge b > 0$ as an invariant. We use then the tactic **takeConjAsInv** with arguments $b > 0$, the part of the loop invariant that is not in the postcondition of the specification, $(\langle q, r\rangle, \langle 0, a\rangle)$, the initialization of the variables, and $r < b$, the termination condition of the loop, and obtain the program

$$
\begin{array}{l}
q, r := 0, a; \\
\textbf{do}\ \ r \geq b \rightarrow \\
\quad q, r : [a = q * b + r \wedge 0 \leq r \wedge b > 0 \wedge r \geq b, \\
\qquad\quad a = q * b + r \wedge 0 \leq r \wedge b > 0 \wedge 0 \leq r < r_0] \\
\textbf{od}
\end{array}
$$

The first proof obligation is $(0 \leq x \wedge x^2 \leq n \wedge (x+1)^2 > n) \Rightarrow (x^2 \leq n \wedge (x+1)^2 > n)$ which is a tautology, since $a \wedge b \Rightarrow b$. The second proof obligation requires us to prove that $(x^2 \leq n \wedge 0 \leq x \wedge (x+1)^2 \leq n \wedge x = x_0)$ implies that $(x+1)^2 \leq n$, which is in the antecedent, that $0 \leq x + 1$, which holds by $0 \leq x$, that $0 \leq n - (x+1)^2$, which follows from $(x+1)^2 \leq n$, and finally that $n - (x+1)^2 < n - {x_0}^2$, which holds since $(x+1)^2 > x^2$ and $x = x_0$.

The tactic is used as it were a single (very powerful) refinement rule. All that is left after its application is to refine the body of the iteration in the standard way to introduce the assignment $q, r; = q + 1, r - b$.

Tactic replConsByVar Another common way of choosing an iteration invariant is replacing a constant in the specification postcondition by a variable. This strategy is captured by the tactic **replConsByVar**.

This tactic has five arguments: the variable declaration $newV : T$, the constant c, and the arguments taken by the tactic **takeConjAsInv**. It applies to any specification $w : [pre, post]$ to introduce an initialized iteration whose invariant is $post[newV/cons] \wedge invBound$ and whose variant is *variant*. The initialization is $lstVar := lstVal$. In this tactic the predicate *invBound* states the range limits of the new variable *newV* which is used as a indexing variable of the

iteration.

Tactic replConsByVar $(newV : T, cons, invBound, (lstVar, lstVal), variant)$
applies to $w : [pre, post]$
do **law varInt**$(newV : T)$;
  $\boxed{\textbf{var}}$ **law strPost**$(post[newV/cons] \wedge newV = cons)$;
    tactic takeConjAsInv$(invBound, (lstVar, lstVal), variant)$; $\boxed{[\![}$
proof obligations
  $1.(post[newV/cons] \wedge newV = cons) \Rightarrow post$
  $2.(post[newV/cons] \wedge newV = cons \wedge invbound) \Rightarrow$
    $(post[newV/cons] \wedge newV = cons)$
  $3.pre \Rightarrow (post[newV/cons] \wedge invBound)[lstVal/lstVar]$
program generated
  $[\![$**var** $newV : T \bullet$
    $lstVar := lstVal$;
    do $\neg\ newV = cons \rightarrow$
      $newV, w : [post[newV/cons] \wedge invBound \wedge \neg\ newV = cons,$
        $post[newV/cons] \wedge invBound \wedge variant]$
    od
  $]\!]$
end

This tactic first introduces the new variable(law **varInt**), then it strengthens the postcondition to substitute the constant by the new variable. Afterwards this tactic calls the tactic **takeConjAsInv** to introduce the iteration.The proof obligation 1 is generated by the law **strPost**, and the others are generated by the tactic **takeConjAsInv**.

As an application example we use the program $r : [a \geq 0 \wedge b \geq 0, r = a^b]$ which makes the variable r receive the value of a^b. The idea of the algorithm we want to derive is to use a variable x like a counter from 0 to b. This variable is initialized with 0 and r is initialize with 1. In each step of the iteration r receives the value $r * a$ and x is incremented by one. This loops ends when $x = b$. We apply the tactic **replConsByVar** with arguments $x : \mathbb{Z}$, the variable which is used to substitute the constant b in the second argument, $0 \leq x \leq b$, the bound limits of x, $(\langle x, r\rangle, \langle 0, 1\rangle)$, the initialization of x and r, and $b - x$, the variant of the iteration. The result of applying **replConsByVar** with these arguments is as follows

$[\![$**var** $x : \mathbb{Z} \bullet$
  $x, r := 0, 1$;
  do $x \neq b \rightarrow$
    $r, x : [r = a^x \wedge 0 \leq x \leq b \wedge x \neq b,$
      $r = a^x \wedge 0 \leq x \leq b \wedge 0 \leq b - x \leq b - x_0]$ **od** $]\!]$

with proof obligations $(r = a^x \wedge x = b) \Rightarrow (r = a^b)$ and $(r = a^x \wedge x = b \wedge 0 \leq x \leq b)$ implies that $(r = a^x \wedge x = b)$, which are tautologies. A third proof obligation is $(a \geq 0 \wedge b \geq 0)$ implies that $1 = a^0$, which holds by a property of the power operator, and that $0 \leq 0 \leq b$, which is the antecedent.

Again, the refinement of the body of the iteration is standard. We only have to apply the law **assig2** to get the attribution $r, x := r * a, x + 1$.

The tactic **takeConjAsInv** embodies what is probably the most commonly used strategy for the developments of iterations. It is used in the more elaborated tactic above, which is in turn used in the following tactic. From these examples, it is clear that tactics not only shorten developments but also document techniques.

Tactic strengInv Sometimes we cannot simply replace a constant by a variable in the post-condition of the specification to determine the invariant. First, we have to change (strenghten) the postcondition.

The tactic **strengInv** has seven arguments: a variable declaration $newV1 : T1$, a predicate *streng* and the rest of the arguments are the same as those taken by the tactic **replConsByVar**. This tactic applies to a specification statement $w : [pre, post]$ to introduce an initialized iteration whose invariant is $(post \wedge streng)[newV2/cons] \wedge invBound$ and variant is *variant*. The initialization is $lstVar := lstVal$. In this tactic the predicate *invBound* states the range limits (typically 0 and *cons*) of the new variable $newV2$ which is used as an indexing variable of the interation. The new variable $newV1$ is used as an auxiliary variable and the predicate *streng* is used to make a link between the new variable introduced and the data used in the specification.

Tactic strengInv
$(newV1 : T1, streng, newV2 : T2, cons, invBound, (lstVar, lstVal), variant)$
applies to $w : [pre, post]$
do **law varInt**$(newV1 : T1)$;
var **law strPost**$(post \wedge streng)$;
tactic replConsByVar$(newV2 : T2, cons, invBound, (lstVar, lstVal), variant)$;
$[\![$
proof obligations
$1.(post \wedge streng) \Rightarrow post$
$2.((pos \wedge streng)[newV2/cons] \wedge newV2 = cons) \Rightarrow (pos \wedge streng)$
$3.((pos \wedge streng)[newV2/cons] \wedge newV2 = cons \wedge invBound) \Rightarrow$
$((pos \wedge streng)[newV2/cons] \wedge newV2 = cons)$
$4.pre \Rightarrow ((pos \wedge streng)[newV2/cons] \wedge invBound)[lstVal/lstVar]$
program generated
$[\![$**var** $newV1 : T1 \bullet$
$[\![$**var** $newV2 : T2 \bullet$
$lstVar := lstVal;$
do $\neg\ newV2 = cons \rightarrow$
$newV1, newV2, w : [(post \wedge streng)[newV2/cons] \wedge$
$invBound \wedge \neg\ newV2 = cons,$
$(post \wedge streng)[newV2/cons] \wedge$
$invBound \wedge variant]$
od $]\!]\ ]\!]$
end

The first step of this tactic is to introduce a new variable which3 is used to change (strenghten) the postcondition. Then, this tactic strengthens the postcondition and calls the tactic **replConsByVar**.

For example, consider $r : [n \geq 0, r = \#\{i, j : \mathbb{N} \mid 0 \leq i < j < n \wedge f[i] \leq 0 \wedge f[j] \geq 0\}]$ which counts how many pairs (i, j) there are in the sequence, such that $f[i] \leq 0 \wedge f[j] \geq 0 \wedge i < j$.

We want to derive an algorithm that makes a linear search in the sequence using a new variable, say m, to index the sequence. The algorithm uses a new variable s, initialized with 0, to count how many non-positive numbers have been found in the search. Once the search finds a non-negative number, the algorithm increments r by the number of non-positive numbers that have been found, s.

To develop this algorithm from the specification we use the tactic **strengInv** with arguments $s : \mathbb{Z}$, the integer which counts the number of non-positive numbers found in the linear search, $s = \#\{i : \mathbb{N} \mid 0 \leq i < n \wedge f[i] \leq 0\}$, the specification of s which is used to strenghten the post-condition, $m : \mathbb{N}$, the integer which is used to index the sequence in the search, n, the constant which will be replaced by m, $0 \leq m < n$, the bound limits of m, $(\langle m, r, s\rangle, \langle 0, 0, 0\rangle)$, the initialization of the variables, and $n - m$, the variant of the iteration.

This application results in the program

$$
\begin{array}{l}
[\![\textbf{var}\, s : \mathbb{Z} \bullet \\
\quad [\![\textbf{var}\, m : \mathbb{Z} \bullet \\
\qquad m, s, r := 0, 0, 0; \\
\qquad \textbf{do}\ \ m \neq n \rightarrow \\
\qquad\quad r, s, m : \left[\begin{array}{l} \left(\begin{array}{l} r = \#\{i, j : \mathbb{N} \mid 0 \leq i < j < m \wedge f[i] \leq 0 \wedge f[j] \geq 0\} \\ s = \#\{i : \mathbb{N} \mid 0 \leq i < m \wedge f[i] \leq 0\} \wedge 0 \leq m \leq n \wedge m \neq n \end{array} \right), \\ \\ \left(\begin{array}{l} r = \#\{i, j : \mathbb{N} \mid 0 \leq i < j < m \wedge f[i] \leq 0 \wedge f[j] \geq 0\} \\ s = \#\{i : \mathbb{N} \mid 0 \leq i < m \wedge f[i] \leq 0\} \wedge 0 \leq n - m < n - m_0 \end{array} \right) \end{array} \right] \\
\qquad \textbf{od} \\
\quad]\!] \\
]\!]
\end{array}
$$

At this point, the tactic application of the tactic has already finished, we only need to refine the body of the iteration to the program

$$
\begin{array}{l}
\textbf{if}\ f[m] < 0 \rightarrow \textbf{skip};\ [\!]\ f[m] \geq 0 \rightarrow r := r + s;\ \ \textbf{fi} \\
\textbf{if}\ f[m] > 0 \rightarrow \textbf{skip};\ [\!]\ f[m] \leq 0 \rightarrow s := s + 1;\ \ \textbf{fi} \\
m := m + 1;
\end{array}
$$

This can be accomplished using the laws **fassig**, **seqCom**, **alt**, **skipIntro**, and **assig2** .

Tactic tailInvariant This strategy is used when we want to develop an algorithm involving an iteration whose invariant is based on a function defined using tail recursion.

This tactic applies to a specification statement $w : [pre, post]$ to introduce an initialized iteration whose invariant is $invConj \wedge invBound$ and variant is $variant$, and a final assignment $lstVar_2 := lstVal_2$, which typically is an assignment of an element of a sequence identified by an index found in the iteration body, to a variable. The initialization of the iteration is $lstVar_1 := lstVal_1$. Inside the iteration body this tactic includes an alternation.

This tactic has ten arguments: the first two arguments are variable declarations, which are used as indexing variables in the iteration. The next argument is a predicate $invBound$. It is used to make a conjunction with the next argument, $notGuard$, which is also a predicate and represents the negation of the guard of the iteration. The next argument is a pair $(lstVar_1, lstVal_1)$, where

$lstVar_1$ is a list of variables and $lstVal_1$ is a list of values. They are used in the initialization of the iteration. An integer expression *variant* is the next argument and represents the variant of the iteration. The next two arguments *guardsList* and *tacticsList* are used to call the tactic **alternation**, which generates an alternation with the guards given as argument and applies the tactics t_i of the list of tactics also given as argument to the corresponding guarded programs. Its definition is very simple and is omitted for the sake of conciseness. Finally we have a pair $(lstVar_2, lstVal_2)$, where $lstVar_2$ is a list of variables and $lstVal_2$ is a list of values. They are used in the final assignment after the iteration has finished.

Tactic tailInvariant
$(var1 : T1, var2 : T2, invConj, notGuard, invBound, (lstVar_1, lstVal_1),$
$variant, guardsList, tacticsList, (lstVar_2, lstVal_2))$
applies to $w : [pre, post]$
do **law varInt**$(var1 : T1, var2 : T2)$;
var **law seqCom**$(invConj \wedge notGuard)$;
(**tactic takeConjAsInv**$(invBound, (lstVar_1, lstVal_1), variant)$;
(**skip** ; **do** **tactic alternation**$(guardsList, tacticsList)$**od**))
;
law assig$(lstVar_2, lstVal_2)$; $[\![]\!]$
proof obligations
$1.(invConj \wedge notGuard \wedge invBound) \Rightarrow (invConj \wedge notGuard)$
$2.\ pre \Rightarrow (invConj \wedge invBound)[lstVal/lstVar]$
$3.\ (invConj \wedge invBound \wedge \neg\ notGuard) \Rightarrow \bigvee guardsList$
$4.\ invConj \wedge notGuard \Rightarrow post[lstVal/lstVar]$
program generated
$[\![$**var** $var1 : T1$; **var** $var2 : T2 \bullet$
$lstVar_1 := lstVal_1$;
do $\neg\ notGuard \rightarrow$
if$(?i.guardsList_i \rightarrow$
$(tacticsList_i\ w : [guardsList_i \wedge invConj \wedge invBound \wedge \neg\ notGuard,$
$invConj \wedge invBound \wedge 0 \leq variant < variant_0]))$ **fi**
od
$lstVar_2 := lstVal_2$;
$]\!]$
end

The tactic **tailInvariant** first introduces two variables (arguments $var1 : T_1$ and $var2 : T_2$). Afterwards it splits the body of the variable block into a sequential composition of two other specifications. The first defines the initialized iteration and the second the final assignment. The tactic applies the tactic **takeConjAsInv** to the first program using the *invBound*, $(lstVar_1, lstVal_1)$ and *variant* arguments. The tactic also applies the **law assig** to the second program using the argument $(lstVar_2, lstVal_2)$. Inside the iteration body, the tactic applies the tactic **alternation** using the arguments *guardsList* and *tacticsList*. The proof obligations 1 and 2 are generated by the tactic **takConjAsInv**. The proof obligation 3 is generated by the law **alt**, and finally the proof obligation 4 is generated by the law **assig**.

As an example we use a program that returns the maximum element of an integer sequence. We have the specification statement $r : [n \geq 0, r = F(0, n)]$, which assigns to r the value of

the maximum element of a sequence A. The function $F\ (n, m)$ returns the maximum element of A between its nth and mth elements. We apply the tactic **tailInvariant** with arguments $x : \mathbb{N}$ and $y : \mathbb{N}$, which are the auxiliar indexing variables,$F(x, y) = F(0, n) \land 0 \leq x \leq y \leq n$, one of the conjuncts which forms the invariant of the iteration, $x = y$, the negation of the iteration guard, *true*, because we do not need to bound the values of the indexing variables in the iteration invariant, $(\langle x, y\rangle, \langle 0, n\rangle)$, the initialization of the iteration, $y - x$, the variant of the iteration, $\langle A[x] \leq A[y], A[x] > A[y]\rangle$, the list of guards of the alternation, the list of tactics $\langle \mathbf{assig2}(\langle x\rangle, \langle x + 1\rangle), \mathbf{assig2}(\langle y\rangle, \langle y - 1\rangle)\rangle$, used in the tactic **alternation** call, and finally $(\langle r\rangle, \langle A[x]\rangle)$, the final assignment. The application of this tactic results in the program

$$
\begin{array}{l}
[\![\mathbf{var}\, x : \mathbb{N};\ y : \mathbb{N} \bullet \\
\quad x, y := 0, 0; \\
\quad \mathbf{do}\ x \neq y \rightarrow \\
\qquad \mathbf{if}\ A[x] \leq A[y] \rightarrow x := x + 1; \\
\qquad\ \ [\!]\ A[x] > A[y] \rightarrow y := y - 1; \\
\qquad \mathbf{fi} \\
\quad \mathbf{od} \\
\quad r := A[x];\ ]\!]
\end{array}
$$

with proof obligations which are long but simple.

The strategy presented in [10] for the development of programs involving procedures is not entirely based on laws, especially as far as recursive procedures are concerned. Moreover an inconsistency has been found in that work [3]. Therefore, we consider the approach of [2], where a set of refinement laws to deal with procedures is presented. These laws, however, are very simple (see Appendix A) and the development of programs usually requires the use of several of them. The following tactics capture commonly used strategies.

Tactic procNoArgs The **tactic procNoArgs** introduces a procedure block that declares a procedure with no parameters. It has two arguments: the name of the procedure which is introduced and its body. This tactic applies to a program p, introduces the procedure, and makes one or more calls to it, depending on the program p.

Tactic procNoArgs $(procName, procBody)$
applies to p
do law procNoArgsIntro$(procName, procBody)$;
  law procNoArgsCall;
program generated $[\![\mathbf{proc}\ procName = procBody \bullet p[procBody \setminus procName]]\!]$
end

The program $p[procBody \setminus procName]$ is that obtained by replacing all occurencies of $procBody$ in p with $procName$.

As an example we consider the program which orders three integers in an increasing order. We can specify such a program as $p, q, r : [p \leq q \leq r \land \lfloor p, q, r\rfloor = \lfloor p_0, q_0, r_0\rfloor]$, where $\lfloor p, q, r\rfloor$ (resp.$\lfloor p_0, q_0, r_0\rfloor$) is the bag with elements p, q and r (resp. p_0, q_0 and r_0). First, we can refine this program to $p, q := p \sqcap q, p \sqcup q;\ q, r := q \sqcap r, q \sqcup r;\ p, q := p \sqcap q, p \sqcup q$. We can then apply the tactic **procNoArgs** with arguments *sort*, the name of the procedure, and $p, q := p \sqcap q, p \sqcup q$, the body of the procedure, and get as result the procedure block $[\![\mathbf{proc}\ sort = p, q := p \sqcap q, p \sqcup q \bullet\ sort;\ q, r := q \sqcap r, q \sqcup r;\ sort]\!]$.

Tactic procCalls Before describing a tactic which introduces procedures with arguments it is useful to define a tactic that introduces parameterized commands. The **tactic procCalls** takes as arguments two lists. The first is a list of parameter declarations and the second is a list of arguments. The declarations have the form **k** $v : T$ where **k** defines how the argument is passed, by value(**val**), by result(**res**) or by value-result(**val_res**), v is the name of the argument, and T its type.

This tactic tries to derive an application of a procedure call with the given parameters to the given arguments. With this purpose, this tactic tries to apply each of the laws that introduces parameterized commands. If one of them succeeds, the tactic goes on with the tail of the list, else, the tactic behaves like **skip** and finishes.

Tactic procCalls $(pars, args)$
applies to $w : [pre, post]$
do (**law callByValue1**$(head'\ args, head'\ pars)$;
  val **tactic procCalls**$(tail\ pars, tail\ args)$) |
  (**law callByValue2**$(head'\ args, head'\ pars)$;
    (**val** **law hideFrame**$(head'\ pars)$);
      tactic procCalls$(tail\ pars, tail\ args)$) |
  (**law callByResult**$(head'\ args, head'\ pars)$;
    res **tactic procCalls**$(tail\ pars, tail\ args)$ |
  (**law callByValueResult**$(head'\ args, head'\ pars)$;
    val_res **tactic procCalls**$(tail\ pars, tail\ args)$) |
  skip
end

The function $head'$ applies to a list, and gives another list that contains just the head of the given list, or is empty if the given list is empty. This tactic is recursive and can generate applications of parameterized commands whose bodies can include further applications. An example of the use of this tactic is presented below.

Tactic procArgs Now we can define the **tactic procArgs** which introduces a parameterized procedure in the scope of the program and makes calls to this procedure. Here the argument *pars* is a (possibly multiple) parameter declaration. We make use of the function *seqToList* to convert *args*, a ;-separated sequence of arguments declarations, to a list of declarations.

Tactic procArgs $(procName, pars, body, args)$
applies to p
do **law procArgsIntro**$(procName, pars, body)$;
  pmain
    tactic procCalls$(seqToList\ pars, args)$; $exhaust$(**law multiArgs**) $[\![]\!]$;
  law procArgsCall
end

This tactic first introduces a procedure with arguments using the law **procArgsIntro**. Then the tactic uses **procCalls** to introduce applications of parameterized commands with parameters *pars* to arguments *args*. Finally, the tactic uses the law **multiArgs** to nested applications of parameterized commands.

As an example we use a program which substitutes the value of a variable by its square value. The specification of such a program can be $x : [0 \leq x, x^2 = x_0]$. We use the tactic **procArgs** with arguments *sqrt*, the name of the procedure, **val** $a : \mathbb{R}$; $b : \mathbb{R}$, the procedure parameters, $b : [0 \leq a, b^2 = a]$, the body of the procedure, and $< x, x >$, the arguments used in the procedure call. The tactic first introduces the procedure using the law **procArgsIntro**. Then, the tactic calls **procCalls** and the body of the procedure is refined to (**val** $a : \mathbb{R} \bullet$ (**res** $b : \mathbb{R} \bullet b : [0 \leq a, b^2 = a]$)($x$))($x$). The next step, the application of the tactic *exhaust*(**law multiArgs**), results in (**val** $a : \mathbb{R}$, **res** $b : \mathbb{R} \bullet b : [0 \leq a, b^2 = a]$)($x, x$). Finally, the tactic applies the law **procArgsCall** to the whole procedure block and we get the program $[\![$ **proc** *sqrts* = (**val** $a : \mathbb{R}$, **res** $b : \mathbb{R} \bullet b : [0 \leq a, b^2 = a]$) $\bullet$ *sqrts*(x, x) $]\!]$.

Tactic recProcArgs This tactic is useful when we want to develop a recursive procedure and have to introduce a variant block with a parameterized procedure. Its definition is

Tactic recProcArgs
($procName, args, variantName, variantExp, body, varsProcF, guardsList, tacticsList$)
applies to p
do **law variantIntro**($procName, args, variantName, variantExp, body$);
pmainvariant
tactic procCalls($seqToList\ args, varsProcF$);
exhaust(**law multiArgs**); $[\![]\!]$;
law procArgsVariantBlockCall;
pbodyvariant
law absAssump; **tactic alternation**($guardsList, tacticsList$); $[\![]\!]$;
law recursiveCall
end

This tactic first introduces a variant block using the law **variantIntro**. Then, as in the previous tactic, it uses the law **procCalls** and the tactic *exhaust*(**law multiArgs**) to introduce an application of a parameterized command. Afterwards, the tactic uses the law **procArgsVariantBlockCall** to introduce a procedure call in the main program of the variant block. The next step of this tactic is to refine the body of the procedure. First, the tactic uses the law **absAssump** to move the assumption that defines the variant, to the precondition of the specification. Then, the tactic applies the tactic **alternation**. Finally the tactic uses the law **recursiveCall** to introduce a recursive call to the program.

As an example, suppose we want to develop a program which calculates the factorial of an integer n. Its specification can be $f : [f = n!]$. The idea is to use a recursive procedure *fact*. We only have to apply **recProcArgs** with arguments *fact*, the name of the procedure, **val** $m : \mathbb{N}$, the argument of *fact*, V, the variant name, m, the variant itself, $f : [f = m!]$, the body of *fact*, $< n >$, the argument of the call in the main program, $< m = 0, m > 0 >$, the guards of the alternation, and a list of the tactics **law assig**($\langle f \rangle, \langle 1 \rangle$) and the sequential composition of **law strPost**($f = m * (m-1)!$), **law callByValue1**($< m, k >, < m - 1, m * k >$), and val (**law weakPre**($0 \leq m < V$); **law absAssump**). This application generates the program

$[\![$ *proc fact* = (**val** $m : \mathbb{N} \bullet$ **if** $m = 0 \rightarrow f := 1 \ [] \ m > 0 \rightarrow f := m * fact(m-1)$ **fi**)
variant V **is** $m \bullet$ *fact*(n) $]\!]$

A few proof obligations are generated but they are simple.

5 Conclusions

We have presented RTL, a language suitable for the definition of refinement tactics. Using RTL we can specify commonly used strategies of program development and use them as transformation rules. This not only shortens the developments, but also improves their readability.

Furthermore, we have defined a number of refinement tactics that capture standard strategies for the development of iterations and more recent formalizations of strategies for the development of procedural abstractions. These strategies have already been put forward in the literature mainly by means of examples, but as far as we know, they have never been formalized and expressed as a transformation rule.

Our language is independent of any particular tool of programming language. On the other hand, we are developing a tool to support the refinement calculus [4] and we intend to extend it to provide support for RTL. The refinement tactics presented here are going to compose a basic repository of tactics. The user is going to be able to apply these tactics as well as define new ones.

The formal semantics of RTL, along with algebraic laws that allows reasoning about RTL tactics will appear elsewhere. This work is based on the Angel semantics. The difference is that Angel is a very general language and its goals have no defined structure. For us, a goal is called *RMCell*, which is a pair with a program as its first element and a list of proof obligations as its second element. The application of a tactic to a *RMCell* returns a list of *RMCells* which is the possible output programs with their equivalent list of proof obligations.

As already mentioned, to our knowledge, the only other work on refinement tactics is that presented in [7], where basically Prolog is used as a tactic language. It is not possible to define a tactic recursively and there are no structural combinators. Also, only a few examples of simple tactics are provided there.

A Laws of Refinement Calculus

Law strPost$(post_2)$ $w : [pre, post_1] \sqsubseteq w : [pre, post_2]$ provided $post_2 \Rightarrow post_1$

Law weakPre(pre_2) $w : [pre_1, post] \sqsubseteq w : [pre_2, post]$ provided $pre_1 \Rightarrow pre_2$

Law assig$(\langle w \rangle, \langle E \rangle)$ $w : [pre, post] \sqsubseteq w := E$ provided $pre \Rightarrow post[w \setminus E]$

Law assig2$(\langle w \rangle, \langle E \rangle)$ $w, x : [pre, post] \sqsubseteq w := E$ provided $(w = w_0) \wedge pre \Rightarrow post[w \setminus E]$

Law fassig$(\langle x \rangle, \langle E \rangle)$For any term E $w, x : [pre, post] \sqsubseteq w, x : [pre, post[x \setminus E]];\ x := E$

Law seqCom(mid) For any mid $w : [pre, post] \sqsubseteq w : [pre, mid];\ w : [mid, post]$

Law alt$(\langle G_0, ..., G_n \rangle)$ $w : [pre, post] \sqsubseteq$ **if** $(?\ i.G_i \rightarrow w : [G_i \wedge pre, post])$ **fi**
provided $pre \Rightarrow G_0 \vee ... \vee G_n$

Law iter$(\langle G_1, ..., G_n \rangle, inv, V)$ For any formula inv, the invariant; and any integer expression V, the variant $w : [pre, post] \sqsubseteq$ **do** $(?\ i.G_i \rightarrow w : [inv \wedge G_i, inv \wedge 0 \leq V \leq V[w \setminus w_0]])$ **od**

Law varInt$(n : T)$ $w : [pre, post] \sqsubseteq [\![$**var** $x : T \bullet w, x : [pre, post]]\!]$
provided x doen not occurs in w, pre, and $post$

Law skipIntro $w, x : [pre, post] \sqsubseteq$ **skip** provided $w = w_0 \wedge pre \Rightarrow post$

Law procNoArgsIntro(pn, p_1) $p_2 = [\![\mathbf{proc}\, pn = p_1 \bullet p_2]\!]$ provided pn is not free in p_2

Law procNoArgsCall $[\![\mathbf{proc}\, pn = p_1 \bullet p_2[p_1]]\!] = [\![\mathbf{proc}\, pn = p_1 \bullet p_2[pn]]\!]$

Law procArgsIntro(pn, par, p_1) $p_2 = [\![\mathbf{proc}\, pn = (par \bullet p_1) \bullet p_2]\!]$
provided pn is not free in p_2

Law procArgsCall
$[\![\mathbf{proc}\, pn = (par \bullet p_1) \bullet p_2[(par \bullet p_1)(a)]]\!] = [\![\mathbf{proc}\, pn = (par \bullet p_1) \bullet p_2[pn(a)]]\!]$

Law callByValue1(f, a) $w : [pre[f \setminus a], post[f, f_0 \setminus a, a_0]] = (\mathbf{val}\, f \bullet w : [pre, post])(a)$ provided f is not in w and w is not free in a

Law callByValue2(f, a) $w : [pre[f \setminus a], post[f_0 \setminus a_0]] = (\mathbf{val}\, f \bullet w, f : [pre, post])(a)$ provided f is not in w and w is not free in $post$

Law callByResult(f, a) $w, a : [pre, post] = (\mathbf{res}\, f \bullet w, f : [pre, post[a \setminus f]])(a)$
provided f is not in w, and is not free in pre or $post$, and and f_0 is not free in $post$

Law callByValueResult(f, a)
$w, a : [pre[f \setminus a], post[f_0 \setminus a_0]] = (\mathbf{val\_res}\ f \bullet w, f : [pre, post[a \setminus f]])(a)$
provided f is not in w, and is not free $post$

Law hideFrame(x) $w, x : [pre, post] \sqsubseteq w : [pre, post[x_0 \setminus x]]$

Law variantIntro$(pr, pars, n, e, p_1)$
$p_2 = [\![\mathbf{proc}\, pr = (par \bullet \{n = e\} p_1)\, \mathbf{variant}\ n\ \mathbf{is}\ e \bullet p_2]\!]$ provided pr and n are not free in e and p_2

Law multiArgs $(\mathbf{par_1}\ f_1 \bullet (\mathbf{par_2}\ f_2)(a_2))(a_1) = (\mathbf{par_1}\ f_1;\ \mathbf{par_2}\ f_2)(a_1, a_2)$

Law procVariantBlockCall $[\![\mathbf{proc}\ pr = (par \bullet p_1)\, \mathbf{variant}\, n\, \mathbf{is}\, e \bullet p_2[(par \bullet p_3)(a)]]\!]$
$= [\![\mathbf{proc}\ pr = (par \bullet p_1)\, \mathbf{variant}\, n\, \mathbf{is}\, e \bullet p_2[(pr)(a)]]\!]$
provided pr is not recursive, n is not free in e and p_3, and $\{n = e\} p_3 \sqsubseteq p_1$

Law absAssump $\{pre'\} w : [pre, post] = w : [pre' \wedge pre, post]$

Law recursiveCall $[\![\mathbf{proc}\ pr = (par \bullet p_1[(par \bullet \{0 \leq e < n\} p_3)(a)])\, \mathbf{variant}\, n\, \mathbf{is}\, e \bullet p_2]\!]$
$= [\![\mathbf{proc}\ pr = (par \bullet p_1[pr(a)])\, \mathbf{variant}\, n\, \mathbf{is}\, e \bullet p_2]\!]$ provided n is not free in p_3 and $p_1[pr(a)]$, and $\{n = e\} p_3 \sqsubseteq p_1[(par \bullet \{0 \leq e < n\} p_3)(a)]$

References

[1] R. J. R. Back. Procedural Abstraction in the Refinement Calculus. Technical report, Department of Computer Science, Åbo - Finland, 1987. Ser. A No. 55.

[2] A. L. C. Cavalcanti, A. Sampaio, and J. C. P. Woodcock. Procedures, Parameters, and Substitution in the Refinement Calculus. Technical Report TR-5-97, Oxford University Computing Laboratory, Oxford - UK, February 1997.

[3] A. L. C. Cavalcanti, A. Sampaio, and J. C. P. Woodcock. An Inconsistency in Procedures, Parameters, and Substitution in the Refinement Calculus. Science of Computer Programming. pages 33(1):87–96, 1999.

[4] S. L. Coutinho, T. P. C. Reis, and A. L. C. Cavalcanti. Uma Ferramenta Educacional de Refinamentos. In *XIII Simpósio Brasileiro de Engenharia de Software*, pages 61 – 64, 1999.

[5] E. W. Dijkstra. *A Discipline of Programming.* Prentice-Hall, 1976.

[6] J. Goguen, A. Stevenes, K. Hobley, and H. Hilberdink. 2OBJ, A Metalogical Framework Based on Equational Logic. *Philosophical Transactions of the Royal Society, Series A*, 339:69 – 86, 1992.

[7] L. Groves, R. Nickson, and M. Utting. A Tactic Driven Refinement Tool. In C. B. Jones, R. C. Shaw, and T. Denvir, editors, *5th Refinement Workshop*, Workshops in Computing, pages 272 – 297. Springer-Verlag, 1992.

[8] A. Kaldewaij. *Programming: The Derivation of Algorithms.* Prentice-Hall, 1990.

[9] A. P. Martin, P. H. B. Gardiner, and J. C. P. Woodcock. A Tactical Calculus. *Formal Aspects of Computing*, 8(4):479–489, 1996.

[10] C. C. Morgan. *Programming from Specifications.* Prentice-Hall, 2nd edition, 1994.

[11] L. C. Paulson. *ML for the Working Programmer.* CUP, 1991.