Refine and Gabriel: Support for Refinement and Tactics

Marcel Oliveira
Computing Laboratory
University of Kent
CT2 7NF
Canterbury, England
mvmo2@kent.ac.uk

Manuela Xavier
Centro de Informática
Universidade Federal de Pernambuco
Caixa Postal 7851,
Cidade Universitária
Recife, Brazil
max@cin.ufpe.br

Ana Cavalcanti
Computing Laboratory
University of Kent
CT2 7NF
Canterbury, England
A.L.C.Cavalcanti@kent.ac.uk

Abstract

Using Morgan's refinement calculus, we can write software in a precise and consistent way. Nevertheless, this may involve long and repetitive developments. Several refinement strategies are useful in different developments, and even in different points of a single development. A lot is gained by identifying these strategies, documenting them as tactics, and using them as single transformation rules. With this motivation, we have designed ArcAngel, a tactic language especially tailored for refinement; we have formalised its semantics and studied its algebraic laws. Even with the use of tactics, however, refinement can be a hard task and the use of tools is essential in practice. In this paper, we present Refine and Gabriel, interactive, user-friendly tools that allow us to use the refinement calculus with the support of ArcAngel tactics.

1. Introduction

Morgan's refinement calculus [17] is a successful technique to develop programs while guaranteeing correctness. From a formal specification, we obtain a program by repeatedly applying correctness-preserving transformation rules (refinement laws).

Using the refinement calculus, however, can be a hard task, as program developments may be long and repetitive. Frequently used strategies of refinement are reflected in sequences of laws that are applied over and over again. Identifying these strategies, documenting them as tactics, and using them in program developments as single transformation rules, is a great help.

We designed and formalised a refinement-tactic language called ArcAngel [22]. It is based on Angel [16], a general language that makes no assumption about the form of proof goals, or about the rules that are applied to them. ArcAngel considers the fact that, by applying refinement laws to a program, we produce not only a program, but also proof obligations. The constructs of ArcAngel are similar to those of Angel, but are adapted to deal with refinement laws and programs. ArcAngel also provides structural combinators which are suitable to apply refinement laws to components of programs.

The semantics of ArcAngel is an adaptation and extension of that of Angel. It can be found in [19, 22], along with over seventy laws of reasoning, their proofs, and their use in a reduction strategy to a normal form. In [21], ArcAngel is used to formalise a large number of refinement strategies available in the literature [15, 17]. Nonetheless, using ArcAngel without tool support is still a hard task. A tool brings further profit in time and effort, and was left as future work in [22].

We present Refine, a tool that supports the use of the refinement calculus, and a plug-in called Gabriel, which supports the use of ArcAngel, and allows its users to create and use tactics of development. An initial version of Refine was presented in [8]; since then, we have extended it with facilities to manage developments, and support for the development of, possibly recursive, procedures. Refine has been used successfully in teaching for almost three years.

In Section 2 we give an overview of the refinement calculus. Section 3 introduces ArcAngel; we present an example of a tactic. Section 4 presents Refine, and Section 5 presents Gabriel and its integration to Refine. Finally, Section 6 discusses related and future work.

2. Refinement Calculus

The refinement calculus is based on an unified language of specification, design and implementation; it makes no distinction between specifications and programs. In this technique, program development consists of refinement law applications to a specification until an adequate program is obtained.

A specification has the form $w : [pre, post]$. It describes a program that, if executed in a state that satisfies the precondition *pre*, changes the variables listed in the frame *w*, so that the final state satisfies the postcondition *post*. If the initial state does not satisfy the precondition, the result cannot be predicted. A precondition *true* can be omitted. The pre and the postcondition are predicates.

In the postcondition, the initial value of a variable is represented by a corresponding 0-subscripted variable. For example, the specification statement $x : [x = x_0^n]$ defines the program that assigns to *x* the *n*th power of its initial value.

Besides the specification statement, the language of the refinement calculus includes all the constructors of Dijkstra's language [9] of guarded commands. Block constructs are also available to declare local variables, logical constants, and procedures. Variable blocks $[\![\mathbf{var}\ \ x : T \bullet p]\!]$ declare a variable *x* to be of type *T*, with a scope restricted to *p*. Similarly, logical constants *c* are declared in blocks of the form $[\![\mathbf{con}\ c : T \bullet p]\!]$.

Procedure blocks $[\![\mathbf{proc}\ \ pname \mathrel{\widehat{=}} body \bullet main]\!]$ declare a procedure *pname* and its *body*, along with the *main* program, where we can use *pname*. If *pname* has parameters, then *body* is a parameterized command [2, 6]. Parameters can be passed by value, by result, or by value-result using the keywords **val**, **res**, and **val-res**. An example is presented below.

$$\begin{array}{l}[\![\mathbf{proc}\ power \mathrel{\widehat{=}} \\ \qquad (\mathbf{val\text{-}res}\ x : \mathbb{N};\ \mathbf{val}\ n : \mathbb{N} \bullet x := x^n]) \\ \bullet\ power(a, b)]\!]\end{array}$$

This program raises *a* to the power of *b*, using a procedure *power* with value-result parameter *x* and value parameter *n*. At this point we depart from Morgan's calculus and adopt Back's approach for the reasons reported in [7]. Nonetheless, we still support a calculational style based on the refinement laws in [6, 5].

The development of recursive procedures uses variant blocks $[\![\mathbf{proc}\ pname \mathrel{\widehat{=}} body\ \mathbf{variant}\ v\ \mathbf{is}\ e \bullet prog]\!]$ [5, 6]. Besides the procedure and the main program, we declare a variant expression *e* named *v*. It is used to guarantee termination in the development of a recursive implementation for *pname*.

Example As an example, we consider a program which, given two integers *a* and *b*, such that $a \geq 0$ and $b > 0$, sets *q* to the quotient of *a* divided by *b*, and sets *r* to the remainder of this division. The initial formal specification is as follows.

$$q, r : [a \geq 0 \wedge b > 0, a = q * b + r \wedge 0 \leq r < b]$$

We derive a program that initialises *q* and *r* with 0 and *a*, and then starts an iteration which runs while $r \geq b$. In each step, it increments *q* by one, and decrements *r* by *b*.

We start our refinement by splitting the specification into two: the first specifies the initialisation of *q* and *r*, and the second is later refined to an iteration. In the sequel, the symbol $\sqsubseteq$ represents the refinement relation. In each step of the refinement, we give the name of the law applied, and the arguments used in the application; the definition of the laws can be found in Appendix A.

$$\begin{array}{ll}\sqsubseteq seqComp(a = q * b + r \wedge 0 \leq r) & \\ q, r : [a \geq 0 \wedge b > 0, a = q * b + r \wedge 0 \leq r]; & \lhd \\ q, r : \left[\begin{array}{l} a = q * b + r \wedge 0 \leq r\ , \\ a = q * b + r \wedge 0 \leq r < b \end{array}\right] & (i)\end{array}$$

The *seqComp* law splits the specification with basis on an intermediate state definition given as argument; it is used as the postcondition of the first resulting specification, and the precondition of the second one. In this case, the intermediate state is characterised by the invariant of the iteration. At each step of the iteration, as *q* is incremented and *r* is decremented, the equality $a = q * b + r$ is maintained, and we never get a value for *r* below 0.

We introduce the initialisation of *q* and *r* with an application of the *assign* law to the first specification. The symbol $\lhd$ on the right indicates the specification to which the next law is applied. As a syntactic sugar, we use assignments as arguments of laws and tactics; for instance, we write **law** $assign(q, r := 0, a)$, but the arguments are actually q, r and $0, a$.

$$\begin{array}{l}\sqsubseteq assign(q, r := 0, a) \\ q, r := 0, a\end{array}$$

The predicate $a \geq 0 \wedge b > 0 \Rightarrow a = 0 * b + a \wedge 0 \leq a$ is generated as proof obligation; its proof is simple, since $0 * b = 0$ and $0 \leq a$ is in the antecedent of the implication. Most of the proof obligations in our examples are very simple; we omit them and their proofs for reasons of conciseness.

Next, we introduce the iteration. We use $r \geq b$ as guard and *r* as variant.

$$\begin{array}{ll}(\text{i}) \sqsubseteq iter(\langle r \geq b \rangle, r) & \\ \mathbf{do}\ \ r \geq b \rightarrow & \\ q, r : \left[\begin{array}{l} a = q * b + r \wedge 0 \leq r \wedge r \geq b\ , \\ a = q * b + r \wedge 0 \leq r \wedge 0 \leq r < r_0 \end{array}\right] & \lhd \\ \mathbf{od} & \end{array}$$

Finally, we introduce the assignment in the body of the iteration. We use the law *assignIV*, which applies to specifications with 0-subscribed variables.

$$\begin{array}{l}\sqsubseteq assignIV(q, r := q + 1, r - b) \\ q, r := q + 1, r - b\end{array}$$

In conclusion, by applying a sequence of refinement laws to the initial specification, we get the following executable program.

$$q, r := 0, a;\ \textbf{do}\ \ r \geq b \rightarrow q, r := q + 1, r - b\ \ \textbf{od}$$

In this simple example, we need four steps to obtain a program. This number is decreased when we use tactics, which is important for large developments.

3. ArcAngel

ArcAngel is a refinement-tactic language which can be used for documenting and analysing program developments and frequently used strategies of development. ArcAngel includes three different kinds of tactics: the simplest tactics are called basic tactics; the combinators of tactics are called tacticals; and tactics to handle parts of programs are called structural combinators.

3.1. Basic Tactics

A law application is expressed as **law** $n(a)$. The application of this tactic to a program may lead to two outcomes: if the law n with arguments a is applicable, then it is actually applied, otherwise the tactic fails. We also have **tactic** $n(a)$; its behaviour is similar, but it applies a tactic called n.

Another basic tactic is **skip**, which always succeeds, does not change the program, and also does not generate proof obligations. The tactic **fail** always fails; and the tactic **abort** neither succeeds nor fails, but runs indefinitely.

3.2. Tacticals

In ArcAngel, tactics t_1 and t_2 can be combined in sequence: $t_1;\ t_2$. This tactic 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, $t_1;\ t_2$ fails. When it succeeds, the proof obligations generated are those resulting from the application of t_1 and t_2.

We can also combine tactics in alternation: $t_1 \mid t_2$. 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. If one of the tactics aborts, the whole tactic aborts. Moreover, the first choice that leads to success is selected. This angelic nature of choice earned Angel and ArcAngel (A Refinement Calculus Angel) their names.

For instance, suppose we have a tactic t that always succeeds if the frame of the specification to which it is applied contains the variable x, and that we want to generalise it to t', which always succeeds. The generalisation t' could be defined as $(\,\textbf{skip} \mid \textbf{law}\ \mathit{expFrame}(\,x\,)\,);\ \ t$. This tactic first applies **skip**, and then attempts to apply t. If this application fails, a backtracking occurs, the law *expFrame* is applied to insert x in the frame, and the tactic attempts again to apply t. This application now succeeds, since the variable x was inserted in the frame. This successfully finishes the application of the whole tactic t'.

The backtracking in the implementation of angelic nondeterminism may lead to inefficient searches. ArcAngel gives to the programmer some control through the cut operator (!). The tactic $!\,t$ behaves like t, except that it returns the first successful application of t. If a subsequent tactic application fails, then the whole tactic fails. The application of $!(\,\textbf{skip} \mid \textbf{law}\ \mathit{expFrame}(\,x\,)\,);\ \ t$ to a specification in which x is not in the frame, for instance, fails since, after applying the first choice of the alternation, **skip**, no backtracking is possible.

ArcAngel has a fixed-point operator that allows us to define recursive tactics. Using this operator, we can, for instance, define a tactic that applies another tactic t exhaustively: $\mu X \bullet (\,t;\ X \mid \textbf{skip}\,)$. This tactic applies t as many times as possible, terminating with success when the application of t fails.

In the tactic **applies to** p **do** t, a meta-program p is introduced to characterise the programs to which this tactic is applicable; the meta-variables used in p can then be used in t. By way of illustration, we have that the meta-program $w : [\mathit{pre}_1 \wedge \mathit{pre}_2, \mathit{post}\,]$ characterises those specifications whose precondition is a conjunction; here, pre_1, pre_2, and *post* are the meta-variables. The commonly used refinement strategy of weakening a precondition by dropping a conjunct can be formalised by the tactic below.

$$\textbf{applies to}\ \ w : [\mathit{pr}_1 \wedge \mathit{pr}_2, \mathit{pt}]\ \textbf{do law}\ \mathit{weakPre}(\mathit{pr}_1)$$

ArcAngel defines two tactics that are used to make tactic assertions. The tactic **succs** t fails whenever t fails, and behaves like **skip** whenever t succeeds. On the other hand, **fails** t behaves like **skip** if t fails, and fails if t succeeds. If the application of t runs indefinitely, then these tacticals behave like **abort**.

3.3. Structural Combinators

Usually, it is desirable to apply tactics to subprograms. For sequential composition, ArcAngel defines the tactic $t_1 \boxed{;} t_2$, which 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 are those arising from both tactic applications. The combinators like $\boxed{;}$ are called *structural combinators*. There is one combinator for each syntactic construct.

The tactic $\boxed{\textbf{if}}\, t_1 \boxed{[\!]} \ldots \boxed{[\!]}\, t_n \boxed{\textbf{fi}}$ applies to an alternation in the form **if** $g_1 \rightarrow p_1\ [\!]\ \ldots\ [\!]\ g_n \rightarrow p_n$ **fi** and returns the result of applying each tactic t_i to the corresponding program p_i. A similar constructor is available for iterations: $\boxed{\textbf{do}}\, t_1 \boxed{[\!]} \ldots \boxed{[\!]}\, t_n \boxed{\textbf{od}}$.

For variable blocks, ArcAngel defines the structural combinator $\boxed{\textbf{var}}\, t \boxed{]\!]}$; similarly, the structural combinator $\boxed{\textbf{con}}\, t \boxed{]\!]}$ applies to logical constant blocks. Each applies its tactic t to the block's body.

The combinators $\boxed{\textbf{pmain}}\, t \boxed{]\!]}$ and $\boxed{\textbf{pmainvariant}}\, t \boxed{]\!]}$ are used in the case of procedure and variant blocks; they apply t to the main program of the blocks. In the case of applying a tactic to a procedure body, we use the combinators $\boxed{\textbf{pbody}}\, t \boxed{]\!]}$ and $\boxed{\textbf{pbodyvariant}}\, t \boxed{]\!]}$. The application of tactics to a procedure body and to the main program of a procedure block or of a variant block, at the same time is also possible. We use the structural combinators $\boxed{\textbf{pbodymain}}\, t_b\ ,\ t_m \boxed{]\!]}$ and $\boxed{\textbf{pmainvariantbody}}\, t_b\ ,\ t_m \boxed{]\!]}$. They apply t_b to the procedure body, and t_m to the main program.

For parameterised commands, ArcAngel defines the tactics $\boxed{\textbf{val}}\, t$, $\boxed{\textbf{res}}\, t$, and $\boxed{\textbf{val-res}}\, t$. Consider, for instance, the application of $\boxed{\textbf{pbody}}\boxed{\textbf{val-res}}\ \textbf{law}\ assignIV(x := x^n) \boxed{]\!]}$ to the procedure block below.

$$[\![\ \textbf{proc}\ power \mathrel{\widehat=} (\textbf{val-res}\ x : \mathbb{N} \bullet x : [x = x_0^n\,]) \\ \bullet\ x : [x = x_0^n\,]\ ]\!].$$

We get the following program as result.

$$[\![\ \textbf{proc}\ power \mathrel{\widehat=} (\textbf{val-res}\ x : \mathbb{N} \bullet x := x^n) \\ \bullet\ x : [x = x_0^n\,]\ ]\!].$$

We also get $x = x_0 \wedge true \Rightarrow x^n = x_0^n$ as proof obligation. If we are not concerned with the type of argument declaration, we use $\boxed{\textbf{parcommand}}\, t$.

A tactic program consists of a sequence of tactic declarations followed by a main tactic that usually makes use of the declared tactics. A tactic declaration takes the form **Tactic** $n(a)\ t$ **end**. The result of applying **Tactic** $n(a)\ t$ **end** is that of applying t, which is named n and uses the arguments a. For documentation purposes, we may include in the declaration the clauses **proof obligations** and **program generated**; the former lists the proof obligations generated by the application of t, and the latter shows the program generated.

3.4. Example

The tactic *takeConjAsInv* declared below aims at the development of an initialised iteration [17, 15]. It applies to specifications in the form $w : [pre, invConj \wedge \neg\ guard]$ and takes *invConj* as part of the iteration invariant.

This tactic has three arguments: a predicate *invBound* that gives limits for indexing variables of the iteration; the initialisation $lstVar := lstVal$ of the iteration variables; and the variant of the iteration *variantExp*. It introduces an initialised iteration with invariant $invBound \wedge invConj$, and variant *variantExp*.

```
Tactic $takeConjAsInv$
$(\,invBound, (lstVar := lstVal), variantExp\,)$
applies to $w : [\,pre, invConj \wedge \neg\ guard\,]$ do
    law $strPost(\,invBound \wedge invConj \wedge \neg\ guard\,)$;
    law $seqComp(\,invBound \wedge invConj\,)$;
    (law $assign(\,lstVar := lstVal\,)$ $\boxed{;}$
        law $iter(\langle guard\rangle, variantExp)$) end
```

Firstly, *takeConjAsInv* strengthens the postcondition (law *strPost*) of the initial specification, adding *invBound* as a conjunct. Then, it introduces a sequence (law *seqComp*); the invariant defines the intermediate state. Finally, it applies the law *assign* to the first program of the composition to derive the initialisation, and the law *iter* to the second program to introduce the iteration.

In Section 2, we refined the specification below.

$$q, r : \left[\begin{array}{l} a \geq 0 \wedge b > 0\ , \\ (a = q * b + r \wedge 0 \leq r) \wedge \neg\ r \geq b \end{array}\right] \qquad \lhd$$

We can apply the tactic *takeConjAsInv* in this development. We use the arguments *true* as the range limit of the index variable b (*invBound*), $q, r\ :=\ 0, a$ as the initialisation ($lstVar := lstVal$), and r as the variant (*variantExp*).

$$\begin{array}{l} \sqsubseteq takeConjAsInv(true, (q, r := 0, a), r) \\ q, r := 0, a; \\ \textbf{do}\ \ r \geq b \rightarrow \\ q, r : \left[\begin{array}{l} a = q * b + r \wedge 0 \leq r \wedge r \geq b\ , \\ a = q * b + r \wedge 0 \leq r \wedge 0 \leq r < r_0 \end{array}\right] \qquad \lhd \\ \textbf{od} \end{array}$$

The proof obligations generated are the same as in Section 2, as is the last step of the development: to introduce the assignment in the body of the iteration. We get the same code in two steps.

Further examples can be found in [19].

4. Refine

Refine is a tool that supports program development based on the refinement calculus. Its interface is composed of a menu and four windows (see Figure 1): the refinement window, which presents the program development; the laws window, which lists the refinement laws; the proof obligations window, which lists the proof obligations generated in the program development; and the code window, which presents the currently developed program.

To illustrate the use of Refine, we consider again our example in Section 2. To start a new program development in Refine, the user must press the start new development button or select this option in the refinement menu. As a re-

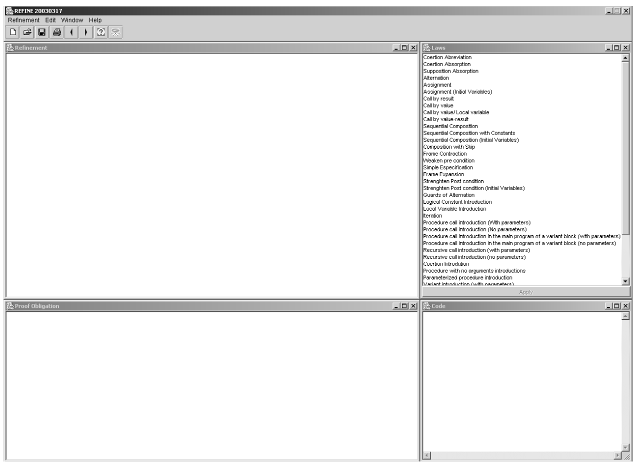

REFINE 20030317
Refinement Edit Window Help
Refinement
Laws
Coertion Abreviation
Coertion Absorption
Supposition Absorption
Alternation
Assignment
Assignment (Initial Variables)
Call by result
Call by value
Call by value/ Local variable
Call by value-result
Sequential Composition
Sequential Composition with Constants
Sequential Composition (Initial Variables)
Composition with Skip
Frame Contraction
Weaken pre condition
Simple Especification
Frame Expansion
Strenghten Post condition
Strenghten Post condition (Initial Variables)
Guards of Alternation
Logical Constant Introduction
Local Variable Introduction
Iteration
Procedure call introduction (With parameters)
Procedure call introduction (No parameters)
Procedure call introduction in the main program of a variant block (with parameters)
Procedure call introduction in the main program of a variant block (no parameters)
Recursive call introduction (with parameters)
Recursive call introduction (no parameters)
Coertion Introdution
Procedure with no arguments introductions
Parameterized procedure introduction
Apply
Proof Obligation
Code

Figure 1. Refine's user interface

sult, the user is asked to type a specification to be refined. In our example, the initial specification is as follows.

```
q,r:[a>=0 & b>0,
     a=q*b+r & 0<=r & not r>=b]
```

We use an ASCII notation for relational and predicate calculus symbols; in our example, `>=` for $\geq$, `&` for $\wedge$, and `not` for $\neg$. The predicate language supported includes all operators in [17]. Moreover, a symbol keyboard is provided to help the user to introduce these operators. The starting program may be a specification, as in our example, or any valid program of the refinement calculus language.

To apply a refinement law, we select the part of the program we want to refine by clicking the left-button of the mouse on it. Afterwards, we select the law we want to apply by clicking the left-button of the mouse on its name in the laws window, and press the apply button. As opposed to many tools described in the literature, especially those based on theorem provers, the interface is completely interactive. Virtually, no extra knowledge is required for its use, except the refinement calculus itself. This is very important for an educational tool.

In our example, the first step is the sequential composition introduction. We select the initial specification in the development window, and the sequential composition law in the laws window. The user is asked to type the argument. After the application of the law, the development, proof obligations, and code windows are refreshed.

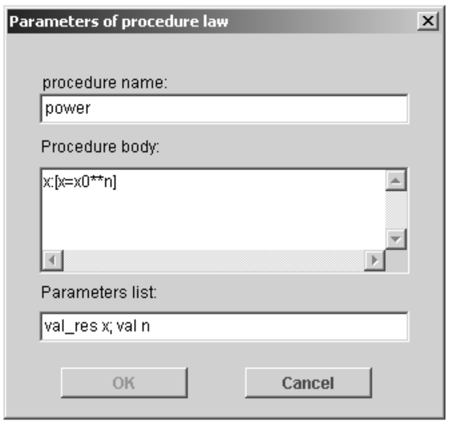
If some error occurs in the law application, an error window is displayed describing the reason. For instance, we cannot apply the sequential composition law to anything other than a specification statement; this is checked by Refine. Moreover, Refine checks all the syntactic restrictions associated with the laws. For example, it checks that the argument given in the application of the sequential composition law does not include 0-subscripted variables.

Since Refine was first presented in [8], a lot of work has been done on it. We introduced facilities for development management, and, even more important, support to the development of procedures. The following example presents a development that introduces a procedure. We also illustrate the development management features of Refine.

We consider, by way of illustration, a program that raises a to its b power. After we start a new development, we introduce `a:[a=a0**b]` as the initial specification. The predicate `a=a0**b` is the ASCII representation for $a = a_0^b$. We may easily develop this program by applying the law that introduces an assignment from a specification with initial variables with argument `a:=a**b`. As result of this application, we get this assignment as final program, and the proof obligation `(a=a0) & true => a**b = a0**b`.

Nevertheless, we may refine this program using the procedure `power` presented in Section 2. In Refine, we can undo previous law applications by pressing the undo button. This undoes the last law application in the development window, removes the proof obligations generated by it, and refreshes the collected code window. In contrast, the redo operation redoes the last undone application. In our example, by pressing the undo button, we return to the situation in which we have the initial specification in the development and the collected code windows, and to an empty proof obligation window.

We restart our refinement by introducing the procedure `power` using the parameterised procedure introduction law. The arguments used for this application are presented in Figure 2: the name and the body of the procedure, and its parameters. With this application, we get the following pro-


Parameters of procedure law

procedure name: power

Procedure body: x:[x=x0**n]

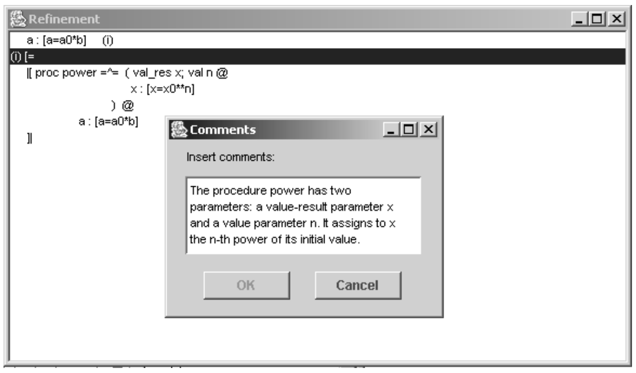
Parameters list: val_res x; val n

Figure 2. Parameterised procedure introduction arguments

cedure block.

```
|[ proc power =^= (val_res x; val n @
        x:[x=x0**n]) @ a:[a=a0**b] ]|
```

We may add comments to any part of the development by clicking with the right-button of the mouse on any line of the development window, and then selecting the insert comments option. With this, a window pops up in which we can insert the comment. In our example, we add a comment to the procedure to explain its functionality (see Figure 3). Another option available with the click of the right-button of the mouse in the development window is the visualisation of a previously inserted comment. With the use of comments, we can record and document important decisions and make the development more readable.



```
    a : [a=a0*b]    (i)
(i) [=
    |[ proc power =^= ( val_res x; val n @
                          x : [x=x0**n]
                      ) @
              a : [a=a0*b]
    ]|
```

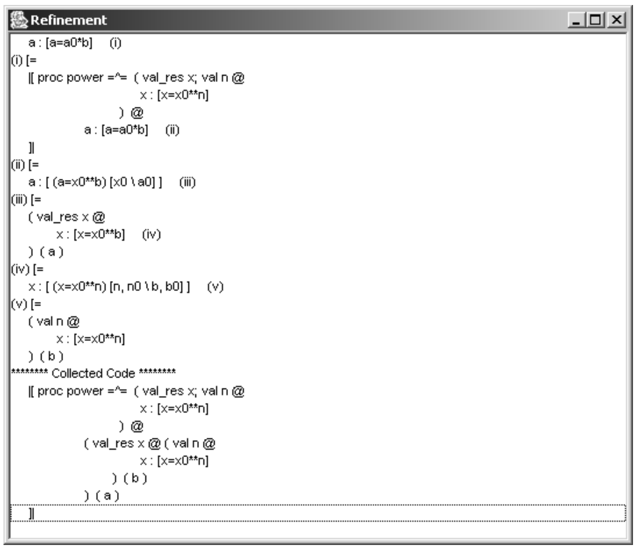
The procedure power has two parameters: a value-result parameter x and a value parameter n. It assigns to x the n-th power of its initial value.

Figure 3. Adding comments

We start the refinement of the main program by strengthening its postcondition, using the predicate `(a=x0**b)[x0\a0]` as argument. The resulting specification matches the pattern required by the call by value-result law, which we apply next. We get `(val_res x @x:[x=x0**b])(a)` as result.

We refine the specification in the parameterised command by strengthening the postcondition, using the predicate `(x=x0**n)[n,n0\b,b0]` as argument, in order to get a specification which matches the pattern required by the call by value law. With this law, we get `(val n @ x:[x=x0**n])(b)` as result.

The next step is to collect the code in the development window, by clicking on the right-button of the mouse and choosing the collect code option. This step is necessary because we want to apply a law to the outer parameterised command, which is not shown in the development window in its current form. This gives us the code presented in Figure 4. We select the whole main program of the procedure block by clicking on all its lines. Then, we apply the multiple parameters law and get the main program



```
    a : [a=a0*b]    (i)
(i) [=
    |[ proc power =^= ( val_res x; val n @
                          x : [x=x0**n]
                      ) @
              a : [a=a0*b]    (ii)
    ]|
(ii) [=
    a : [ (a=x0**b) [x0 \ a0] ]    (iii)
(iii) [=
    ( val_res x @
          x : [x=x0**b]    (iv)
    ) ( a )
(iv) [=
    x : [ (x=x0**n) [n, n0 \ b, b0] ]    (v)
(v) [=
    ( val n @
          x : [x=x0**n]
    ) ( b )
******** Collected Code ********
    |[ proc power =^= ( val_res x; val n @
                          x : [x=x0**n]
                      ) @
              ( val_res x @ ( val n @
                          x : [x=x0**n]
                    ) ( b )
              ) ( a )
    ]|
```

Figure 4. Collecting code

`(val_res x ; val n@x:[x=x0**n])(a,b)`. The parameterised statement we obtain is the same as that in the body of `power`; we are ready to introduce a procedure call.

Next, we collect again the code in order to get the whole procedure block together. We select it and apply the parameterised procedure call introduction law. This gives us the following procedure block.

```
|[ proc power=^=(val_res x; val n @
        x:[x=x0**n]) @ power(a,b) ]|
```

Finally, we refine the body of the procedure `power` to `x:=x**n`. The resulting program is as follows.

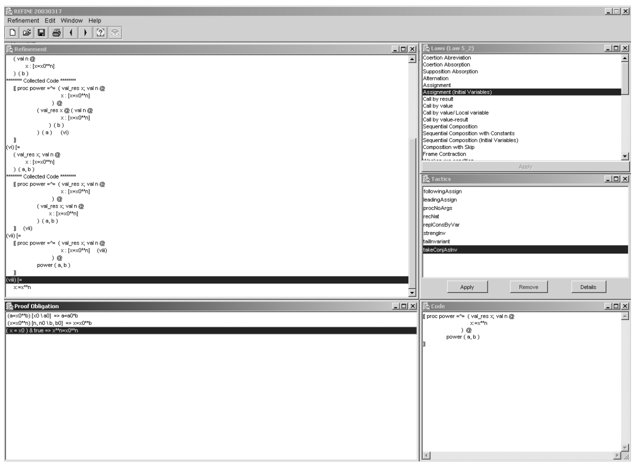
```
|[ proc power=^=(val_res x; val n @
        x:=x**n) @ power(a,b) ]|
```

Only three proof obligations are generated in the whole development (see Figure 5). If we click on any of them, the law application that generated it is highlighted.

At any time of the development the user can save the current development in order to edit it later. Besides, the user can print all the information of a development. This consists of the development itself, proof obligations, collected code, and comments, or a combination of them.

5. Gabriel

Gabriel brings to the users of Refine the possibility of dealing with tactics. It works as a plug-in and adds one window to Refine: the tactics window, which lists the tactics available for application in program developments. Most of the tactics presented in [21, 19], which covers the vast



```
  ( val n @
       x : [x=x0**n]
  ) ( b )
******** Collected Code ********
  [[ proc power =^=  ( val_res x; val n @
                          x : [x=x0**n]
                      ) @
              ( val_res x @ ( val n @
                          x : [x=x0**n]
                     ) ( b )
              ) ( a )     (vi)
  ]]
(vi) [=
  ( val_res x; val n @
       x : [x=x0**n]
  ) ( a, b )
******** Collected Code ********
  [[ proc power =^=  ( val_res x; val n @
                          x : [x=x0**n]
                      ) @
              ( val_res x; val n @
                     x : [x=x0**n]
              ) ( a, b )
  ]]    (vii)
(vii) [=
  [[ proc power =^=  ( val_res x; val n @
                          x : [x=x0**n]    (viii)
                      ) @
              power ( a, b )
  ]]
(viii) [=
  x:=x**n
```

```
Coertion Abreviation
Coertion Absorption
Supposition Absorption
Alternation
Assignment
Assignment (Initial Variables)
Call by result
Call by value
Call by value/ Local variable
Call by value-result
Sequential Composition
Sequential Composition with Constants
Sequential Composition (Initial Variables)
Composition with Skip
Frame Contraction
```

```
followingAssign
leadingAssign
procNoArgs
recNat
replConsByVar
strengInv
tailInvariant
takeConjAsInv
```

```
(a=x0**b) [x0 \ a0]  => a=a0*b
(x=x0**n) [n, n0 \ b, b0]  => x=x0**b
( x = x0 ) & true => x**n=x0**n
```

```
[[ proc power =^=  ( val_res x; val n @
                        x:=x**n
                    ) @
           power ( a, b )
]]
```

Figure 5. Proof obligations and law applications

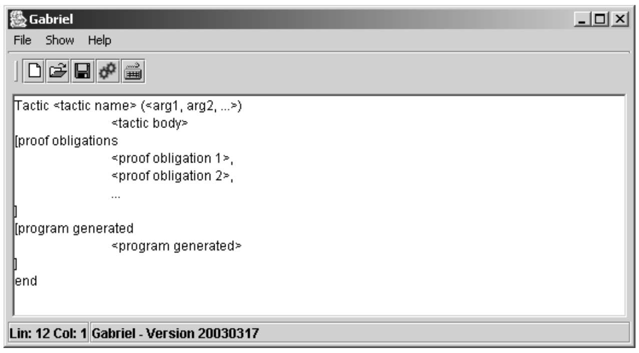
majority of those in the literature, are initially available in Gabriel; however, the user can add his own tactics.

Gabriel is activated from Refine by pressing the Gabriel's button in Refine's menu, or by selecting the Gabriel option in the window menu of Refine. The following operations are available in Gabriel: create a tactic using ArcAngel; edit a tactic; generate a tactic, which verifies syntactically the tactic, and inserts the tactic in the tactics window; and remove a tactic. After its generation, a tactic can be applied in program developments of Refine.

Gabriel has a simple user interface, which is presented in Figure 6. The buttons can be used to: start a new tactic; open an existing tactic; save a tactic; generate a tactic and insert it into the tactics list of Refine; and open a symbols keyboard, which is used to insert the ASCII version of ArcAngel.

Gabriel's user interface was projected and tested using LUCID [18], the User-Centered Interface Design [14] technique. First, we made an action-object analysis, which consists of building a directed-graph describing the activities involved in program refinement, and building an object-tree containing the objects of Refine and Gabriel's user interface. Using this material, we analysed the correspondence of program refinement activities and the user-interface of Refine and Gabriel. Finally, we tested Refine and Gabriel's usability. This test was made with potential users of the tool and consisted of creating and using a tactic using Gabriel and Refine. After using the tool, the users were interviewed. Positive points and difficulties were raised in an interview.

The positive aspects pointed were the easiness of open-



```
Tactic <tactic name> (<arg1, arg2, ...>)
                  <tactic body>
[proof obligations
                  <proof obligation 1>,
                  <proof obligation 2>,
                  ...
]
[program generated
                  <program generated>
]
end
```

Figure 6. Gabriel's user interface

ing Gabriel, seeing the result of a tactic creation, and applying laws and tactics, and the nice integration with Refine. The suggestions made were the inclusion of a symbol keyboard, of ArcAngel documentation in the help, of a generate tactic button, a show tactic/law details facility for the tactic/law list, and of a tactic template to start new tactics. These facilities were incorporated to Refine and Gabriel.

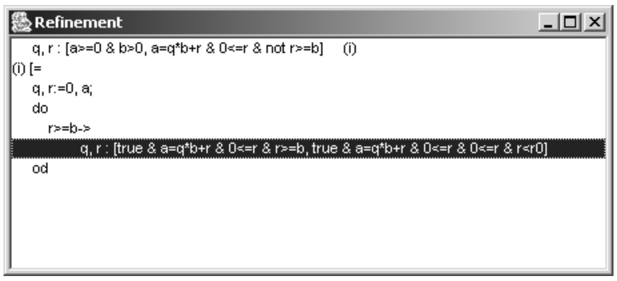
Gabriel supports an ASCII version of ArcAngel. Table 1 presents the ASCII version of some ArcAngel's constructs. During a tactic generation, Gabriel verifies if the laws used in its definition are supported by Refine. In Gabriel the arguments of the laws are typed. The existing types, and examples of argument declarations are presented in [19].

ArcAngel	Gabriel
$\boxed{;}$	|; |
$\boxed{\textbf{if}}\ \boxed{[]}\ \boxed{\textbf{fi}}$	| *if* | | [] | | *fi* |
$\boxed{\textbf{var}}\ \boxed{]]}$	| *var* | |] ||

Table 1. ArcAngel's ASCII derivation

We consider our refinement example presented in Section 3. After starting a new development and entering our new specification, as before, we select the initial specification and the tactic *takeConjAsInv*. When the apply button in the tactic window is pressed, Gabriel requires the arguments of the tactic application. The values inserted in our example are `true` for *invBound*; `q,r` for *lstVar*; `0,a` for *lstVal*; and `r` for *variantExp*.

Gabriel applies the tactics after the insertion of the last argument value. In the application of the **applies to** *p* **do** *t* constructor, an unification algorithm is used; in our example, the meta-variables *w*, *pre*, *invConj*, and *guard* (see the definition of *takeConjAsInv* in Section 3.4) are unified with `q,r`, `a>=0 & b>0`, `a=q*b+r & 0<=r`, and `r>=b`, respectively. The program development window, the proof obligations window and the collected code window are refreshed. The resulting development window of our example is presented in Figure 7. The last step of our refinement is



```
    q, r : [a>=0 & b>0, a=q*b+r & 0<=r & not r>=b]     (i)
(i) [=
    q, r:=0, a;
    do
       r>=b->
            q, r : [true & a=q*b+r & 0<=r & r>=b, true & a=q*b+r & 0<=r & 0<=r & r<r0]
    od
```

Figure 7. Application of tactic *takeConjAsInv*

the assignment introduction in the iteration. It is carried out using only the facilities of Refine, as explained in the previous section.

In order to use a new tactic, the user must define it using Gabriel and insert it into Refine's tactics list. As case studies on the use of Refine and Gabriel, we have developed all B programming examples presented in [1]. One of them considers a development strategy for programs that calculate the value of $f(n)$, where f is a function on natural numbers that is recursively defined in terms of another function g as shown below, and n is a natural number.

$$\begin{array}{l} f(0) = a \\ f(n+1) = g(f(n)) \end{array} \tag{1}$$

Using ArcAngel, we were able to capture this strategy as the refinement tactic *recNat* presented in Figure 8. This tactic has two arguments: the value `a` of the function f at 0, and the function `g`. It applies to specifications `r:[n>=0,r=f(n)]`, which assign to `r` the value of `f(n)`. The result is the program

```
|[ var k @ r,k := a,0 ;
    do k<n -> r,k := g(r),k+1 od ]|.
```

It initialises `r` to `a`, and iterates `n` times applying `g` to `r`. First, *recNat* introduces the variable `k`. Then, it strengthens the postcondition, relating the variable `k` to the old postcondition; the result is a specification that matches the pattern required by the tactic *takeConjAsInv*, which is invoked in sequence. Finally, *recNat* refines the body of the iteration.

To define this tactic, the user must open Gabriel from Refine by pressing the Gabriel's button. A tactics template is presented, which can be edited to get the tactic in Figure 8. Afterwards, the user must save the text by pressing the save button in Gabriel, choosing a name of a file, and pressing the OK button. Now, the user can generate the tactic by pressing the generate button in Gabriel or choosing the generate option in the file menu of Gabriel. The success of this generation depends on the tactic being syntactically correct and referring only to valid laws. Success leads to the insertion of the tactic in the list of Refine. However, if the generation fails, Gabriel shows an error message indicating the line of the error, the invalid construct which originated the error, and a detailed parser message.

```
Tactic recNat (EXP(a),EXP(g))
  applies to r:[n>=0,r=f(n)]
  do
    law varInt(DECS(k));
    |var|
       law strPost(PRED((0<=k & k<=n &
                         r=f(k)) &
                         not k<n));
       tactic takeConjAsInv(PRED(true),
                            IDS(r,k),
                            EXPS(a,0),
                            EXP(n-k));
       (skip |;| |do|
                    law assign(IDS(r,k),
                               EXPS(g(r),
                                    k+1))
                  |od|)
    |]||
  program generated
    |[ var k @ r,k := a,0 ;
       do k<n -> r,k := g(r),k+1 od ]|
end
```

Figure 8. Tactic *recNat*

As an example of using the tactic *recNat*, let us consider the natural number exponentiation function $exp(x)(y)$, which can be recursively defined as

$$\begin{array}{l} exp(x)(0) = 1 \\ exp(x)(y+1) = mult(x)(exp(x)(y)). \end{array} \tag{2}$$

Using Refine, we can start the development of the program that assigns to r the expression $exp(x)(y)$ with the specification `r:[y>=0,r=exp_x(y)]`. Since Refine does not support expressions of the form `f(x)(y)`, we use the notation `f_x(y)` to represent such expressions. By applying *recNat* with arguments `1` and `mult_x`, we get the following program.

```
|[ var k @ r,k := 1,0 ;
    do k<y -> r,k := mult_x(r),k+1 od ]|
```

The refinement is accomplished in just one step.

6. Conclusions

In this paper, we presented tools to support the use of the refinement calculus and ArcAngel. This is a refinement-tactic language, that can be used to describe commonly used program development strategies. Using these tactics as transformation rules shortens developments and improve their readability. The literature presents some other tactic languages [10, 25, 26, 23, 24, 11, 12, 3, 27, 28, 4, 13]. However, as far as we know, apart from Angel, which we extend and adapt, none of them has a formal semantics and reasoning laws.

Our tool, Gabriel, extends Refine which we developed to support the application of the refinement calculus. Refine has been successfully used as an educational tool for more than three years. Gabriel adds the facility to define and apply tactics of development. Together, Refine and Gabriel provide powerful tool support for program developments using the refinement calculus and ArcAngel.

Both Refine and Gabriel are available from [20]. They were developed using Java, and amount to sixty thousand lines of code. In the site, we can also find UML documentation of their design, a tutorial, and example developments.

Several existing tools provide support for the use of the refinement calculus and tactics. Some [10, 11, 12, 4] use languages like Prolog for defining tactics and, in this case, the user needs to learn complex languages to achieve his goals. In [13] a goal-oriented approach is adopted: refining consists of proving that the final program implements the initial specification. Since we do not know the final program from the beginning of the refinement, this does not seem convenient. The Proxac system [23, 24] does not define any language for tactic definition. Furthermore, as far as we know, it is not possible to deal with procedure and variant blocks using any of the existing refinement tools [3, 27, 28].

Refine has already proved to be very useful as an educational tool; Gabriel has proved to be very promising: we have used both to refine many programming examples, including most of the programming examples presented in [1, chap.10]. We believe that, together, they are an added motivation for the application of the refinement calculus. There is, however, more work to be done.

With the aim of building a user-friendly tool, whose use is as straightforward as possible, we decided to build Refine (and Gabriel) from scratch. We did not rely on an existing theorem prover. As a drawback, Refine and Gabriel do not provide support for the discharge of the proof obligations. We do not plan to replicate work by implementing yet another theorem prover; we plan to integrate them to a suitable existing one.

Another substantial piece of future work is the support of higher-order tactics; this requires a revision of the semantics and laws of ArcAngel. Finally, as already mentioned, ArcAngel has an extensive suite of transformation laws that can be used to reason about tactics. An interesting extension to Gabriel is the support to the application of these laws. In this way, we can develop and transform tactics of development, in much the same way as we develop programs. At the moment, however, we are concentrating on case studies. We expect them to reveal any extensions to ArcAngel and Gabriel that are important in practice, which we will then address before moving to the more adventurous work.

Acknowledgements

The work of Marcel Oliveira was partially supported by the Computing Laboratory of the University of Kent. The work of Ana Cavalcanti is partially supported by CNPq: grants 520763/98-0 and 472204/01-7. We are grateful to Jim Woodcock for his valuable collaboration in the work on ArcAngel.

References

[1] J.-R. Abrial. *The B-book: assigning programs to meanings*. Cambridge University Press, 1996.

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

[3] R. J. R. Back and J. von Wright. Refinement Concepts Formalised in Higher Order Logic. *Formal Aspects of Computing*, 2:247–274, 1990.

[4] D. Carrington, I. Hayes, R. Nickson, G. Watson, and J. Welsh. A program refinement tool. *Formal Aspects of Computing*, 10(2):97–124, 1998.

[5] A. L. C. Cavalcanti. *A Refinement Calculus for Z*. PhD thesis, Oxford University Computing Laboratory, Oxford - UK, 1997. Technical Monograph TM-PRG-123, ISBN 00902928-97-X.

[6] A. L. C. Cavalcanti, A. C. A. Sampaio, and J. C. P. Woodcock. Procedures and Recursion in the Refinement Calculus. *Accepted for publication in Journal of the Brazilian Computer Society*, 1998.

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

[8] 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, Florianópolis - SC, 1999. Sessão de Ferramentas.

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

[10] 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.

[11] Lindsay Groves. Adapting formal derivations. Technical Report 1995.CS-TR-95-9, 1995.

[12] Lindsay Groves. Deriving programs by combining and adapting refinement scripts. Technical Report 1995.CS-TR-95-13, 1995.

[13] J. Grundy. A Window Inference Tool for Refinement. In C. B. Jones, R. C. Shaw, and T. Denvir, editors, *5th Refinement Workshop*, Workshops in Computing, pages 230 – 254. Springer-Verlag, 1992.

[14] Inc. John Wiley & Sons, editor. *The Elements of User Interface Design*. Springer Verlag, 1997.

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

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

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

[18] M. V. M. Oliveira. Teste de Usabilidade de REFINE e T - REFINE. Technical report, Centro de Informática - Universidade Federal de Pernambuco, Pernambuco - Brazil, December 2001. At http://www.cs.kent.ac.uk/˜mvmo2/gabriel/.

[19] M. V. M. Oliveira. ArcAngel: a Tactic Language for Refinement and its Tool Support. Master's thesis, Centro de Informática - Universidade Federal de Pernambuco, Pernambuco - Brazil, 2002. At http://www.ufpe.br/sib/.

[20] M. V. M. Oliveira. *Refine-Gabriel Project Page*, 2002. At http://www.cs.kent.ac.uk/˜mvmo2/gabriel/.

[21] M. V. M. Oliveira and A. L. C. Cavalcanti. Tactics of Refinement. In *XIV Simpósio Brasileiro de Engenharia de Software*, pages 117 – 132, 2000.

[22] M. V. M. Oliveira, A. L. C. Cavalcanti, and J. C. P. Woodcock. ArcAngel: a Tactic Language for Refinement. *Formal Aspects of Computing*, 15(1):28 – 47, 2003.

[23] Jan L. A. van de Snepscheut. Proxac: An editor for program transformation. Technical Report 1993.cs-tr-93-33, 1993.

[24] Jan L. A. van de Snepscheut. Mechanised support for stepwise refinement. In Jürg Gutknecht, editor, *Programming Languages and System Archtectures*, volume 782 of *Lecture Notes in Computer Science*, pages 35–48. Springer, March 1994. Zurich, Switzerland.

[25] T. Vickers. An Overview of a Refinement Editor. In *5th Australian Software Engineering Conference*, pages 39–44, Sydney - Australia, May 1990.

[26] T. Vickers. A language of refinements. Technical Report TR-CS-94-05, Computer Science Department, Australian National University, 1994.

[27] J. von Wright. Program Refinement by Theorem Prover. In D. Till, editor, *6th Refinement Workshop*, Workshops in Computing, pages 121 – 150, London - UK, 1994. Springer-Verlag.

[28] J. von Wright, J. Hekanaho, P. Luostarinen, and T. Långbacka. Mechanizing Some Advanced Refinement Concepts. *Formal Methods in System Design*, 3:49–81, 1993.

A. Refinement Laws

Law $\mathit{strPost}(pt_2)$. $w : [pr, pt_1] \sqsubseteq w : [pr, pt_2]$,
provided $pt_2 \Rightarrow pt_1$

Law $\mathit{weakPre}(pr_2)$. $w : [pr_1, pt] \sqsubseteq w : [pr_2, pt]$,
provided $pr_1 \Rightarrow pr_2$

Law $\mathit{assign}(w := E)$. $w : [pr, pt] \sqsubseteq w := E$,
provided $pr \Rightarrow pt[w \setminus E]$

Law $\mathit{seqComp}(mid)$.
$w : [pr, pt] \sqsubseteq w : [pr, mid];\ w : [mid, pt]$,
provided *mid* and *pt* have no free initial variables.

Law $\mathit{assignIV}(w := E)$. $w, x : [pr, pt] \sqsubseteq w := E$,
provided $(w = w_0) \land pre \Rightarrow post[w \setminus E]$.

Law $\mathit{iter}(\langle G_1, ..., G_n \rangle, V)$. Let *inv*, the invariant, be any formula; let V, the variant, be any integer-valued expression. Then, if GG is the disjunction of the guards,
$w : [inv, inv \land \neg\ GG]$
$\sqsubseteq$
do $(?\ i.G_i \rightarrow$
$w : [inv \land G_i, inv \land 0 \leq V \leq V[w \setminus w_0]])$ **od**
inv and G_i may not contain initial variables.

Law $\mathit{expFrame}(x)$.
$w : [pr, pt] \sqsubseteq w, x : [pr, pt \land x = x_0]$.

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

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

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

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

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