Teaching programming methodology using Event B

Dominique Méry

Teaching programming methodology using Event B

Dominique Méry[1,2]

LORIA
Université Henri Poincaré Nancy 1
Vandoeuvre lès Nancy,France

Abstract

Event B is supported by the RODIN platform and provides a framework for teaching programming methodology based on the famous pre/post specifications, together with the refinement. We illustrate a methodology based on Event B and the refinement by developing Floyd's algorithm for computing the shortest distances of a graph, which is based on an algorithm design technique called dynamic programming. The development is based on a paradigm identifying a non-deterministic event with a procedure call and by introducing control states. We discuss points related to our lectures at the university.

Keywords: Event B, refinement, seuential algorithm, pattern, proof, teaching, formal method, recursive procedure, development, correct by construction

1 Foreword

It is a great pleasure to thank Henri Habrias for his lectures on B at the University of Nantes. He understood the rôle of mathematics in the curriculum of computer scientists and the impact of the B methodology in industry and in education. Moreover, he was not only teaching notations but concepts that the young computer scientist, who has attended his lectures, will understand when the maturity will be there. He helps our community to propagate *the two mammels of computer science*, namely abstraction and refinement and is becoming now the King Henri. He was a solid support to B. It is a very modest exercise to discuss with Jean-Raymond, You and colleagues of our meeting in Nantes. Thanks Henri!

2 Introduction

Overview. Event B is supported by the RODIN platform and provides a framework for teaching programming methodology based on the famous pre/post specifications, together with the refinement. We illustrate a methodology based on Event B and the refinement by

[1] Email: mery@loria.fr

[2] Work of Dominique Méry is supported by grant No. ANR-06-SETI-015-03 awarded by the Agence Nationale de la Recherche.

developing algorithms for computing the shortest distances of a graph, which is based on an algorithm design technique called dynamic programming. Floyd's algorithm is redeveloped and we add comments on the complexity of proofs and on the discovery of invariant; it should be considered as an illustration of a technique introduced in a joint paper with D. Cansell[7]. The development is based on a paradigm identifying a non-deterministic event with a procedure call and by introducing control states. We discuss points related to our lectures at different levels of the university. It is also a way to introduce a pattern used for developing sequential structured programs.

Progamming methodology. The development of structured programs is carried out either using bottom-up techniques, or top-down techniques; we show how refinement and proof can be used to help in the top-down development of structured imperative programs. When a problem is stated, the incremental proof-based methodology of event B[6] starts by stating a very abstract model and further refinements lead to finer-grain event-based models which are used to derive an algorithm[3]. The main idea is to consider each *procedure call* as an *abstract event* of a model corresponding to the development of the *procedure*; generally, a procedure is specified by a pre/post specification and then the refinement process leads to a set of events, which are finally combined to obtain the *body of the procedure*. The refinement process can be considered as an *unfolding* of *calls* statements under preservation of invariants. It means that the abstraction corresponds to the procedure call and the body is derived using the refinement process. The refinement process may also use recursive procedures and supports the top-down refinement. The procedure call simulates the abstract event and the refinement guarantees the correctness of the resulting algorithm. A preliminary version[7] introduces ideas on a case study and provides an extended abstract of the current paper.

Proof-based Development. Proof-based development methods[5,1,13] integrate formal proof techniques in the development of software systems. The main idea is to start with a very abstract model of the system under development. Details are gradually added to this first model by building a sequence of more concrete events. The relationship between two successive models in this sequence is that of *refinement*[5,1]. The essence of the refinement relationship is that it preserves already proved *system properties* including safety properties and termination. A development gives rise to a number of, so-called, *proof obligations*, which guarantee its correctness. Such proof obligations are discharged by the proof tool using automatic and interactive proof procedures supported by a proof engine[4]. At the most abstract level it is obligatory to describe the static properties of a model's data by means of an "invariant" predicate. This gives rise to proof obligations relating to the consistency of the model. They are required to ensure that data properties which are claimed to be invariant are preserved by the events of the model. Each refinement step is associated with a further invariant which relates the data of the more concrete model to that of the abstract model and states any additional invariant properties of the (possibly richer) concrete data model. These invariants, so-called *gluing invariants* are used in the formulation of the refinement proof obligations. The goal of a event B development is to obtain a *proved model* and to implement the correctness-by-construction[12] paradigm. Since the development process leads to a large number of proof obligations, the mastering of proof complexity is a crucial issue. Even if a proof tool is available, its effective power is limited by classical results over logical theories and we must distribute the complexity of proofs over the components of the current development, e.g. by refinement. Refinement has the potential to decrease the complexity of the proof process whilst allowing for traceability of requirements. The price to pay is to face possibly complex mathematical theories and difficult proofs. The re-use of developed models and the structuring mechanisms available in B help in decreasing the complexity.

Organization of the paper. Section 2 introduces introduces definitions related to the methodology and details for representing the pattern used for designing the algorithm. Section 3 describes the development of the *shortest-path* problem and the relationship between models and programs; it illustrates the methodology for developing structured programs. Section 4 develops Floyd's algorithm using the same pattern and discusses issues related to the implementation in the C programming language. Finally, we conclude our work in the last section.

3 The modelling framework

We do not recall oncepts of the Event B modelling language developed by J.-R. Abrial[2,6]; we sketch the general methodology we are applying. The ingredients for describing the modelling process based on events and model refinement can be found in [2,6]. We assume that the goal is to solve a given problem described by a semi-formal mathematical text and we assume that the problem is defined by a precondition and a postcondition[13]. The modelling process starts by identifying the domain of the problem and it is expressed using the concept of **CONTEXT**. A **CONTEXT** PB (see Figure 1) states the theoretical notions required to be able to express the problem statement in a formal way. The **CONTEXT** PB declares

- a domain D which is the global set of possible values of the current system.
- a list of constants x, which is specifying the input of the system under development, P, which is the set of values for x defining the precondition, and Q, which is a binary relation over D defining the postcondition of the problem.
- a list of axioms assigns types to constants and adds knowledges to the RODIN environment; for instance, the axiom 5 states that there is always a solution y, when the input value x satisfies the precondition P.

A **CONTEXT** may include a clause THEOREMS containing properties derivable in the theory defined by sets, contants and axioms; theorems are discharged using the proof assistant of the tool RODIN. The underlying language is a set-theoretical language partially given in Table 1. When an expression E is given, a well-definedness condition is generated by the tool; this point llows us to check that some side conditions are true. For instance, the expression $f(x)$ generates a condition as $x \in dom(f)$.

The first model provides the declaration of the procedure call. Variables y are *call-by-reference* parameters, constants x are *call-by-value* parameters and carrier sets s are used to type informations and also for defining a generic procedure:

CONTEXT PB

SETS
D

CONSTANTS
x, P, Q

AXIOMS
$axm1 : x \in D$ $/*$ x belongs to a general set of the problem domain $*/$
$axm2 : P \subseteq D$ $/*$ P is a set defining the precondition $*/$
$axm3 : Q \subseteq D \times D$ $/*$ Q is a binary relation over S defining the postcondition $*/$
$axm4 : x \in P$ $/*$ x is supposed to satisfy the precondition P $*/$
$axm5 : \forall a \cdot a \in P \Rightarrow (\exists b \cdot a \mapsto b \in Q)$ $/*$ there is at least one solution for each data x satisfying the precondition P $*/$

END

Fig. 1. Context for modelling the problem PB

Name	Syntax	Definition
Binary relation	$s \leftrightarrow t$	$\mathcal{P}(s \times t)$
Composition of relations	$r_1; r_2$	$\{x, y \mid x \in a \wedge y \in b \wedge$
		$\exists z.(z \in c \wedge x, z \in r_1 \wedge z, y \in r_2)\}$
Inverse relation	r^{-1}	$\{x, y \mid x \in \mathcal{P}(a) \wedge y \in \mathcal{P}(b) \wedge y, x \in r\}$
Domain	$\mathsf{dom}(r)$	$\{a \mid a \in s \wedge \exists b.(b \in t \wedge a \mapsto b \in r)\}$
Range	$\mathsf{ran}(r)$	$\mathsf{dom}(r^{-1})$
Identity	$\mathsf{id}(s)$	$\{x, y \mid x \in s \wedge y \in s \wedge x = y\}$
Restriction	$s \lhd r$	$\mathsf{id}(s); r$
Co-restriction	$r \rhd s$	$r;\ \mathsf{id}(s)$
Anti-restriction	$s \mathbin{⩤} r$	$(\mathsf{dom}(r) - s) \lhd r$
Anti-co-restriction	$r \mathbin{⩥} s$	$r \rhd (\mathsf{ran}(r) - s)$
Image	$r[w]$	$\mathsf{ran}(w \lhd r)$
Overriding	$q \mathbin{\lhd\!\!-} r$	$(\mathsf{dom}(r) \mathbin{⩤} q) \cup r$
Partial Function	$s \mathbin{⇸} t$	$\{r \mid r \in s \leftrightarrow t \wedge (r^{-1}; r) \subseteq \mathsf{id}(t)\}$

Table 1
Set-theoretical notation for event B models

procedure call(**x; var** y)

precondition $y = y_0 \wedge Init(y_0, x, D) \wedge \widetilde{P}(x)$

postcondition $\widetilde{Q}(x, y)$

```
MACHINE PREPOST
SEES PB
VARIABLES
    y
INVARIANTS
    inv1 : y ∈ D
EVENTS
INITIALISATION
    BEGIN
        act1 : y :∈ D
    END
EVENT call
    BEGIN
        act1 : y : |(x ∈ P ∧ x ↦ y′ ∈ Q)
    END
END
```

Fig. 2. Machine defining the model for modelling the problem PB

Figure 2 describes the complete model for the problem PB; it is expressd by a generic procedure stating the pre/post-specification. The term *procedure* can be substituted by the term *method*. The current status of the development can be represented as follow:

$$\texttt{call(x,y)} \xrightarrow{call-as-event} \text{PREPOST} \xrightarrow{\text{SEES}} \text{PB}$$

The statement of a given problem in the Event B modelling language is relatively direct, as long as we are able to express the mathematical underlying theory using the mechanism of contexts. The existence of a solution y for each value x is assumed to be an axiom; however, it would be better to derive the property as a theorem and it means that we should develop a way to validate axioms to ensure the consistency of the underlying theory.

The next section illustrates the technique used for developing new algorithms. We think that it is a good way to teach the design of algorithms. HOARE logic[10] provides a very interesting framework for dealing with specifications an development and our work shows how the ingredients of HOARE logic can be used to provide a general framework for developing sequential programs correct by construction. Event B and the RODIN plateform can be used to teach basic notions like pre and postconditions, invaraint, verification and finally design-by-contract.

Teacher's note: *The challenge of the teacher is to relate the Event B notations to the notations of the programming language. We have used the Event B notations in lectures on fixed-point theory and on the explanation of sequential algorithms. It is then clear that we should provide more systematic rules for deriving algorithms. The management of definitions using a tool, like RODIN, helps students to understand why a function call like $f(x)$ generates conditions like $x \in dom(f)$. Nobody can cheat with the tool. Moreover, when a tool is available for a free download, it is really a teachermate.*

4 The Shortest Path Problem

4.1 Summary of the problem

Floyd's algorithm[9] computes the shortest distances of a graph and is based on an algorithmic design technique called dynamic programming: simpler subproblems are first solved before the full problem is solved. It computes a distance matrix from a cost matrix: the costs of the shortest path between each pair of vertices are in $O(|V|^3)$ time.

Teacher's note: *In the case of Floyd's algorithm, there is a mathematical definition of the matrix we have to compute from a starting state defining the initial basic link between nodes with cost. The function is called* d *and should be first defined in a context of the problem.*

The set of nodes N is $1..n$, where n is a constant value and the graph is simply represented by the distance function d ($d \in N \times N \times N \rightarrow\!\!\!\!\!\!+ \mathbb{N}$) and when the function is not defined, it means that there is no vertex between the two nodes. The relation of the graph is defined as the domain of the function d. n is clearly greater than 1 and it means that the set of nodes is not empty.

The distance function d is defined inductively from bottom to top a ccording to the dynamic programming principle and the next axioms define this function:

- $\boxed{axm1 : d \in N \times N \times N \rightarrow\!\!\!\!\!\!+ \mathbb{N}}$

- $\boxed{axm5 : \forall i \cdot i \in N \Rightarrow 0 \mapsto i \mapsto i \in dom(d) \land\ d(0 \mapsto i \mapsto i) = 0}$

- $$axm6 : \forall i, j, k \cdot \left(\begin{array}{l} \left(\begin{array}{l} k-1 \mapsto i \mapsto j \in dom(d) \\ \land\ (k-1 \mapsto i \mapsto k \notin dom(d) \lor k-1 \mapsto k \mapsto j \notin dom(d)) \end{array} \right) \\ \Rightarrow \\ \left(k \mapsto i \mapsto j \in dom(d) \land\ d(k \mapsto i \mapsto j) = d(k-1 \mapsto i \mapsto j) \right) \end{array} \right)$$

- $$axm7 : \forall i, j, k \cdot \left(\begin{array}{l} k-1 \mapsto i \mapsto j \in dom(d) \\ \land\ k-1 \mapsto i \mapsto k \in dom(d) \\ \land\ k-1 \mapsto k \mapsto j \in dom(d) \\ \land\ d(k-1 \mapsto i \mapsto j) \leq d(k-1 \mapsto i \mapsto k) + d(k-1 \mapsto k \mapsto j) \end{array} \right) \\ \Rightarrow \\ \left(\begin{array}{l} k \mapsto i \mapsto j \in dom(d) \\ \land\ d(k \mapsto i \mapsto j) = d(k-1 \mapsto i \mapsto j) \end{array} \right)$$

- $$axm8 : \forall i, j, k \cdot \left(\begin{array}{l} k-1 \mapsto i \mapsto j \in dom(d) \\ \wedge\ k-1 \mapsto i \mapsto k \in dom(d) \\ \wedge\ k-1 \mapsto k \mapsto j \in dom(d) \\ \wedge\ d(k-1 \mapsto i \mapsto j) > d(k-1 \mapsto i \mapsto k) + d(k-1 \mapsto k \mapsto j) \end{array}\right) \\ \Rightarrow \\ \left(\begin{array}{l} k \mapsto i \mapsto j \in dom(d) \\ \wedge\ d(k \mapsto i \mapsto j) = d(k-1 \mapsto i \mapsto k) + d(k-1 \mapsto k \mapsto j) \end{array}\right)$$

- $$axm9 : \forall i, j, k \cdot \left(\begin{array}{l} \left(\begin{array}{l} k-1 \mapsto i \mapsto j \notin dom(d) \\ \wedge\ k-1 \mapsto i \mapsto k \in dom(d) \\ \wedge\ k-1 \mapsto k \mapsto j \in dom(d) \end{array}\right) \\ \Rightarrow \\ k \mapsto i \mapsto j \in dom(d) \end{array}\right)$$

The optimality property is derived from the definition of d itself, since it starts by defining bottom elements and applies an optimal principle summarized as follows: $D_{i+1}(a, b) = Min(D_i(a, b), D_i(a, i + 1) + D_i(i + 1, b))$ and means that the distances in D_i represent paths with intermediate vertices smaller than i; D_{i+1} is defined by comparing new paths including $i + 1$. D_i is defined by a partial function over $N \times N \times N$. The partiality of d leads to some possible problems for computing the minimum and when at least one term is not defined, we should define a specific definition for the resulting term. Floyd's algorithm provides an algoithmic process for obtaining a matrix of all shortest possible paths with respect to a given initial matrix representing links between nodes together with their distance. Our first attempt was based on the computation of a shortest path between to given nodes a and b. The resulting matrix is called R and a boolean variable FD tells us if the shortest path exists. By the way, this first attempt is not the strict Floyd's algorithm but it will use the same principle of computation for the resulting matrix R.

The first step defines the *context* of the problem and the context is validated by the RODIN platform[14]. We decide to design an algorithm which is computing the value of the shortest path between two given nodes but using the same principle than Floyd's algorithm.

Teacher's note: *The validation of the context SHORTESTPATH0 helps us to define carefully the function d. The translation of mathematocal properties is made easier by the notion of partial function. The expression $D_{i+1}(a,b) = Min(D_i(a,b), D_i(a,i+1) + D_i(i+1,b))$ hides possible underfinedness and generally the non-existence of an edge between two nodes is defined by an extra value like ∞. We have to compute the following value $\lambda i,j \in N.d(l \mapsto i \mapsto j)$ but the λ notation is not directly usable in the B notations. However, we are computing in fact the value of d for the triple $l \mapsto i \mapsto j$ because it seems to be simpler to state.*

4.2 Writing the function call

The first model provides the declaration of the procedure shortestpath. Variables D and FD are *call-by-reference* parameters, constants l, a, b, D are *call-by-value* parameters:

procedure shortestpath$(l,a,b,G;$ **var** $D,FD)$

precondition $G = d_0 \wedge\ FD = FALSE \wedge\ l > 0 \wedge\ a \in N \wedge\ b \in N$

postcondition $(FD = true \Rightarrow D = d(l,a,b))$

We apply the *Call as Event principle* and we have to define a new model called SHORTESTPATH1 , which is defining an event corresponding to the action of calling the procedure.

`shortestpath(l,a,b,g,D,FD)` $\xrightarrow{call-as-event}$ SHORTESTPATH1 $\xrightarrow{\text{SEES}}$ SHORTESTPATH0

Teacher's note: *The event is considered as a function call; we can explain at this time that the event is triggered because the guard is true. It is not a precondition.*

The new model SHORTESTPATH1 is using definitions of the context SHORTESTPATH0 . The event FLOYDKO models the fact that the call of `floyd` is returning a value FALSE for FD: there is no path between a and b. The event FLOYDOK returns the value TRUE for FDand the value of the minimal path from a to b. The two events are also interpreted by a procedure which is called with respect to the existence of a path.

MACHINE SHORTESTPATH1
SEES SHORTESTPATH0

VARIABLES
D
FD

INVARIANTS
$inv1 : D \in N \times N \twoheadrightarrow \mathbb{N}$
$inv2 : FD \in BOOL$

EVENTS

INITIALISATION
BEGIN
$act1 : D :\mid \left(\begin{array}{l} D' \in N \times N \twoheadrightarrow \mathbb{N} \\ \wedge (\forall i, j \cdot 0 \mapsto i \mapsto j \in dom(d) \Rightarrow i \mapsto j \in dom(D') \wedge D'(i \mapsto j) = d(0 \mapsto i \mapsto j))) \end{array} \right)$
$act2 : FD := FALSE$
END

EVENT shortestpathOK
WHEN
$grd1 : l \mapsto a \mapsto b \in dom(d)$
THEN
$act1 : D(a \mapsto b) := d(l \mapsto a \mapsto b)$
$act2 : FD := TRUE$
END

EVENT shortestpathKO
WHEN
$grd1 : l \mapsto a \mapsto b \notin dom(d)$
THEN
$act1 : FD := FALSE$
END

END

Now, we have two events really non-deterministic, since they are defined using the constant d which should be computed in fact!. The solution is to refine the model SHORTESTPATH1 into a new model SHORTESTPATH2 which reduces non-determinism.

Teacher's note: *It is very important to explain the difference between a flexible [11] variable and a rigid variable. Rigid variable like d denotes values which are defined as mathematical static objects and flexible variables denotes a name which is assigned to a value depending on the current state.*

4.3 Refining the procedure call

The main idea is to unfold the calls or to refine the events to get a model which is closer to an algorithm. We introduce several new variables:

- D and FD are both variables of the models SHORTESTPATH1 and SHORTESTPATH2 .
- c ($inv1 : c \in C$) expresses the control flow and the possible values of c are in the set C ($axm15 : C = \{start, end, step1, step2, step3, finalstep\}$).
- $D1$, $D2$ and $D3$ are three variables storing the values required for computing the next value of D at a given step; the values may be undefined and the undefinedness is controlled by the three variables $FD1$, $FD2$ and $FD3$. Variables are typed according to the following part of the invariant:
  - $\boxed{inv2 : D1 \in \mathbb{Z}}$
  - $\boxed{inv3 : D2 \in \mathbb{Z}}$

· $\boxed{inv4 : D3 \in \mathbb{Z}}$

· $\boxed{inv5 : FD1 \in BOOL}$

· $\boxed{inv6 : FD2 \in BOOL}$

· $\boxed{inv7 : FD3 \in BOOL}$

We do not give more details for the invariant and we will give later the details of the invariant of the current model. First we give the different events of the model SHORTESTPATH2 .

The event
INITIALISATION
is simply setting the variables as follows: $act1 : D := D0$, $act2 : FD := FALSE$, $act3 : FD1 := FALSE$, $act4 : FD2 := FALSE$, $act5 : FD3 := FALSE$, $act6 : D1 :\in \mathbb{Z}$, $act7 : D2 :\in \mathbb{Z}$, $act8 : D3 :\in \mathbb{Z}$, $act10 : c := start$.

Since $d(0 \mapsto i \mapsto j)$ models the existence of an elementary path from i to j, $D0$ is defined by the following axioms:

- $\boxed{axm12 : D0 \in N \times N \rightarrowtail\!\!\!\!\!\rightarrow \mathbb{N}}$
- $\boxed{axm13 : dom(D0) = \{i \mapsto j | 0 \mapsto i \mapsto j \in dom(d)\}}$
- $\boxed{axm14 : \forall i, j \cdot i \mapsto j \in dom(D0) \Rightarrow D0(i \mapsto j) = d(0 \mapsto i \mapsto j)}$

Now, we can introduce refinement of existing events of SHORTESTPATH1 and new events which are not in the abstraction.

4.3.1 Refining events of SHORTESTPATH1

First, we give elements for competing the invariant; the typing informations can be completed as follows and they correspond to an analysis of the definition of d. We introduce a new variable c which is expressing the control state and whose possible values are given by the set C: $C = \{start, end, step1, step2, step3, finalstep\}$. We summarize the different steps for computing D.

> **Teacher's note:** *Using a graphical notation helps to communicate the meaning of control assertions. The steps of the algorithm appear. Moreover, steps provide a guide for defining the invariant which is based on the construction of d.*

$$\begin{tikzcd}
 & c = start \arrow[dl, "l{\neq}0\ floyd\ call"'] \arrow[dr, "l{=}0"] & \\
c \in \{step1, step2, step3\} \arrow[r, "floyd\ call\ return"] & c = finalstep \arrow[r, "analysis"] & c = end
\end{tikzcd}$$

Teacher's note: *The invariant is based on the decomposition into steps and each step analyses the definition of values required for computing the minimum of $D1$ and $D2{+}D3$. The invariant should take into account th definedness of these values and the tool helps us to complete the invariant.*

The *analysis* step provides a decision depending on the values of $D1$, $D2$ and $D3$, if they are defined. The boolean expression $FD1 \wedge\ (FD2 \vee FD3)$ is the key for updating $D(a \mapsto b)$ and it is triggered , when the control is finalstep.

Teacher's note: *The expression $D_{i+1}(a,b) = Min(D_i(a,b), D_i(a,i{+}1){+}D_i(i{+}1,b))$ should be carefully analysed and it allows us to derive specific conditions for structuring the algorithm.*

When the control is at start:

- when l is initially equal to 0, D and d are equal too; D is defined when d is defined and reciprocally:
  - $\boxed{inv20 : c = start \wedge\ a \mapsto b \notin dom(D) \wedge\ l = 0 \Rightarrow 0 \mapsto a \mapsto b \notin dom(d)}$
  - $$\boxed{inv8 : \left(\begin{array}{l} \left(\begin{array}{l} c = start \\ \wedge\, a \mapsto b \in dom(D) \\ \wedge\, l = 0 \end{array} \right) \\ \Rightarrow \\ \left(\begin{array}{l} 0 \mapsto a \mapsto b \in dom(d) \\ \wedge\, D(a \mapsto b) = d(0 \mapsto a \mapsto b) \end{array} \right) \end{array} \right)}$$
- when l is not equal to 0 and there is no path from a to b with intermadiate nodes whose numbers is smaller than $l-1$, $a \mapsto b$ is not in D.

- $$inv34: \left(\begin{pmatrix} c = start \\ \wedge\, l \neq 0 \\ \wedge\, l-1 \mapsto a \mapsto b \notin dom(d) \\ \wedge\, l-1 \mapsto l \mapsto b \notin dom(d) \end{pmatrix} \\ \Rightarrow \\ a \mapsto b \notin dom(D) \right)$$

- $$inv37: \left(\begin{pmatrix} c = start \\ \wedge\, l-1 \mapsto a \mapsto b \notin dom(d) \\ \wedge\, l-1 \mapsto a \mapsto l \notin dom(d) \end{pmatrix} \\ \Rightarrow \\ a \mapsto b \notin dom(D) \right)$$

When the control is at end:

If the control is at end, the invariant enumerates the different cases for the resulting computation. The variable D should contain the values correspondin to l.

- $$inv12: \begin{array}{l} \begin{pmatrix} c = end \\ \wedge\, FD = TRUE \end{pmatrix} \\ \Rightarrow \\ \begin{pmatrix} a \mapsto b \in dom(D) \\ \wedge\, l \mapsto a \mapsto b \in dom(d) \\ \wedge\, D(a \mapsto b) = d(l \mapsto a \mapsto b) \end{pmatrix} \end{array}$$

- $inv14: c = end \wedge\ FD = FALSE \Rightarrow a \mapsto b \notin dom(D) \wedge\ l \mapsto a \mapsto b \notin dom(d)$

- $inv18: c = end \wedge\ l \mapsto a \mapsto b \notin dom(d) \Rightarrow FD = FALSE$

- $inv19: c = end \wedge\ a \mapsto b \notin dom(D) \Rightarrow FD = FALSE$

- $inv21: c = end \wedge\ a \mapsto b \in dom(D) \Rightarrow FD = TRUE$

When the control is at finalstep:

The invariant states that the variables $FD1$, $FD2$ and $FD3$ are related to the definition of the expression $Min(D(a,b), D(a,l) + D(l,b))$. $Min(D(a,b), D(a,l) + D(l,b))$. is defined,if, end only, if $FD1 \wedge\ (FD2 \vee FD3)$. The invariant explores the different cases for the definition of D for the given pairs. Moreover, the values are stired in the variables $D1$, $D2$ and $D3$ when defined.

- $$\boxed{inv11 : \begin{array}{l} c = finalstep \wedge\ FD3 = TRUE \\ \Rightarrow \\ l-1 \mapsto l \mapsto b \in dom(d) \wedge\ D3 = d(l-1 \mapsto l \mapsto b) \end{array}}$$

- $$\boxed{inv15 : \begin{array}{l} c = finalstep \wedge\ FD1 = TRUE \\ \Rightarrow \\ l-1 \mapsto a \mapsto b \in dom(d) \wedge\ D1 = d(l-1 \mapsto a \mapsto b) \end{array}}$$

- $$\boxed{inv16 : \begin{array}{l} c = finalstep \wedge\ FD2 = TRUE \\ \Rightarrow \\ l-1 \mapsto a \mapsto l \in dom(d) \wedge\ D2 = d(l-1 \mapsto a \mapsto l) \end{array}}$$

- $$\boxed{inv13 : \left(\begin{array}{l} \left(\begin{array}{l} c = finalstep \\ \wedge\, FD1 = FALSE \\ \wedge\, (FD2 = FALSE \vee FD3 = FALSE) \end{array}\right) \\ \Rightarrow \\ \left(\begin{array}{l} l \mapsto a \mapsto b \notin dom(d) \\ \wedge\, a \mapsto b \notin dom(D) \end{array}\right) \end{array}\right)}$$

- $\boxed{inv24 : c = finalstep \wedge\ FD3 = FALSE \Rightarrow l-1 \mapsto l \mapsto b \notin dom(d)}$

- $\boxed{inv27 : c = finalstep \wedge\ FD2 = FALSE \Rightarrow l-1 \mapsto a \mapsto l \notin dom(d)}$

- $\boxed{inv29 : c = finalstep \wedge\ FD1 = FALSE \Rightarrow l-1 \mapsto a \mapsto b \notin dom(d)}$

- $$inv38 : \left(\begin{pmatrix} c = finalstep \\ \wedge\ FD1 = TRUE \\ \wedge\ (FD2 = FALSE \vee FD3 = FALSE) \\ \\ \end{pmatrix} \\ \Rightarrow \\ \begin{pmatrix} l \mapsto a \mapsto b \in dom(d) \\ \wedge\ d(l \mapsto a \mapsto b) = d(l-1 \mapsto a \mapsto b) \end{pmatrix} \right)$$

The diagram shows that `shortestpath` is made up of three steps.

$$\begin{tikzcd} c = step1 \arrow[d, "evaluation\ FD1{,}D1"'] & & \\ c = step2 \arrow[r, "evaluation\ FD2{,}D2"] & c = step3 \arrow[r, "evaluation\ FD3{,}D3"] & c = finalstep \end{tikzcd}$$

When the conrol is in $\{step1, step2, step3\}$:

- When the control is in $\{step1, step2, step3\}$, since $\boxed{inv28 : c \neq start \wedge\ c \neq end \Rightarrow l \neq 0}$, l is not equal to 0.
- When the control is at $step1$, l is not equal to 0. There are two conditions for the undefinedness of D in relationship to d.
  - $$inv33 : \begin{array}{l} c = step1 \wedge\ l-1 \mapsto a \mapsto b \notin dom(d) \wedge\ l-1 \mapsto l \mapsto b \notin dom(d) \\ \Rightarrow \\ a \mapsto b \notin dom(D) \end{array}$$
  - $$inv36 : \begin{array}{l} c = step1 \wedge\ l-1 \mapsto a \mapsto b \notin dom(d) \wedge\ l-1 \mapsto a \mapsto l \notin dom(d) \\ \Rightarrow \\ a \mapsto b \notin dom(D) \end{array}$$
- When the control is in $step2$, either the evaluation of $D1$ is successful or not.
  - $inv9 : c = step2 \wedge\ FD1 = TRUE \Rightarrow l-1 \mapsto a \mapsto b \in dom(d) \wedge\ D1 = d(l-1 \mapsto a \mapsto b)$

· $\boxed{inv22 : c = step2 \wedge\ FD1 = FALSE \Rightarrow l-1 \mapsto a \mapsto b \notin dom(d)}$

· $inv32 :$
$$c = step2 \wedge\ FD1 = FALSE \wedge\ l-1 \mapsto l \mapsto b \notin dom(d) \\ \Rightarrow \\ a \mapsto b \notin dom(D)$$

· $inv35 :$
$$c = step2 \wedge\ l-1 \mapsto a \mapsto l \notin dom(d) \wedge\ FD1 = FALSE \\ \Rightarrow \\ a \mapsto b \notin dom(D)$$

- When the control is in $step3$, either the evaluation of $D2$ is successful or not.

· $inv10 :$
$$c = step3 \wedge\ FD2 = TRUE \\ \Rightarrow \\ l-1 \mapsto a \mapsto l \in dom(d) \wedge\ D2 = d(l-1 \mapsto a \mapsto l)$$

· $inv17 :$
$$c = step3 \wedge\ FD1 = TRUE \\ \Rightarrow \\ l-1 \mapsto a \mapsto b \in dom(d) \wedge\ D1 = d(l-1 \mapsto a \mapsto b)$$

· $\boxed{inv23 : c = step3 \wedge\ FD2 = FALSE \Rightarrow l-1 \mapsto a \mapsto l \notin dom(d)}$

· $\boxed{inv25 : c = step3 \wedge\ FD1 = FALSE \Rightarrow l-1 \mapsto a \mapsto b \notin dom(d)}$

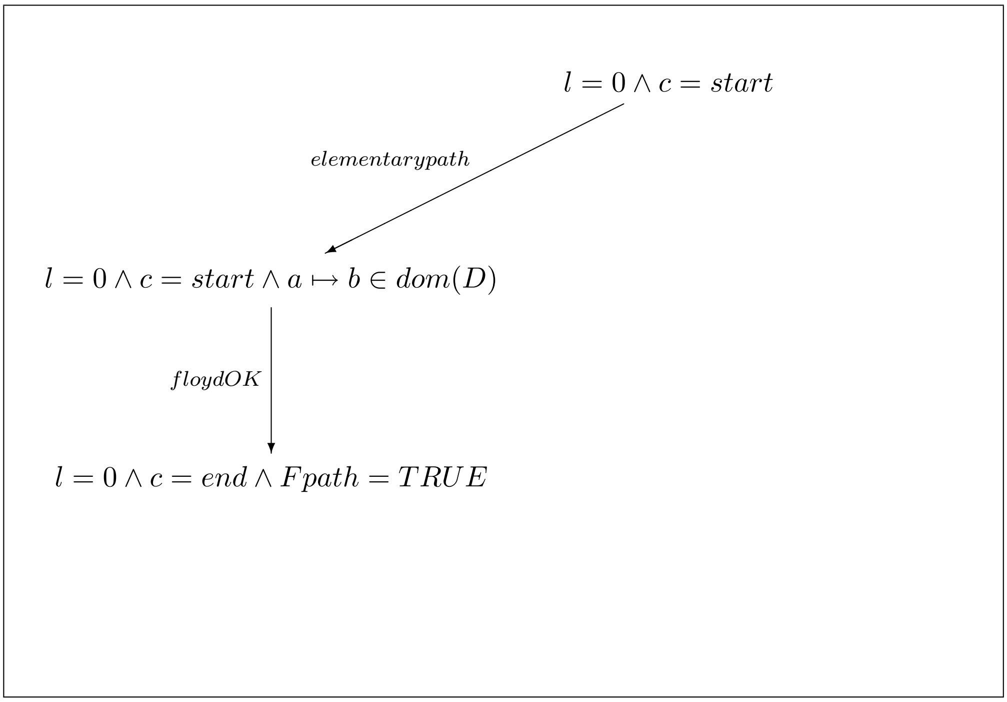
· $\boxed{inv26 : c \neq finalstep \wedge\ c \neq end \wedge\ 0 \mapsto a \mapsto b \notin dom(d) \Rightarrow a \mapsto b \notin dom(D)}$

· $\boxed{inv30 : c = step3 \wedge\ FD1 = FALSE \wedge\ FD2 = FALSE \Rightarrow a \mapsto b \notin dom(D)}$

· $inv31 :$
$$c = step3 \wedge\ FD1 = FALSE \wedge\ l-1 \mapsto l \mapsto b \notin dom(d) \\ \Rightarrow \\ a \mapsto b \notin dom(D)$$

Refining shortestpathOK

Now, we define each transition between the different steps according to the invariant. We consider severall possible cases depending on l and other conditions. When the value of l is 0 and when D is defined for the pair $a \mapsto b$, it means that there is a path between a and b without any intermediate node. It is the basic case and one returns the value TRUE for FD. The control is set to end, since the procedure is completed:



EVENT shortestpathOK
REFINES shortestpathOK
  WHEN
    $grd2 : l = 0$
    $grd1 : a \mapsto b \in dom(D)$
    $grd3 : c = start$
  THEN
    $act2 : FD := TRUE$
    $act3 : c := end$
  END

When the control expresses the accessibility of the last control point ($c = finalstep$) and when the three values $D1$, $D2$ and $D3$ are defined and satisfy the condition $D1 \leq D2 + D3$, we can update D in $a \mapsto b$ by $D1$. In fact, the value is not modified. The control is set to the final control point called end. There is a path and FD is set to TRUE.

EVENT shortestpathcallOKmin
REFINES shortestpathOK

WHEN
$grd1 : FD1 = TRUE \land\ FD2 = TRUE \land\ FD3 = TRUE$
$grd2 : D1 \leq D2 + D3$
$grd3 : c = finalstep$
THEN
$act1 : D(a \mapsto b) := D1$
$act2 : FD := TRUE$
$act3 : c := end$
END

The next case is stating that there is a new path from a to b, which is shortest than the current one ($grd3 : D1 > D2 + D3$) and we should update D by the new value $D2 + D3$.

EVENT shortestpathcallOKmax
REFINES shortestpathOK
WHEN
$grd1 : FD1 = TRUE \land\ FD2 = TRUE \land\ FD3 = TRUE$
$grd2 : c = finalstep$
$grd3 : D1 > D2 + D3$
THEN
$act1 : D(a \mapsto b) := D2 + D3$
$act2 : c := end$
$act3 : FD := TRUE$
END

The next possible case is that the value $D1$ is not defined; it means that there is not yet a path from a to b and we have discovered that there is a node which can be reached from a and which can reach b. Hence, the variable D is defined in $a \mapsto b$ by the value $D2 + D2$.

EVENT shortestpathFD2FD3
REFINES shortestpathOK
WHEN
$grd1 : c = finalstep$
$grd2 : FD1 = FALSE \land\ FD2 = TRUE \land\ FD3 = TRUE$
THEN
$act1 : D(a \mapsto b) := D2 + D3$
$act2 : FD := TRUE$
$act3 : c := end$
END

Finally, when either $D2$ or $D3$ is not defined, the value of D is not modified and remains equal to $D1$.

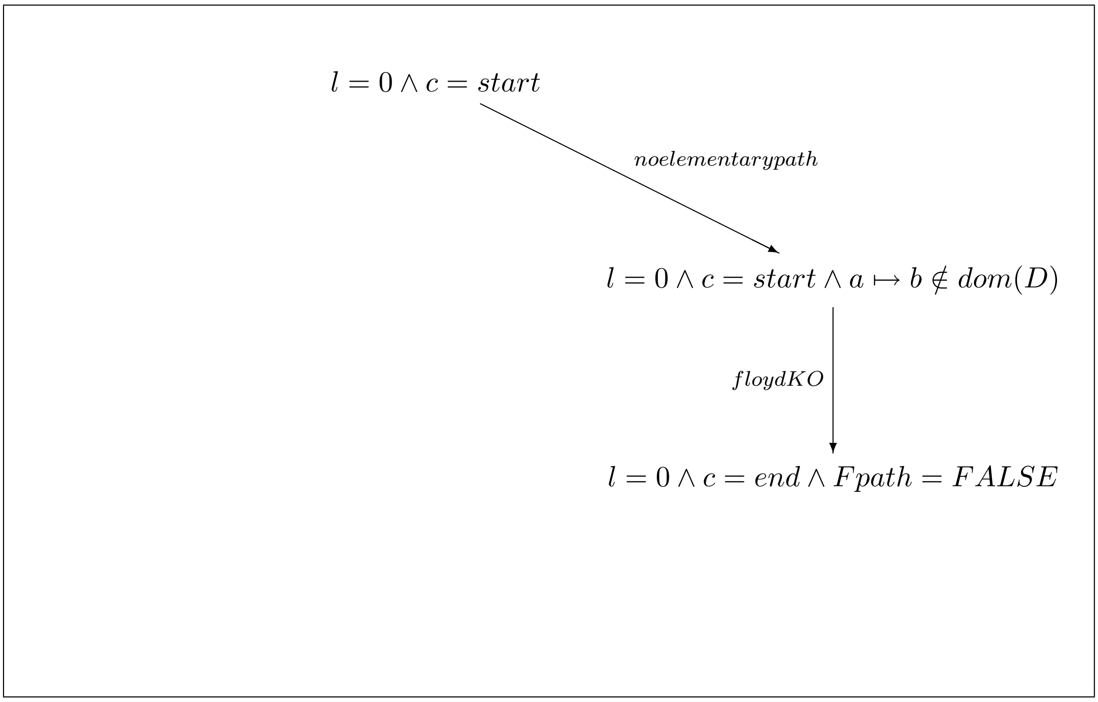
EVENT shortestpathFD1
REFINES shortestpathOK
  WHEN
    $grd1 : c = finalstep$
    $grd2 : FD1 = TRUE \wedge \ (FD1 = FALSE \vee FD2 = FALSE)$
  THEN
    $act1 : D(a \mapsto b) := D1$
    $act2 : c := end$
    $act3 : FD := TRUE$
  END

The refinement of abstract events should be completed by events which compute the values $D1$, $D2$ and $D3$.

Refining shortestpathKO

We consider severall possible cases depending on l and other conditions.

When the value of l is 0 and when D is not defined for the pair $a \mapsto b$, it means that there is no elementary path between a and b. It is the basic case and one returns the value FALSE for FD. The control is set to end, since the procedure is completed:



EVENT shortestpathKO
REFINES shortestpathKO
WHEN
$grd2 : l = 0$
$grd1 : a \mapsto b \notin dom(D)$
$grd3 : c = start$
THEN
$act1 : FD := FALSE$
$act2 : c := end$
END

When the value of l is not 0 and when $D1$ is not defined and either $D2$ is not defined, or $D3$ is not defined, for the pair $a \mapsto b$, it means that there is no path between a and b. One returns the value FALSE for FD. The control is set to end, since the procedure is completed:

EVENT shortestpathKOelse
REFINES shortestpathKO
WHEN
$grd1 : c = finalstep$
$grd2 : FD1 = FALSE \wedge \ (FD2 = FALSE \vee FD2 = FALSE)$
THEN
$act1 : c := end$
$act2 : FD := FALSE$
END

4.3.2 Introducing new events in SHORTESTPATH2

The first new event models the calling step of the procedure `floyd` and it transfers the control to the control point $step1$.

EVENT shortestpathcallone
WHEN
$grd1 : l > 0$
$grd2 : c = start$
THEN
$act1 : c := step1$
END

Now, we consider the three steps for computing $D1$, $D2$ and $D3$.

Calling the procedure `floyd` for evaluating $D1$ and $FD1$

The event $shortestpathcalltwook$ simulates the procedure for computing $D1$, which is $d(l-1 \mapsto a \mapsto b)$ and which is successfully computed, since $FD1$ is TRUE. The event $shortestpathcalltwoko$ simulates the procedure for computing $D1$, which is $d(l-1 \mapsto a \mapsto b)$ and which is unsuccessfully computed, since $FD1$ is FALSE.

EVENT floydcalltwook
  WHEN
    $grd1 : c = step1$
    $grd2 : l-1 \mapsto a \mapsto b \in dom(d)$
  THEN
    $act1 : D1 := d(l-1 \mapsto a \mapsto b)$
    $act2 : FD1 := TRUE$
    $act3 : c := step2$
  END

EVENT shortestpathcalltwoko
  WHEN
    $grd1 : l-1 \mapsto a \mapsto b \notin dom(d)$
    $grd2 : c = step1$
  THEN
    $act1 : FD1 := FALSE$
    $act2 : c := step2$
  END

Calling the procedure `floyd` for evaluating $D2$ and $FD2$

The event $shortestpathcallthreeok$ simulates the procedure for computing $D2$, which is $d(l-1 \mapsto a \mapsto l)$ and which is successfully computed, since $FD2$ is TRUE. The event $shortestpathcallthreeko$ simulates the procedure for computing $D2$, which is $d(l-1 \mapsto a \mapsto l)$ and which is unsuccessfully computed, since $FD2$ is FALSE.

EVENT shortestpathcallthreeok
  WHEN
    $grd1 : c = step2$
    $grd2 : l-1 \mapsto a \mapsto l \in dom(d)$
  THEN
    $act1 : D2 := d(l-1 \mapsto a \mapsto l)$
    $act2 : FD2 := TRUE$
    $act3 : c := step3$
  END

EVENT shortestpathcallthreeko
  WHEN
    $grd1 : c = step2$
    $grd2 : l-1 \mapsto a \mapsto l \notin dom(d)$
  THEN
    $act1 : c := step3$
    $act2 : FD2 := FALSE$
  END

Calling the procedure `floyd` for evaluating $D3$ and $FD3$

The event $shortestpathcallfourok$ simulates the procedure for computing $D3$, which is $d(l-1 \mapsto l \mapsto b)$ and which is successfully computed, since $FD3$ is TRUE. The event $shortestpathcallfourko$ simulates the procedure for computing $D3$, which is $d(l-1 \mapsto l \mapsto b)$ and which is unsuccessfully computed, since $FD3$ is FALSE.

EVENT shortestpathcallfourok
  WHEN
    $grd1 : c = step3$
    $grd2 : l - 1 \mapsto l \mapsto b \in dom(d)$
  THEN
    $act1 : c := finalstep$
    $act2 : D3 := d(l - 1 \mapsto l \mapsto b)$
    $act3 : FD3 := TRUE$
  END

EVENT shortestpathcallfourko
  WHEN
    $grd1 : c = step3$
    $grd2 : l - 1 \mapsto l \mapsto b \notin dom(d)$
  THEN
    $act1 : FD3 := FALSE$
    $act2 : c := finalstep$
  END

4.4 Producing the shortestpath procedure

The shortestpath procedure can be derived from the list of events of the model SHORTESTPATH2 and we structure events into conventional programming structures like `while` or `if` statements. J.-R. Abrial[3] has proposed several rules for producing algorithmic statements. The next diagram gives the complete description of the process we have followed:

$$\begin{tikzcd}
\texttt{shortestpath(l,a,b,G,D,FD)} \arrow[r, "call-as-event"] \arrow[d, "call"'] & \text{SHORTESTPATH1} \arrow[r, "\text{SEES}"] \arrow[d, "\text{REFINEMENT}"'] & \text{SHORTESTPATH0} \\
\texttt{floyd} & \text{SHORTESTPATH2} \arrow[l, "mapping"'] \arrow[ur, "\text{SEES}"] &
\end{tikzcd}$$

The procedure header is `shortestpath(l,a,b,G,D,FD)` and the text of the procedure is given by the algorithms 1 and 2.

Algorithm 1: Algorithm Version 1

precondition : $l \in 1..n \wedge$
postcondition : D, FD
local variables: $FD1, FD2, FD3 \in BOOL$

$FD := FALSE;$
$FD1 := FALSE;$
$FD2 := FALSE;$
$FD3 := FALSE;$
if $l = 0$ **then**
  if $(a, b) \in dom(D)$ **then**
    $FD := TRUE;$
    $R := D[a, b];$
  else
    $FD := FALSE;$
else
  $floyd(l - 1, a, b, D1, FD1); floyd(l - 1, a, l, D2, FD2); floyd(l - 1, l, b, D3, FD3);$
  case $FD1 \wedge FD2 \wedge FD3$
    if $D1 < D2 + D3$ **then**
      $R := D1;$
    else
      $R := D2 + D3;$
    ;
    $FD := TRUE;$
  ;
  case $FD1 \wedge (\neg FD2 \vee \neg FD3)$
    $R := D1;$
    $FD := TRUE;$
  ;
  case $\neg FD1 \wedge (FD2 \wedge FD3)$
    $R := D2 + D3;$
    $FD := TRUE;$
  ;
  case $\neg FD1 \wedge (\neg FD2 \vee \neg FD3)$
    $FD := FALSE;$
  ;
;

The two next frames are containing C codes produced for the two algorithms 4.4 and 4.4; we have produced the C codes by hand and we have forgotten that C arrays starts by 0 and it means that our initial calls were wrongly written. It is clear that we need a way to produce codes in a mechanized way. Moreover, there are some conditions to check and some interactions to manage with the user to help in choices.

Teacher's note: *It is the time to recall that we are planning to use a real programming language and that we should represent abstract objects by concrete objects. It would be better to add informations on the integers of computer scientists and it is easy to add the constraint.*

```
/*  N = 1..n-1 */
void shortestpath (int l, int a, int b,int g[][n], int *D, int *FD)
{
  int D1,D2,D3,FD1,FD2,FD3;
  *FD = 0; FD1=0;FD2=0;FD3=0;
  if (l==0)
    {
      if (g[a][b] != NONE)
       { *FD = 1; *D = g[a][b];}
    }
  else
    {
       shortestpath(l-1,a,b,g,&D1,&FD1); shortestpath(l-1,a,l,g,&D2,&FD2);
       shortestpath(l-1,l,b,g,&D3,&FD3);
      if (FD1 == 1 && (FD2==1 &&  FD3==1))
        {  if (D1 < D2+D3)
           {*D= D1;}
          else
           {*D=D2+D3;};
           *FD = 1;
        }
      else if (FD1==1 && ( FD2==0 ||  FD3==0))
        {*D= D1;*FD = 1;}
      else if (FD1==0 &&  ( FD2 == 1 && FD3==1)) {*D=D2+D3; *FD=1;}
      else /* (FD1==0 && ( FD2==0 || FD3==0)) */ { *FD = 0;}
    }
}
```

```
/*  N = 1..n-1 */
void shortestpath (int l, int a, int b,int g[][n], int *D, int *FD)
{
  int D1,D2,D3,FD1,FD2,FD3;

  *FD = 0; FD1=0;FD2=0;FD3=0;
  if (l==0)
    {
      if (g[a][b] != NONE)
       { *FD = 1; *D = g[a][b];}
    }
  else
    {
       shortestpath(l-1,a,b,g,&D1,&FD1);
       if (FD1 == 1) {
          shortestpath(l-1,a,l,g,&D2,&FD2);
          if (FD2==1) {
            shortestpath(l-1,l,b,g,&D3,&FD3);
            if (FD3==1) {
            if (D1 < D2+D3)
              {*D= D1;}
            else
              {*D=D2+D3;};
            *FD = 1;}
            else
              {*D=D1;*FD=1;}}
          else
            {*D=D1;*FD=1;}}
       else
         {
           if  ( FD2 == 1 && FD3==1) {*D=D2+D3; *FD=1;}
       else
         {*FD=0;};}
    }}
```

The complete development has a cost related to proof obligations. The refinement generates 493 proof obligations and 328 proof obligations were automaically discharged. 165 proof obligations were manually discharged with minor interactions.

Algorithm 2: Algorithm Version 2

precondition : $l \in 1..n \wedge a, b \in \mathbb{N} \wedge G \in N \times N \nrightarrow N$
postcondition : D, FD
local variables: $FD1, FD2, FD3 \in BOOL$

```
FD := FALSE;
FD1 := FALSE;
FD2 := FALSE;
FD3 := FALSE;
if l = 0 then
    if (a, b) ∈ dom(D) then
        FD := TRUE;
        R := D[a, b];
    else
        FD := FALSE;
else
    shortestpath(l − 1, a, b, D1, FD1);
    if FD1 then
        shortestpath(l − 1, a, l, D2, FD2);
        if FD2 then
            shortestpath(l − 1, l, b, D3, FD3);
            if FD3 then
                if D1 < D2 + D3 then
                    R := D1;
                else
                    R := D2 + D3;
                ;
                FD := TRUE;
            else
                R := D1;
                FD := TRUE;
            ;
        else
            R := D1;
            FD := TRUE;
        ;
    else
        if FD2 ∧ FD3 then
            R := D2 + D3;
            FD := TRUE;
        else
            FD := FALSE;
        ;
;
```

model	Total	Auto	Manual	Reviewed	Undischarged
SHORTESTPATH0	8	8	0	0	0
SHORTESTPATH1	5	4	1	0	0
SHORTESTPATH2	493	328	165	0	0
Global	506	340	166	0	0

Teacher's note: *Proof obligations are not very difficult to discharge;there were based on the properties of* d *and it was boring to click the tool for discharging mechanichally them. Efforts were made on the definition of* d*.*

Now, it turns that our goal was to get Floyd's algorithm and we have an algorithm for computing the existence or the non existence of a shortest path between two nodes. The next section address the question.

5 Floyd's algorithm

We can use the developed algorithm to produce a result equivalent to Floyd's execution. In fact, we apply our algorithm on each pair of possible nodes and we store it in a matrix. The algorithm 5 describes the real algorithm which can be found in any lecture notes.

Algorithm 3: Floyd's Algorithm Wikipedia

precondition : $l \in 1..n \wedge matrix \in N \times N \mapsto N$
postcondition : $matrix \in N \times N \mapsto N \wedge$
local variables: $FD1, FD2, FD3 \in BOOL$

foreach $k = 1; k <= n; k++$ **do**
  foreach $i = 1; i <= n; i++$ **do**
    foreach $j = 1; j <= n; j++$ **do**
      if $matrix[i][j] > (matrix[i][k] + matrix[k][j])$ **then**
        $matrix[i, j] = matrix[i][k] + matrix[k][j]$

Now, we are considering the problem of derivation of this solution. In fact, the development starts from the same context. Two new constants are defined namely DF and Daf. Df is the final value of the matrix D correponding to d for the value l. C is simpler and is defined as follows: $\boxed{axm15 : C = \{start, end, call, finalstep\}}$.

New axioms define new constants:

- $\boxed{axm39 : Df \in N \times N \mapsto N}$
- $\boxed{axm40 : dom(Df) = \{u \mapsto v | l \mapsto u \mapsto v \in dom(d)\}}$

- $\boxed{axm41 : \forall u, v \cdot u \mapsto v \in dom(Df) \Rightarrow Df(u \mapsto v) = d(l \mapsto u \mapsto v)}$
- $\boxed{axm42 : Daf \in N \times N \nrightarrow N}$
- $\boxed{axm43 : l \neq 0 \Rightarrow dom(Daf) = \{u \mapsto v | l - 1 \mapsto u \mapsto v \in dom(d)\}}$
- $\boxed{axm44 : l \neq 0 \Rightarrow (\forall u, v \cdot u \mapsto v \in dom(Daf) \Rightarrow Daf(u \mapsto v) = d(l - 1 \mapsto u \mapsto v))}$
- $\boxed{axm22 : l = 0 \Rightarrow Df = D0}$
- $\boxed{axm23 : l \neq 0 \Rightarrow D0 \subseteq Daf}$
- $\boxed{axm24 : l \neq 0 \Rightarrow Daf \subseteq Df}$
- $\boxed{axm25 : l \neq 0 \Rightarrow (\forall u, v \cdot u \mapsto v \in dom(Daf) \Rightarrow Daf(u \mapsto v) = d(l - 1 \mapsto u \mapsto v))}$
- $\boxed{axm26 : \forall u, v \cdot u \mapsto v \in dom(Df) \Rightarrow Df(u \mapsto v) = d(l \mapsto u \mapsto v)}$
- $\boxed{axm27 : \forall u, v \cdot u \mapsto v \in dom(D0) \Rightarrow D0(u \mapsto v) = d(0 \mapsto u \mapsto v)}$
- $$\boxed{axm28 : \begin{array}{l} \left(l \neq 0 \right) \\ \Rightarrow \\ \forall u, v, w \cdot \left(\begin{array}{l} w \mapsto v \in dom(Df) \\ \wedge\ w \mapsto u \in dom(Daf) \\ \wedge\ u \mapsto v \in dom(Daf) \\ \wedge\ Daf(w \mapsto v) > Daf(w \mapsto u) + Daf(u \mapsto v) \end{array} \right) \end{array}}$$

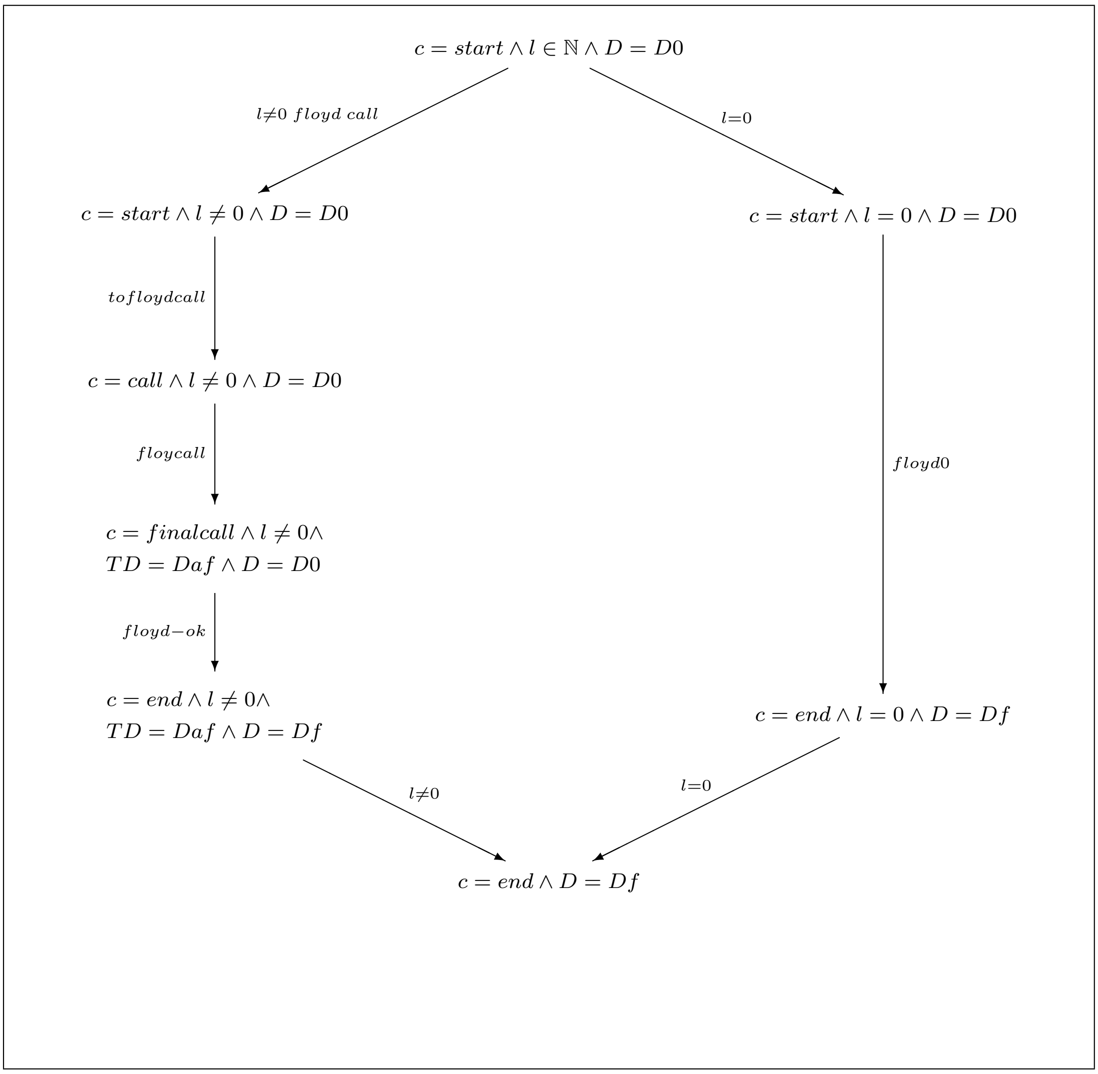
The new model FLOYD1 assigns the value Df to D. The new relationship between models and call is given by the next diagram:

$$\boxed{\texttt{floyd} \xrightarrow{call-as-event} \mathsf{FLOYD1} \xrightarrow{\text{SEES}} \mathsf{FLOYD0}}$$

The problem is to refine the model FLOYD1 to get a list of events which lead to an algorithm. The two constants Df and Daf are used to state the final step and the intermediate step:

- Daf is the result of the call of the under construction algorithm for $l-1$
- Df is the final value which is computed from Daf.

We obtain the following diagram for expressing events corresponding to Floyd's algorithm:



The new model has three variables: c, D, TD.

- $\boxed{inv6 : TD \in N \times N \rightarrow\!\!\!\!\!\rightarrow N}$
- $\boxed{inv1 : c \in C}$
- $\boxed{inv2 : c = start \Rightarrow TD = D0 \wedge\ D = D0}$

- $\boxed{inv3 : c = end \Rightarrow D = Df}$
- $\boxed{inv4 : c = call \Rightarrow TD = D0 \land\ l \neq 0}$
- $\boxed{inv5 : c = finalstep \Rightarrow TD = Daf \land\ l \neq 0}$

Initial conditions over variables are defined by $act1 : D := D0$, $act2 : c := start$, $act3 : TD := D0$. Events are very simple to write from the diagram:

EVENT floyd-ok
REFINES floyd
  WHEN
    $grd1 : c = finalstep$
  THEN

$$
act1 : D, c : | \left(
\begin{array}{l}
\left(D' \in N \times N \nrightarrow N \wedge \; c' = end \right) \\
\wedge \left(\forall w, v \cdot \left(\begin{array}{l}
\left(\begin{array}{l}
w \in N \wedge \; v \in N \\
\wedge \, w \mapsto v \in dom(TD) \\
\wedge \, w \mapsto l \in dom(TD) \\
\wedge \, l \mapsto v \in dom(TD) \\
\wedge \, TD(w \mapsto v) > TD(w \mapsto l) + TD(l \mapsto v)
\end{array} \right) \\
\Rightarrow \\
D'(w \mapsto v) = TD(w \mapsto l) + TD(l \mapsto v)
\end{array} \right) \right) \\
\wedge \left(\forall w, v \cdot \left(\begin{array}{l}
\left(\begin{array}{l}
w \mapsto v \in dom(TD) \\
\wedge \, w \mapsto l \in dom(TD) \\
\wedge \, l \mapsto v \in dom(TD) \\
\wedge \, TD(w \mapsto v) \leq TD(w \mapsto l) + TD(l \mapsto v)
\end{array} \right) \\
\Rightarrow \\
\left(\begin{array}{l}
w \mapsto v \in dom(D') \\
\wedge \, D'(w \mapsto v) = TD(w \mapsto v)
\end{array} \right)
\end{array} \right) \right) \\
\wedge \left(\forall u, v \cdot \left(\begin{array}{l}
\left(\begin{array}{l}
u \mapsto v \in dom(TD) \\
\wedge \, (u \mapsto l \notin dom(TD) \vee l \mapsto v \notin dom(TD))
\end{array} \right) \\
\Rightarrow \\
\left(\begin{array}{l}
u \mapsto v \in dom(D') \\
\wedge \, D'(u \mapsto v) = TD(u \mapsto v)
\end{array} \right)
\end{array} \right) \right) \\
\wedge \left(\forall u, v \cdot \left(\begin{array}{l}
\left(\begin{array}{l}
u \mapsto v \notin dom(TD) \\
\wedge \, (u \mapsto l \in dom(TD) \\
\wedge \, l \mapsto v \in dom(TD))
\end{array} \right) \\
\Rightarrow \\
\left(\begin{array}{l}
u \mapsto v \in dom(D') \\
\wedge \, D'(u \mapsto v) = TD(u \mapsto l) + TD(l \mapsto v)
\end{array} \right)
\end{array} \right) \right) \\
\wedge \left(\forall u, v \cdot \left(\begin{array}{l}
\left(\begin{array}{l}
u \mapsto v \notin dom(TD) \\
\wedge \, (u \mapsto l \notin dom(TD) \vee l \mapsto v \notin dom(TD))
\end{array} \right) \\
\Rightarrow \\
u \mapsto v \notin dom(D')
\end{array} \right) \right)
\end{array}
\right)
$$

END

The event
EVENT floyd-ok
uses a structure of Event B, which is assigning a value to variables and values are in a set. The set can be either empty, a singleton or a general set. In our case, the statement defines only one possible singleton and then the statement is clearly deterministic. However, we can subtitute the event by a call of a new procedure and we should starts a new development in another development using the same principle. We get the nestesd loops.

The two algorithms 5 and 5 are produced from the set of events. The recursive version is simply derived using the control points. The second algorithm is the iterative version which is produced by applying the classical transformations over recursive algorithms. The function nld is derived from an independant development by applying the same pattern.

EVENT floyd0
REFINES floyd
  WHEN
    $grd1 : c = start \wedge\ l = 0$
  THEN
    $act2 : c := end$
  END

EVENT tofloydcall
  WHEN
    $grd1 : c = start \wedge\ l > 0$
  THEN
    $act1 : c := call$
  END

EVENT floydcall
  WHEN
    $grd1 : c = call$
  THEN
    $act1 : c := finalstep$
    $act2 : TD := Daf$
  END

Algorithm 4: Recursive algorithm floyd

precondition : $l \in 1..n \wedge G$
postcondition : D
local variables: TD

$TD := D0$;
if $l \neq 0$ **then**
    $floyd(l-1, G, TD)$;
    $D := nld(TD)$;
else
    $D := TD$;
;

Algorithm 5: Non-recursive algorithm floyd

precondition : $l \in 1..n \wedge G$
postcondition : D
local variables: TD

$TD := D0$;
$l := 0$;
while $l \neq 0$ **do**
    $TD := d(TD)$;
;
$D := TD$;

6 Concluding Remarks and Perspectives

The main objective of the paper is to show how we can develop a sequential structured algorithm using a one-step refinement strategy. We have illustrated the technique introduced by Cansell and Méry in [7] and we have made more precise details left unspecified in the paper. The paper has tried to give hints and advoces to the teacher who wants to use the technique for teaching *correct-by-construction* algorithmics using a tool which is clearly a very good mate for controling the development. You may have questions on the treatment of arithmetics. The technique of developmment is a top/down approach, which is clearly well known in earlier works of Dijkstra[8,13], and to use the refinement for controlling the correctness of the resulting algorithm. It relies on a more fundamental question related to the notion of *problem to solve*. It is also an illustration of the application of the *call-as-event* pattern.

What we have learnt from the case study is summarized as follows:

(i) Developing a first abstract one-shot model using pre/post-condition. It provides the declarations part of the procedure (method) related to the one-shot model. The basic structure to express is the function d which the key of the problem. Constants of the model are defined as *call-by-value* parameters and variable of the model are *call-by-reference* parameters, The context SHORTESTPATH0 is clearly reusable and we have reused it for the effective algorithm of Floyd.

(ii) Refining the abstract model to obtain the body of procedure. New variables are defined as *local variables*. The refinement introduces control states which provide a way to structure the body of the procedure. We have clearly the first control point namely *start* and the last control point namely *end*. The diagram helps to decompose the procedure into steps of the call and a special control point called *call* is introduced. The main question is to obtain a deterministic transition system in the new refinement model.

(iii) If there are still remaining non-deterministic events, we can eliminate the non-deterministic events by developing each non-deterministic event in a specific B development starting by the statement of a new problem expressed by the non-deterministic event itself. In fact, it is what is done with the last version of Floyd's algorithm and the event computing D' from TD is clearly refined to get two nested loops.

(iv) Proof obligations are relatively easy to check because the invariant is written by a list of properties of d according to d. Evene if the number of *manual* proof obligations is high, it was very easy to discharge them using the prover and to reuse former intercative ones.

(v) The translation of Event B model into a C program was carried out by hand and we did a mistake. We forgot that C arrays are starting the index by 0 and it leads to a bad call. We should mechanize this step to avoid this mistake.

Now, if we have to teach concepts, it is easier to teach how to write concepts and definitions using notations provided by Event B (see for instance the table1). You will get a way to check definitions and the type checker is sometimes cruel. We can discuss on many questions using this methodology: coding of numbers, preconditions, postconditions, invariant, assertions, proofs, ldots and questions can lead to replies which are pertinent because of the proof tool.

Future works will provide more case studies and tools for supporting the link between models and codes. We aim to enrich the RODIN tools[14] by specific plug-ins managing libraries of models and implementing new proof obligations.

References

[1] J.-R. Abrial. *The B book - Assigning Programs to Meanings*. Cambridge University Press, 1996.

[2] J.-R. Abrial. B$^{\#}$: Toward a synthesis between z and b. In D. Bert and M. Walden, editors, *3nd International Conference of B and Z Users - ZB 2003, Turku, Finland*, Lectures Notes in Computer Science. Springer, June 2003.

[3] J.-R. Abrial. Event based sequential program development: Application to constructing a pointer program. In *FME 2003*, pages 51–74, 2003.

[4] J.-R. Abrial and D. Cansell. Click'n prove: Interactive proofs within set theory. In *TPHOL 2003*, pages 1–24, 2003.

[5] R. Back. On correct refinement of programs. *Journal of Computer and System Sciences*, 23(1):49–68, 1979.

[6] Dominique Cansell and Dominique Méry. *The event-B Modelling Method: Concepts and Case Studies*, pages 33–140. Springer, 2007. See [**?**].

[7] Dominique Cansell and Dominique Méry. Proved-patterns-based development for structured programs. In Volker Diekert, Mikhail V. Volkov, and Andrei Voronkov, editors, *CSR*, volume 4649 of *Lecture Notes in Computer Science*, pages 104–114. Springer, 2007.

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

[9] Robert W. Floyd. Algorithm 97: Shortest path. *Commun. ACM*, 5(6):345, 1962.

[10] C. A. R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12:576–580, 1969.

[11] Leslie Lamport. The temporal logic of actions. *ACM Trans. Program. Lang. Syst.*, 16(3):872–923, 1994.

[12] Gary T. Leavens, Jean-Raymond Abrial, Don Batory, Michael Butler, Alessandro Coglio, Kathi Fisler, Eric Hehner, Cliff Jones, Dale Miller, Simon Peyton-Jones, Murali Sitaraman, Douglas R. Smith, and Aaron Stump. Roadmap for enhanced languages and methods to aid verification. In *Fifth Intl. Conf. Generative Programming and Component Engineering (GPCE 2006)*, pages 221–235. ACM, October 2006.

[13] C. Morgan. *Programming from Specifications*. Prentice Hall International Series in Computer Science. Prentice Hall, 1990.

[14] project RODIN. Rigorous open development environment for complex systems. http://rodin-b-sharp.sourceforge.net/, 2004. 2004–2007.