Distilling structure in Taverna scientific workflows: a refactoring approach – Additional file

Sarah Cohen-Boulakia*[1,2] , Jiuqiang Chen[1,2,3] , Paolo Missier[4] , Carole Goble[5] , Alan R Williams[5] and Christine Froidevaux[1,2]

[1]Laboratoire de Recherche en Informatique, CNRS UMR 8623, Université Paris Sud, France
[2]AMIB group, INRIA Saclay, France
[3]School of Information Science and Engineering, Lanzhou University, Lanzhou, Gansu, China
[4]University of Newcastle, United Kingdom
[5]University of Manchester, United Kingdom

Email: cohen@lri.fr; chenj@lri.fr; paolo.missier@ncl.ac.uk; carole.goble@manchester.ac.uk; alanrw@cs.man.ac.uk; christine.froidevaux@lri.fr;

*Corresponding author

Abstract

This document provides the additional file for the paper "Distilling structure in Taverna scientific workflows: A refactoring approach". It describes in details the algorithms designed in the refactoring approach introduced.

Pseudo code of the DownDistillation and Distill procedures

```
1 DownDistillation(IN GG[q,v], IN/OUT DSGG: graphs, IN q: node,
2                  IN/OUT SetAU: set of graphs, IN/OUT ListRed: set of nodes)
3 Distill(GG[q,v], DSGG, q);
4 ListRed ← ListRed ∪ {new reduction nodes of GG[q,v]};
5 SetAU ← SetAU ∪ {new autonomous subgraphs of GG[q,v]};
6 foreach autonomous subgraph GG[a,b] in SetAU do
7 |  Distill(GG[a,b], DSGG, a)
8 end
9 End DownDistillation
```

Figure 1: Pseudo-code of the DownDistillation procedure

Additional functions used are introduced here after.

- The following procedure makes use of the function **OKTransformation(p,q,GG)** which specifies the conditions for p and q to be merged. It is true iff the following conditions are satisfied:
  - p and q are copies of each other;
  - p and q are involved in some anti-pattern (A) or (B) in GG;
  - for any autonomous subgraph G' of GG, every time p appears in G', q appears in G' too. This last condition ensures us that we do not remove an anti-pattern by a transformation that would make an SP-graph becoming non-SP.
- The function **SameOrientedPath(p,q,GG)** is true iff there is at least a directed path dp in GG such that p and q belong to dp.
- **Visited** is a function allowing to mark nodes as visited or unvisited.

Note that, for example, in Figure 5 of the main paper, $OKTransformation(9, 11, GG)$ is false.

The next page provides the complete pseudo-code of the **Distill** procedure.

```
1  Distill(IN GG: graph; IN/OUT DSGG: graph; IN x: node)
2  v ← sink(GG);
3  ListRed ← set of reduction nodes of GG;
4  SetAU ← set of autonomous subgraphs of GG;
5  Visited(GG[q, v]) ← false /* set all the nodes of GG[q, v] unvisited */
6  foreach successor p of x in GG do
       /* search for copies of p */
7      if Visited(p) ← false then
8          p1 ← p; flagp ← true;
9          while flagp do
               /* flagp allows to consider all the unvisited descendant of p1 if necessary */
10             Distilled ← false /* Distilled says if some transformation on p1 has been done */
11             foreach successor q of x in GG, such that q ≠ p1 do
                   /* successors of x different from p1 are potentially copies of p1 */
12                 q1 ← q; flagq ← true;
13                 while flagq do
                       /* flagq allows to consider all the unvisited descendant of q1 if necessary */
14                     if Visited(q1)=false and SameOrientedPath(p1,q1, GG)=false then
15                         if OKTransformation(p1,q1,GG)=true then
                               /* q1 is a copy of p1 in some anti-pattern and transformation can be performed */
16                             transformation on DSGG, replacing q1 by mergeq;
17                             flagq ← false; distilled ← true; /* loop on q is stopped */
18                         else
                               /* no transformation has been done on p1 and q1 */
19                             if outDegree(q1) ≠ 1 then
20                                 if there exists a single autonomous subgraph GG[q1, y] in SetAU then
21                                     q1 ← y; /* the loop on q1 is continued with the sink of the unique autonomous
                                           subgraph */
22                                 else
                                       /* there is no autonomous subgraph GG[q1, y] in SetAU or more than one */
23                                     if q1 is a reduction node in Listred then
                                           /* search for anti-patterns from reduction node q1 */
24                                         DownDistillation(GG[q1, v], DSGG, q1, SetAU, ListRed);
25                                         Visited(GG[q1, v]) ← false;
26                                         if outdegree(q1) > 1 then flagq ← false;
27                                     else
28                                         flagq ← false;
                                           /* q1 is not a reduction node or there is no autonomous subgraph GG[q1, y] in
                                              SetAU the loop on q is stopped */
29                                     end
30                                 end
31                                 q1 ← the successor of q1 /* outDegree(q1) = 1 */
32                             end
33                         end
34                         flagq ← false;
35                     end
36                 end
37             end
38             /* while loop on flagp continues*/
39             if distilled then
                   /* if p1 has been merged with some other node then search for anti-patterns from p1 */
40                 DownDistillation(GG[p1, v], DSGG, p1, SetAU, ListRed);
41                 Visited(x, mergeq) ← true; /* set all the nodes on all paths from x to mergeq as visited */
42             else
                   /* p1 has not been merged */
43                 if outDegree(p1) ≠ 1 then
44                     if there exists a single autonomous GG[p1, y] in SetAU then
45                         p1 ← y;
46                     else
47                         flagp ← false;
48                     end
49                 else
50                     let p1 ← the successor of p1 /* outDegree(p1) = 1 */
51                 end
52             end
53         end
54     end
55 end
56 EndDistill
```

Figure 2: Pseudo-code of the Distill procedure