Fiona – Functional Interaction Analysis for Open Workflow Nets

Peter Massuthe
massuthe@informatik.hu-berlin.de

Daniela Weinberg
weinberg@informatik.hu-berlin.de

Humboldt-Universität zu Berlin, Institut für Informatik,
Unter den Linden 6, 10099 Berlin, Germany
http://www.informatik.hu-berlin.de/top/tools4bpel/fiona

February 29, 2008

Abstract

This manual is for *Fiona*, Version 1.0. *Fiona* is a tool to automatically analyze the interactional behavior of open workflow nets (*oWFN*). This manual does not explain how to setup or install *Fiona*. For this information please read the Installation Manual which is part of the distribution or can be downloaded from the *Fiona* website. Last update: February 29, 2008.

Fiona is part of the Tools4*BPEL* project funded by the Bundesministerium für Bildung und Forschung. See http://www.informatik.hu-berlin.de/top/tools4bpel for details.

Contents

1 Overview

1.1 Introduction

Fiona is a tool to automatically analyze the interactional behavior of open workflow nets (*oWFN*) [MRS05]. It provides the following techniques:

- It checks for the controllability of the given net by computing the interaction graph [Wei04].
- It calculates the operating guideline [MRS05] for the net.
- It matches an open workflow net (*oWFN*) with an already existing operating guideline [MRS05].
- It can do various calculations on operating guidelines, including checks for equality and equivalence, computing the product of two operating guidelines, computing the number of characterized services, simulating and filtering.

Fiona uses operating guidelines and oWFNs as its input. oWFNs are the output of the tool *BPEL2oWFN*. Thus, any *BPEL* (*Business Process Execution Language for Web Services*) process can easily be analyzed.
To compute the states of the graph nodes *Fiona* uses the efficient algorithms that were implemented in the model checking tool *LoLA*.
Fiona was developed by Daniela Weinberg, Peter Massuthe, Karsten Wolf, Kathrin Kaschner, Christian Gierds, Jan Bretschneider and Martin Znamirowski. It is part of the Tools4*BPEL* project funded by the German Bundesministerium für Bildung und Forschung.
See `http://www.informatik.hu-berlin.de/top/tools4bpel` for details.

2 Installation

There are two possibilities to get *Fiona* running. Either using the precompiled binaries for a variety of operating systems or compiling the source code yourself. Both, the binaries and the source code can be found in the download section of the website:

```
http://www2.informatik.hu-berlin.de/top/tools4bpel/fiona/
download.html
```

2.1 Using the binaries

Even when using the binaries, *Fiona* will not have its complete functionality unless *GraphViz dot* is installed. In order to create graphical output *Fiona* uses system calls to *dot*, so either *dot* has to be callable in the same form which fiona is called or no graphical output will be created. Not having *dot* installed should not interfere with execution in any other way than lacking graphical output.

2.2 Using the source code

Unpack the the `fiona-2.0.tar.gz` into one directory keeping the archived folder structure. Before fiona can be compiled a number of tools has to be available at the shell. For a detailed list of those tools refer to the installation guide given in the `/doc`-directory. Supposing that all necessary tools are available you can compile fiona by entering the source directory and typing the following commands:

```
automake -i
./configure
make
```

To check if the compilation was succesful you can also type `make check`. The check should pass all tests except for those mentioned in section 10 on page 19.

3 Invoking *Fiona*

Fiona's primary uses are deciding controllability of and building operating guidelines for services given as *oWFN*s. Therefore the standard invocations of *Fiona* are:

- checking controllability: `fiona -t ig inputNet.owfn`
- calculating the operating guideline: `fiona -t og inputNet.owfn`

where `inputNet.owfn` contains an *oWFN* written in the appropriate format 4 on page 10. The option `-t ig` lets *Fiona* generate the interaction graph of the given net. In case the graph's size is not too big, a `png` graphic file is created that shows the interaction graph. Furthermore an output is written on the command line indicating the size of the graph and the statement whether the oWFN is controllable or not.
The option `-t og` lets *Fiona* generate the operating guideline of the given *oWFN*. In case the graph's size is not too big, a `png` graphic file is created that shows the operating guideline. Furthermore an output is written on the command line indicating various statistics of the operating guideline.
Since the start of the *fiona* development process a lot of additional operations that primarily deal with operating guidelines have been included. A short description of those functions is given in this section, for more details on operating guidelines see section 6 on page 18.
For more examples, see subsection 3.2 on page 8.
Fiona can be called without any parameter. In this case, it calculates the interaction graph of the *oWFN*, that is being read from the standard input (`stdin`).
The invocationscheme is as follows:

```
fiona [OPTION]* [FILES]*
```

Note: Every file read will be automatically recognized as an oWFN or operating guideline by its content or throw an error. Fiona supports reading multiple files by the '*' operator, for example:

```
fiona -t og *.owfn
```

3.1 Options

Fiona supports the following command-line options:

Help Function Print an overview of the command-line options and exit.
Command: `--help` or `-h`

Version Information Print the complete version information.
Command: `--version` or `-v`

Debugging Set the debuglevel to a value between 1-5
Command: `--debug = <level>` or `-d <level>`
Parameters:

1	show nodes and dfs information
2	show analysis information (i.e. colors)
3	show information on events and states
4	show more detailed information
5	show detailed information on literally everything

Modus Operandi Selects the `<type>` of action you want fiona to process

Command: `--type=<type>` or `-t <type>`

Parameters:

`og`	generate the operating guidelines for every given oWFN
`ig`	generate the interaction graphs for every given oWFN
`smallpartner`	generate the interaction graph and synthesize a small partner oWFN for every given oWFN
`mostpermissivepartner`	generate the interaction graph and synthesize the most permissive partner oWFN for every given oWFN
`distrubuted`	generate the operating guideline and annotate it for distrubuted controllability for every given oWFN
`match`	check if given oWFN matches with an operating guideline given
`simulation`	check whether the first OG characterizes more strategies than the second one
`filter`	reduces the first OG to the point that it simulate the second OG if possible
`equality`	check whether two OGs characterize the same strategies
`equivalence`	check whether two OGs (given as BDDs) are equivalent
`productog`	calculate the product OG of all given OGs
`isacyclic`	check a given OG for cycles
`count`	count the number of services that are characterized by a given OG
`png`	generate png files from all given of oWFNs

Set message maximum Set the maximum number of same messages per state to `<level>`.
Command: `--messagemaximum = <level>` or `-m <level>`

Reduce IG Apply reduction rules to the generated IG.
Command: `--reduceIG` or `-r`

Reduce Node states Use node reduction (IG or OG) which stores less states per node of IG/OG and reduces memory, but increases time.
Command: `--reduce-nodes` or `-R`

Show additional information Different display options according to the output of operating guidelines.

Command: `--show = <parameter>` or `-s <parameter>`

Parameters:

`allnodes`	show nodes of all colors
`blue`	show blue nodes only (default)
`rednodes`	show blue and red nodes (no empty node and no black nodes)
`empty`	show empty node
`allstates`	show all calculated states per node
`deadlocks`	show all but transient states

BDD Construction enable BDD construction (only relevant for OG) argument `<reordering>` specifies reodering

Command: `--BDD = <reordering>` or `-b <reordering>` Parameters:

0	CUDD_REORDER_SAME
1	CUDD_REORDER_NONE
2	CUDD_REORDER_RANDOM
3	CUDD_REORDER_RANDOM_PIVOT
4	CUDD_REORDER_SIFT
5	CUDD_REORDER_SIFT_CONVERGE
6	CUDD_REORDER_SYMM_SIFT
7	CUDD_REORDER_SYMM_SIFT_CONV
8	CUDD_REORDER_WINDOW2
9	CUDD_REORDER_WINDOW3
10	CUDD_REORDER_WINDOW4
11	CUDD_REORDER_WINDOW2_CONV
12	CUDD_REORDER_WINDOW3_CONV

13	CUDD_REORDER_WINDOW4_CONV
14	CUDD_REORDER_GROUP_SIFT
15	CUDD_REORDER_GROUP_SIFT_CONV
16	CUDD_REORDER_ANNEALING
17	CUDD_REORDER_GENETIC
18	CUDD_REORDER_LINEAR
19	CUDD_REORDER_LINEAR_CONVERGE
20	CUDD_REORDER_LAZY_SIFT
21	CUDD_REORDER_EXACT

On the Fly BDD Construction Enable BDD construction on the fly (only relevant for OG) argument `<reordering>`.
Command: `--OnTheFly = <reordering>` or `-B <reordering>`
Parameters: See *BDD Construction* for possible parameters!

Output prefix Sets a prefix string for all output files.
Command: `--output = <filename prefix>` or `-o <filename prefix>`

No output No output will be generated at all.
Command: `--no-output` or `-Q`

Additional parameters Further modification of the execution.
Command: `--parameter = <parameter>` or `-p <parameter>`
Parameters:

no-png	does not create a PNG file

3.2 Examples

This section features some use cases for *Fiona*:

```
fiona -t og -b 0 -p no-png *.owfn
```

Generates an OG reordered by CUDD_REORDER_SAME and saves the OG in a file named `[oldfilename].og` for every oWFN in the current folder. No graphical output is generated.

```
fiona -t png *.owfn
```

Generates a *graphviz dot* graph of each oWFN in the same folder.

```
fiona -t productog *.og
```

Generates an operating guideline product from all operating guidelines in the same folder.

```
fiona -t match -d 5 toMatch.og toBeMatched.owfn
```

Checks if the oWFN toBeMatched.owfn matches with the toMatch.og operating guideline with all debug information being shown in the process.

4 Input Formats

Fiona can have *oWFN*s and *operating guidelines* as its input. The following subsections describe the corresponding file formats.

4.1 *oWFN* File Format

An open workflow net is a Petri net with an *interface*, i.e. two additional sets of places: *input places* and *output places*. Additionally an open workflow net has a set of final markings. To represent *oWFN*s the *LoLA*-format was extended to implement this class of petri nets. Following is an example for a valid *(*oWFN) file.

```
{ sample oWFN }

PLACES
  INTERNAL
    p1,
    p2;
  INPUT
    a;
  OUTPUT
    x,
    y,
    z;

INITIALMARKING
  p1:  2;

FINALCONDITION
  p2 = 4 AND ALL_OTHER_PLACES_EMPTY;

TRANSITION t1
CONSUME
  p1,
  a;
PRODUCE
  p2:2,
  x,
  y,
  z;
```

4.1.1 Structure of the *oWFN* file format

The structure of a net is determined by its places, transitions and the edges in between of those. In the oWFN-Format edges are not seperately defined, but are part of the transition definitions at the end of the file. An oWFN file consists of 4 sections:

1. place definitions
2. initial marking
3. final markings
4. transition definitions

Comments in the oWFN code can be written in between { and }.

4.1.2 Places

In the `PLACES` section the places of the net are specified. Here we distinguish between `INTERNAL`, `INPUT` and `OUTPUT` places. A `PLACES` section of a net looks like this:

```
PLACES
  INTERNAL
    p1,
    p2;
  INPUT
    a;
  OUTPUT
    x,
    y,
    z;
```

In this example we have two internal, one input and three output places. The names of the places must be disjoint over all three categories, thus a place cannot be an input and an output place at the same time.
After the `PLACES` follow the `INITIALMARKING` and `FINALMARKING` respectively `FINALCONDITION` sections, which are explained later on.

4.1.3 Transitions

After the three first sections the transitions of the net including their attached edges are defined. The fourth section does not have a headline like the first three, but rather one for each transition. A transition definition looks like this:

```
TRANSITION t1
CONSUME
  p1,
  a;
PRODUCE
  p2:2,
  x,
  y,
  z;
```

This example defines a transition named `t1` with six edges. Two that lead from the places `p1` and `a` to the transition and four that lead from the transition to the places `p2`, `x`, `y` and `z`, with the edge to `p2` having a weight of 2. A single place can be present under both `CONSUME` and `PRODUCE` in the same transition, such a structure is called a loop. Any other transitions of the net would just be listed further with the same syntax, thus starting with the `TRANSITION` keyword, followed by the name of the transition followed by the `CONSUME` and `PRODUCE` parts containing places, that must have been defined in the `PLACES` section beforehand. `CONSUME` and `PRODUCE` are allowed to be empty, thus representing a transition which is not in any way connected to the rest of the net.

4.1.4 Initial Marking

After the `PLACES` section comes the `INITIALMARKING` section. Every place which is not listed in the `INITIALMARKING` section is implicitly assumed to have zero tokens. If one wants to create an initial state that differs from all places being empty one has to list places, which are to contain a number of tokens, in the `INITIALMARKING` section. Such a section looks as following:

```
INITIALMARKING
  p1:  2;
```

After this definition the place `p1` would contain two tokens. One can initially mark more than one place, by seperating the places by commata and finishing the section with a semicolon. If a place is listed without a colon and the corresponding number of tokens, the *oWFN* file format implies the place is marked with one token.

4.1.5 Final Markings

A final marking is defined either as a `FINALMARKING` or a `FINALCONDITION`. The main difference is that a `FINALMARKING` can only describe one final state while a `FINALCONDITION` can describe multiple final states.

Final Marking The `FINALMARKING` section follows the same syntax as the `INITIALMARKING` section. Thus one can list a number of places with corresponding numbers of tokens. The final state described by a `FINALMARKING` is meant to be a state in which the marking exactly matches the described one, including not mentioned places as having a token number of zero. In the same way this means, that a `FINALMARKING` can only describe exactly one final state. A final marking looks like this:

```
FINALMARKING
  p2:  4;
```

This would describe a final state in which the place `p2` has exactly four tokens and all other places are empty.

Final Condition The syntax of a `FINALCONDITION` is different from that of a `FINALMARKING` . If one wants to express that the place `p2` has to contain four tokens, the `FINALCONDITION` would look like this:

```
FINALCONDITION
  p2 = 4;
```

This `FINALCONDITION` describes any state in which there are exactly four tokens on the place `p2`. One could have instead used a different relation than equality. The *oWFN* file format supports equal, not equal, lesser, greater, less or equal and greater or equal. The two examples are semantically different. While the examplary `FINALCONDITION` accepts any state with four tokens on `p2` as a final state the examplary `FINALMARKING` only accepts one final state, and that is where `p2` has four tokens and all other places have zero tokens. If one wants to create a `FINALCONDITION` equal to the exemplary `FINALMARKING`, one can use the boolean keywords `AND`, `OR` and `NOT`. The equal definition would then look as follows:

```
FINALCONDITION
  p2 = 4 AND p1 = 0 AND a = 0 AND
  x = 0 AND y =0 AND z = 0;
```

All markings that fullfill the boolean formula are considered a final state of the oWFN. In this case it is exactly the same finalstate, as defined by the `FINALMARKING` with `p2` containing four tokens and all other places zero. It can quickly become a nuisance to write or read such a formula, especially if the *oWFN*s become big. That is why there is a keyword that can gather up most of the information.

```
FINALCONDITION
  p2 = 4 AND ALL_OTHER_PLACES_EMPTY;
```

The meaning of this final condition is exactly the same as of the one mentioned before. If one requires only the internal places of his oWFN to be empty, one can use the keyword `ALL_OTHER_INTERNAL_PLACES_EMPTY`. Requiring the same for the external places only (that are all non-internal or interface places) can be achieved with the keyword `ALL_OTHER_EXTERNAL_PLACES_EMPTY`. All three keywords can be combined at will. Saying

```
FINALCONDITION p2 = 4 AND ALL_OTHER_INTERNAL_PLACES_EMPTY
  AND ALL_OTHER_EXTERNAL_PLACES_EMPTY;
```

is the same as

```
FINALCONDITION p2 = 4 AND ALL_OTHER_PLACES_EMPTY;
```

All three keywords can only be used as the right hand side of a conjunction with another state predicate while the whole conjunction is then again a state predicate. So the rule is

```
statepredicate := statepredicate
  AND ALL_OTHER_PLACES_EMPTY
```

The scope of `ALL_OTHER_PLACES_EMPTY` is the left hand side of state predicate of the above conjunction. All places that are not mentioned in the left hand side predicate are then required to be empty.
For one to decide the scope of `ALL_OTHER_PLACES_EMPTY`, it is crucial that one understands all bracing and precedence rules of state predicate formulas and that one can construct the fully braced form of his formula in his mind! Suppose a different oWFN with places `p1`, `p2`, `p3`, and `p4`. Then the formula:

```
FINALCONDITION
p1 = 1 AND ALL_OTHER_PLACES_EMPTY OR
p2 = 1 AND ALL_OTHER_PLACES_EMPTY;
```

expands to:

```
FINALCONDITION
(p1 = 1 AND p2 = 0 AND p3 = 0 AND p4 = 0) OR
(p2 = 1 AND p1 = 0 AND p3 = 0 AND p4 = 0);
```

because `AND` takes precedence over `OR`.
In the same oWFN the formula

```
FINALCONDITION
p1 = 1 AND ALL_OTHER_PLACES_EMPTY AND p2 = 1;
```

expands to

```
FINALCONDITION
p1 = 1 AND p2 = 0 AND p3 = 0 AND p4 = 0 AND p2 = 1;
```

because `AND` is left associative. So the fully braced form of the original formula is:

```
FINALCONDITION
(((p1 = 1) AND ALL_OTHER_PLACES_EMPTY) AND (p2 = 1));
```

Consequently `(p1 = 1)` is the scope of `ALL_OTHER_PLACES_EMPTY`. Therefore `ALL_OTHER_PLACES_EMPTY` expands to `((p2 = 0) AND (p3 = 0) AND (p4 = 0))`. Putting it all together and removing redundant braces results in

```
FINALCONDITION
p1 = 1 AND p2 = 0 AND p3 = 0 AND p4 = 0 AND p2 = 1;
```

which is the formula that was stated as the expansion in the first place.
As one might have already noticed, this formula is unsatisfiable because it requires `p2` to be empty and marked at the same time. Therefore it is recommended to use `ALL_OTHER_PLACES_EMPTY` only at the very end of an n-ary conjunction (or anywhere along with proper explicit bracing).
The same rules, of course, apply to `ALL_OTHER_INTERNAL_PLACES_EMPTY` and `ALL_OTHER_EXTERNAL_PLACES_EMPTY`.

4.2 File Format OG

Fiona can have the og file format as its input.
The og file consists of 3 sections:

1. `NODES`
2. `INITIALNODE`
3. `TRANSITIONS`

In the first section, all nodes of the graph are specified. Each definition contains the node's name and the boolean annotation of the node. Additionally a color for the node can be provided, which is either red or blue. Red nodes are those that are not part of the final operating guideline, but rather left over from building the operating guideline and have not been deleted yet for diagnosis reasons. Blue nodes are equal to colorless nodes and are part of the actual operating guideline. The different nodes are divided by commata, a semicolon finishes the section.
Every node follows this syntax:

```
nodeName:nodeAnnotation[:nodeColor]
```

In the second section, one of the nodes is determined as *root node* by its name, the section is finished by a semicolon:

```
nodeName
```

In the last section all edges of the graph are specified. The different edges are divided by commata, a semicolon finishes the section. Every edge follows this syntax:

```
sourceNodeName -> targetNodeName:edgeAnnotation
```

Here is an example for a valid og file:

```
NODES
  0 : ((!a)) : blue,
  1 : ((true)) : blue,
  2 : ((final + ?y) * (!b + ?y)) : blue,
  3 : ((final)) : blue;

INITIALNODE
  0;

TRANSITIONS
  0 -> 1 : ?x,
  0 -> 2 : !a,
  2 -> 3 : ?y;
```

5 Checking Controllability

5.1 Interaction Graph

- `ig`
- `smallpartner`
- `diagnosis`

The interaction graph shows the states of the service, treating all inner states with the same further communication behaviour as equal. By analyzing the interaction graph controllability of the service can be efficiently decided. For more information on interaction graphs see [Wei04].

5.2 Operating Guidelines

- `og`
- `mostpermissivepartner`
- `distributed`
- `productog`
- `match`
- `productog`
- `simulates`
- `filter`
- `isacyclic`
- `count`

The operating guideline of a service characterizes every possible behaviour of the service's environment that leads to a correct execution of the service, thus its termination. Any behaviour of an environment, that is a subgraph of the operating guideline, includes the root node and fullfills every nodes boolean annotation is a strategy for this service. For more detailed information on operating guidelines refer to [MRS05].

10 Limitations and Bugs

10.1 Limitations

The current version of *Fiona* has several limitations:

Number of Strategies The number of strategies, which are determined by the given og can only be calculated if the og is acyclic and deterministic. It will also hold with a warning in case the calculation becomes to ressourceful. The complexity of the used algorithm is not polynomial.

Sequences oWFNs model asynchronous communication behaviour. Even if the net has already sent a message - therefore put a token on an output place - there is no mechanism to stop the net until the message is consumed. Thus a simple sequence of sent messages quickly results in a large operating guideline, which models every order of the consumption of those sent messages. Examples can be taken from the `/tests/sequence_suite` directory

10.2 Known Bugs

In the current version of Fiona are the following issues:

Problem The test suite in the current release does not pass all tests. The 'samples' and the 'bddtest' tests do not work properly.

Solution The test suite is going to be fixed.

10.3 Reporting Bugs

If you find a bug in *Fiona*, please first check that it is not a known bug listed in 'Known Bugs'. Otherwise please send us an electronic mail to `weinberg@informatik.hu-berlin.de`. Include the version number which you can find by running `fiona --version`. Also include in your message the input `oWFN` process and the output that the program produced. We will try to answer your mail within a week.
If you have other questions, comments or suggestions about *Fiona*, contact us via electronic mail to `weinberg@informatik.hu-berlin.de`.