hibou_label README

A small-step approach to multi-trace checking against interactions

September 2020

1 Introduction

HIBOU (for Holistic Interaction Behavioral Oracle Utility) provides utilities for the analysis of traces and multi-traces collected from the execution of Distributed Systems against interaction models. This document is the README of the implementation (software tool) that accompany the paper.

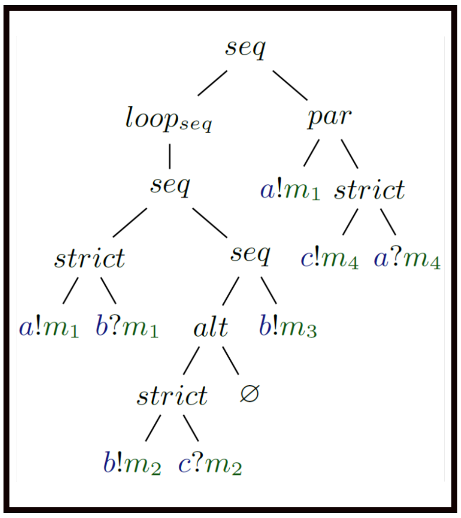
The formal approach is described in the accompanying paper (submitted paper), and verified using the accompanying coq proof.

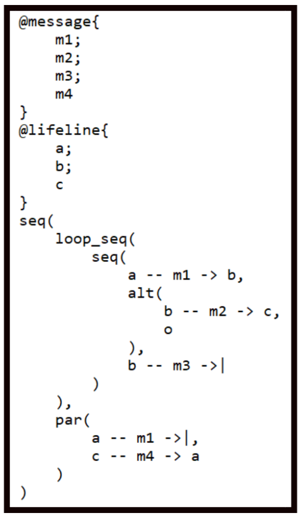
This present version "hibou_label" treats labelled interaction models.

2 Entry language : Interactions (.hsf files)

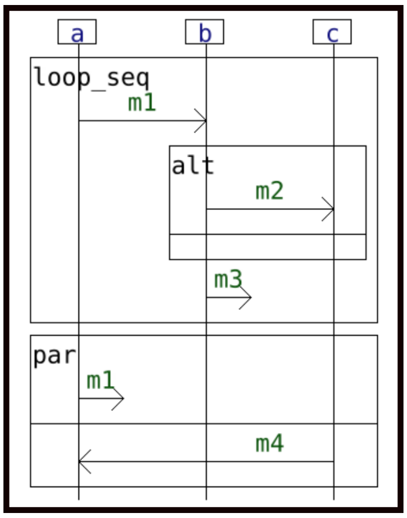
Interaction models are specified with ".hsf" (Hibou Specification File) files. The figure below illustrates:

- on the left the model of the interaction as a binary tree (mathematical model)
- in the middle the encoding using the entry langage of HIBOU (PEG grammar)
- on the right the resulting sequence diagram as drawn by HIBOU





```
@message{
     m1;
     m2;
     m3;
     m4
}
@lifeline{
     a;
     b;
     c
}
seq(
     loop_seq(
          seq(
               a -- m1 -> b,
               alt(
                    b -- m2 -> c,
                    o
               ),
               b -- m3 ->|
          )
     ),
     par(
          a -- m1 ->|,
          c -- m4 -> a
     )
)
```



2.1 Signature Declaration

The signature of the interaction model is declared in the "`@message`" and "`@lifeline`" sections of the ".hsf" file. It suffices then to list the different message names and lifeline names that will be used in the model.

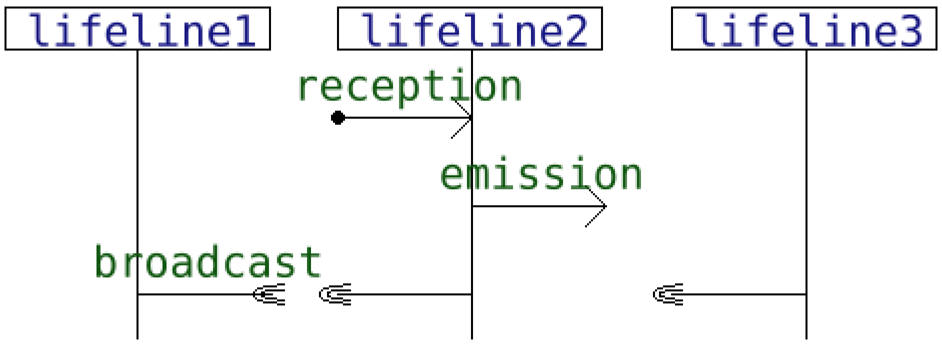
For instance, in the example above (which you can find in the "examples" folder), we have the following:

```
@message{
    m1;
    m2;
    m3;
    m4
}
@lifeline{
    a;
    b;
    c
}
```

2.2 Interaction Term

Interactions are terms of a formal language, that can be specified using a simple and intuitive inductive language. Those terms are build inductively from the composition of basic buildings blocks using specific operators.

2.2.1 Basic building blocks



The most basic interactions can either be:

- the empty interaction, that specify an absence of observable behavior and which is encoded with "o" or "$\varnothing$"
- the reception of a message "m" on a lifeline "a" from the environment, which is encoded with "$m- > a$"
- the emission of a message "m" from lifeline "a" to the environment, which is encoded using "$a - -m- > |$"
- the passing of a message "m" from lifeline "a" to lifeline "b", which is encoded with "$a - -m- > b$" (here message "m" is passed asynchronously from lifeline "a" to "b")
- an asynchronous broadcast, which can be encoded with "$a - -m- > (b, c, \cdots)$"

2.2.2 Operators

Interaction terms can be composed inductively using some operators so as to build more complex interaction terms. We define the following operators:

- "*strict*", "*seq*" and "*par*", which are the "scheduling operators"
- "*alt*" which is the "alternative operator"
- "*loop_strict*", "*loop_seq*" and "*loop_par*" which are "repetition operators"

Scheduling operators. Scheduling operators specify how the execution of different sub-interactions can be scheduled w.r.t. one another. Given sub-interactions "$i1$", "$i2$", ..., and "in", and given a scheduling operator "f", we encode using "$f(i1, i2, ..., in)$" the interaction which results from the scheduling with "f" of those sub-interactions. We allow n-ary expressions in the entry language but the interaction term is constructed as a binary tree in hibou.

- the "*strict*" operator specifies strict sequencing i.e. preceding sub-interactions must be executed entirely before any following interaction can be
- the "*seq*" operator specifies weak sequencing i.e. a strict scheduling is only enforced between actions occurring on the same lifeline
- the "*par*" operator allow any interleaving of the executions of sub-interactions

Alternative operator. The "*alt*" operator specified exclusive alternative choice between the execution of sub-interactions. As for the scheduling operators, we allow n-ary expressions.

Repetition operators. Loop operators are unary operators that represent the repetition of a given sub-interaction, each repeated behavior being sequenced w.r.t. the others using one of the 3 scheduling operators.

For instance "$loop\_seq(m- > a)$" is equivalent to the infinite alternative "$alt(\varnothing, m- > a, seq(m- > a, m- > a), ...)$".

2.2.3 Example

From the previously defined building blocks and operators we can define interaction terms such as the one below:

```
seq(
    loop_seq(
        seq(
            a -- m1 -> b,
            alt(
                b -- m2 -> c,
                o
            ),
            b -- m3 ->|
        )
    ),
    par(
        a -- m1 ->|,
        c -- m4 -> a
    )
)
```

2.3 Process options

Additionally, one can specify in the header of a ".hsf" file a number of options that will then be used if this ".hsf" file is exploited in some algorithmic process.

The "`@analyze_option`" section specifies options to be used when the model is exploited as a reference for the analysis of a multi-trace with the "analyze" command.

The "`@explore_option`" section specifies options to be used when the model is exploited for the exploration of its possible executions with the "explore" command.

We will detail the available options when describing the "analyze" and "explore" commands.

3 Entry language : traces & multi-traces (.htf files)

When a distributed system is executed, we can collect logs on the external interfaces of its sub-system so as to have a trace of the events that were observed during the execution. We consider two kinds of events that may occur on the interface of sub-systems: emissions and receptions of messages.

As the lifelines of an interaction model represent the sub-systems of the distributed system modelled by the interaction, we can understand those collected logs - called traces - as sequences of words "$a!m$" or "$a?m$" - called actions - where "a" is a lifeline and "m" a message.

3.1 Centralized traces

A centralized trace is a sequence of any such action regardless of the lifelines on which they occur. Below is given, in the syntax accepted by HIBOU, such a centralized trace. This trace must be specified in a ".htf" file, which stands for Hibou Trace File. Here, the "[#all]" signifies that the trace is defined over all lifelines.

```
[#all] a!m1.b?m1.b!m2.c?m2
```

3.2 Multi-traces

However, it is not always possible to collect such centralized traces. More often that not, we have a local log/trace for each sub-system of the distributed system. As such, we use the notion of multi-trace.

Multi-traces are sets of traces called its components. Each component is defined over a subset of lifelines called a co-localization that is disjoint to that of any other component.

In the paper version of the algorithm, we restricted co-localizations to singletons in order to simplify the presentation. However, our approach still works when considering the more general case of co-localizations.

On the example below is given an example of ".htf" file which defines a multi-trace composed of 2 components:

- on the co-localization of the 2 lifelines "a" and "b", the local trace "$b!m.a!m$" has been logged
- on the localization of the "c" lifeline, the local trace "$c?m.c?m$" has been logged

```
{
    [a,b] b!m.a!m;
    [c]   c?m.c?m
}
```

Let us note that we can also analyze global traces simply by defining a multi-trace with a single component as is done below. Here we used the "#all" keyword to state that this component is defined over all the lifelines defined in "@lifeline".

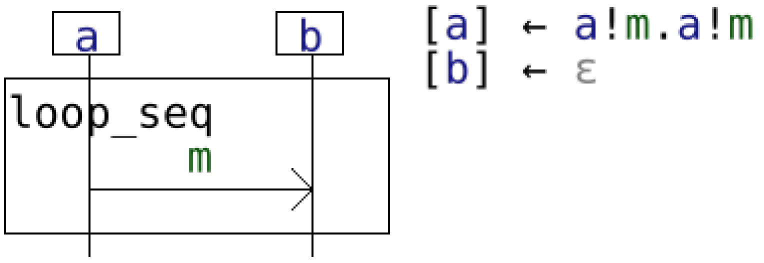
```
{
    [#all] a!m1.b?m1.b!m2.c?m2
}
```

We can also use the "#any" keyword to state that a given multi-trace component is defined over all the lifelines that appear in the subsequent trace definition. For example below is defined a multi-trace that is the same than the one in our previous multi-trace example. The first component is defined over lifelines "a" and "b", and the second over lifeline "c".

```
{
    [#any] b!m.a!m;
    [#any] c?m.c?m
}
```

3.3 Lifelines with no specified trace component

Using a ".htf" file is done with regards to a ".hsf" file which defines the model against which the multi-trace will be analyzed. In the illustration below, the multi-trace on the right "$(a!m.a!m, \epsilon)$" is to be analyzed against the interaction "$loop\_seq(a - -m- > b)$".



If, in a given ".htf" file, traces on some lifelines are not specified, the corresponding lifelines are automatically fitted with the empty trace. As such, w.r.t. the interaction on the left in the illustration above, we can specify the multi-trace "$(a!m.a!m, \epsilon)$" in a ".htf" file either with:

```
{
    [a] a!m.a!m;
    [b]
}
```

Or simply with (for instance):

```
[#any] a!m.a!m
```

Here lifeline "b" has no specified canal. As such, it will be automatically fitted with a dedicated canal containing the empty trace "ϵ".

4 Command Line Interface

The functionnalities of HIBOU are accessed via a Command Line Interface (CLI). For the version presented in this repository, you will have to launch the executable "hibou_label" (or "hibou_label.exe" on Windows OSs) from a terminal.

4.1 Help

The HIBOU executable provides a small documentation about its interface. This can be accessed by typing "`hibou_label help`" or "`hibou_label -h`". It explains that there are 4 sub-commands in HIBOU:

- "draw" (to be used as "`hibou_label draw <.hsf file>`"), which draws as a sequence diagram a given interaction
- "analyze" (to be used as "`hibou_label analyze <.hsf file> <.htf file>`"), which analyze a multi-trace w.r.t. an interaction
- "explore" (to be used as "`hibou_label explore <.hsf file>`"), which computes (partially or totally if possible) the semantics of a given interaction
- "help", which is the present help

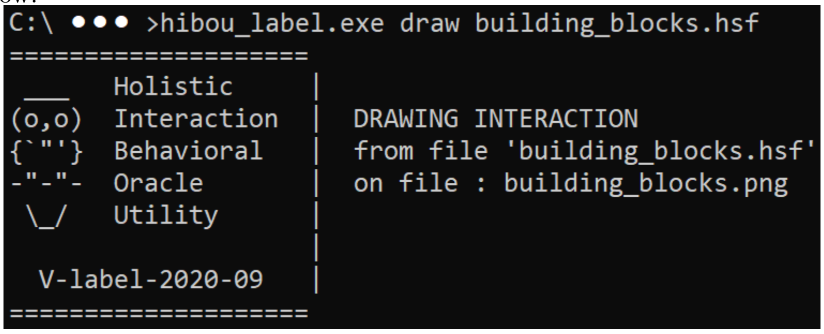
Each sub-command also has its dedicated documentation:

- "`hibou_label draw -h`" provides a small documentation for the drawing utility
- "`hibou_label analyze -h`" provides a small documentation for the "analyze" sub-command
- "`hibou_label explore -h`" provides a small documentation for the "explore" sub-command

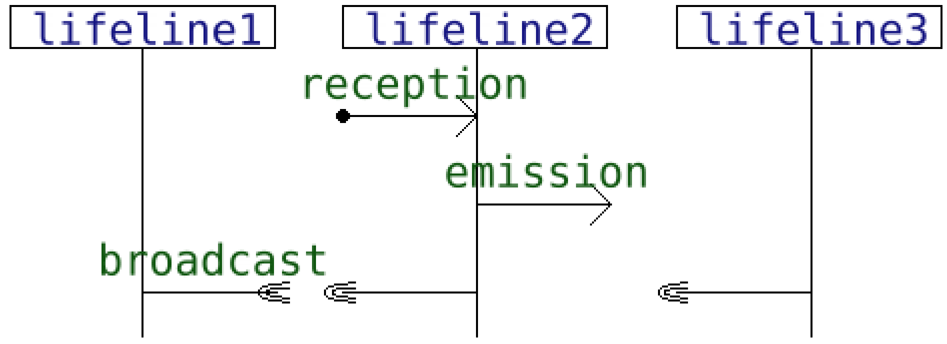
In the following, we provide more details about those sub-commands.

4.2 Draw

Diagrams, such as the one on the previous images can be drawn using the "hibou draw" command as exemplified below.



```
C:\ ••• >hibou_label.exe draw building_blocks.hsf
====================
 ___   Holistic     |
(o,o)  Interaction  |  DRAWING INTERACTION
{`"'}  Behavioral   |  from file 'building_blocks.hsf'
-"-"-  Oracle       |  on file : building_blocks.png
 \_/   Utility      |
                    |
  V-label-2020-09   |
====================
```



4.3 Analyze

The "analyze" sub-command of HIBOU can analyse multi-traces w.r.t. interactions. For any multi-trace and any interaction, it returns a verdict about the conformity of the multi-trace w.r.t. a certain semantics of the interaction.

We consider two semantics:

- "*AccMult*", the semantics which is that of exactly accepted multi-traces (projections of accepted global traces)
- "*SemMult*", the semantics which is that of multi-traces which are obtained from projecting prefixes of accepted global traces

In hibou_label, we implemented accordingly two modes for the analysis:

- the "`accept`" mode, which returns a verdict "*Pass*" if the multi-trace is in "*AccMult*" or "*Fail*" if not
- the "`prefix`" mode, which returns "*Pass*" if the multi-trace is in "*AccMult*", "*WeakPass*" if it is in "*SemMult*" but not in "*AccMult*", and either "*Inconc*" or "*Fail*" if not, as explained in Sec.4.5 of the accompanying paper.

To select one of those modes ("`prefix`" is the default), you can write "`semantics=accept`" or "`semantics=prefix`" in the "`@analyze_option`" section of the input ".hsf" file.

In the following we will only consider example using the "`prefix`" mode given that it is the more advanced one.

We have proven in the accompanying Coq proof that the verdict "*Pass*" is equivalent to the membership of the multi-trace to the "*AccMult*" semantics of the interaction.

4.3.1 Analysing multi-traces

Our approach to analysis consists in consuming one-by-one the head elements of the multi-trace i.e. the actions which are at the beginning of its trace components.

To do so, we compare them with the actions that are immediately executable within the initial interaction model. If there is a match between such an action - called a frontier action - and a trace action, we can compute another interaction model which describes what can happen in the remainder of the execution. We then repeat the process with this new interaction model and the new multi-trace on which we have removed the consumed action.

Consequently, any given analysis opens-up paths which are successions of couples (interaction,multi-trace). Each such path terminates either with:

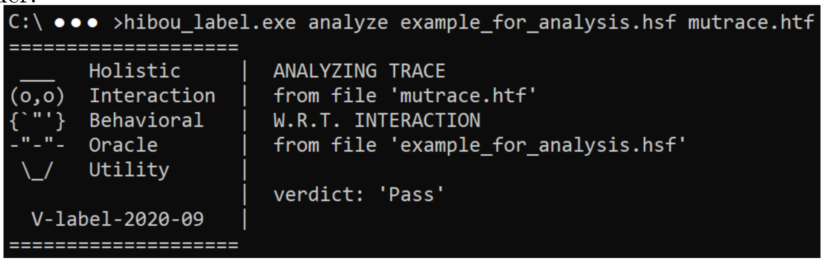
- in "`accept`" mode, either:
  - a "*Cov*" local verdict, when the multi-trace has been entirely consumed and the interaction can express the empty execution (statically verified on the interaction term)
  - or an "*UnCov*" local verdict in the other cases (i.e. either when the consumption of the multi-trace is impossible, or when the multi-trace has been emptied but the interaction cannot express the empty execution)
- in "`prefix`" mode either:

- a "*Cov*" local verdict, when the multi-trace has been entirely consumed and the interaction can express the empty execution
- a "*TooShort*" local verdict, when the multi-trace has been entirely consumed but the interaction cannot express the empty execution
- an "*Out*" local verdict, when no component of the multi-trace can be entirely consumed
- a "*LackObs*" local verdict, when some but not all of the components of the multi-trace can be entirely consumed

From those local verdicts, the global verdict is inferred:

- in "`accept`" mode:
  - "*Pass*" is returned if there exists a path terminating in "*Cov*"
  - "*Fail*" is returned otherwise
- in "`prefix`" mode:
  - "*Pass*" is returned if there exists a path terminating in "*Cov*"
  - "*WeakPass*" is returned if there are no paths leading to "*Cov*" but there exist one terminating in "*TooShort*"
  - "*Inconc*" is returned if there are no paths leading to either "*Cov*" or "*TooShort*" but there is one leading to "*LackObs*"
  - "*Fail*" is returned otherwise

Example 1 Below is given an example analysis, that you can reproduce by using the files from the "examples" folder.



```
C:\ ●●● >hibou_label.exe analyze example_for_analysis.hsf mutrace.htf
===================
  ___   Holistic     |  ANALYZING TRACE
(o,o)  Interaction  |  from file 'mutrace.htf'
{`"'}  Behavioral   |  W.R.T. INTERACTION
-"-"-  Oracle       |  from file 'example_for_analysis.hsf'
 \_/   Utility      |
                    |  verdict: 'Pass'
  V-label-2020-09   |
===================
```

The analysis of the multi-trace specified in the "mutrace.htf" file against the interaction specified in the "`example_for_analysis.hsf`" file yields the "*Pass*" global verdict.

For this analysis we used the following options, declared in the "`@analyze_option`" section of "`example_for_analysis.hsf`".

```
@analyze_option{
    loggers = [graphic=svg];
    semantics = prefix;
    strategy = DFS;
    goal = Pass
}
```

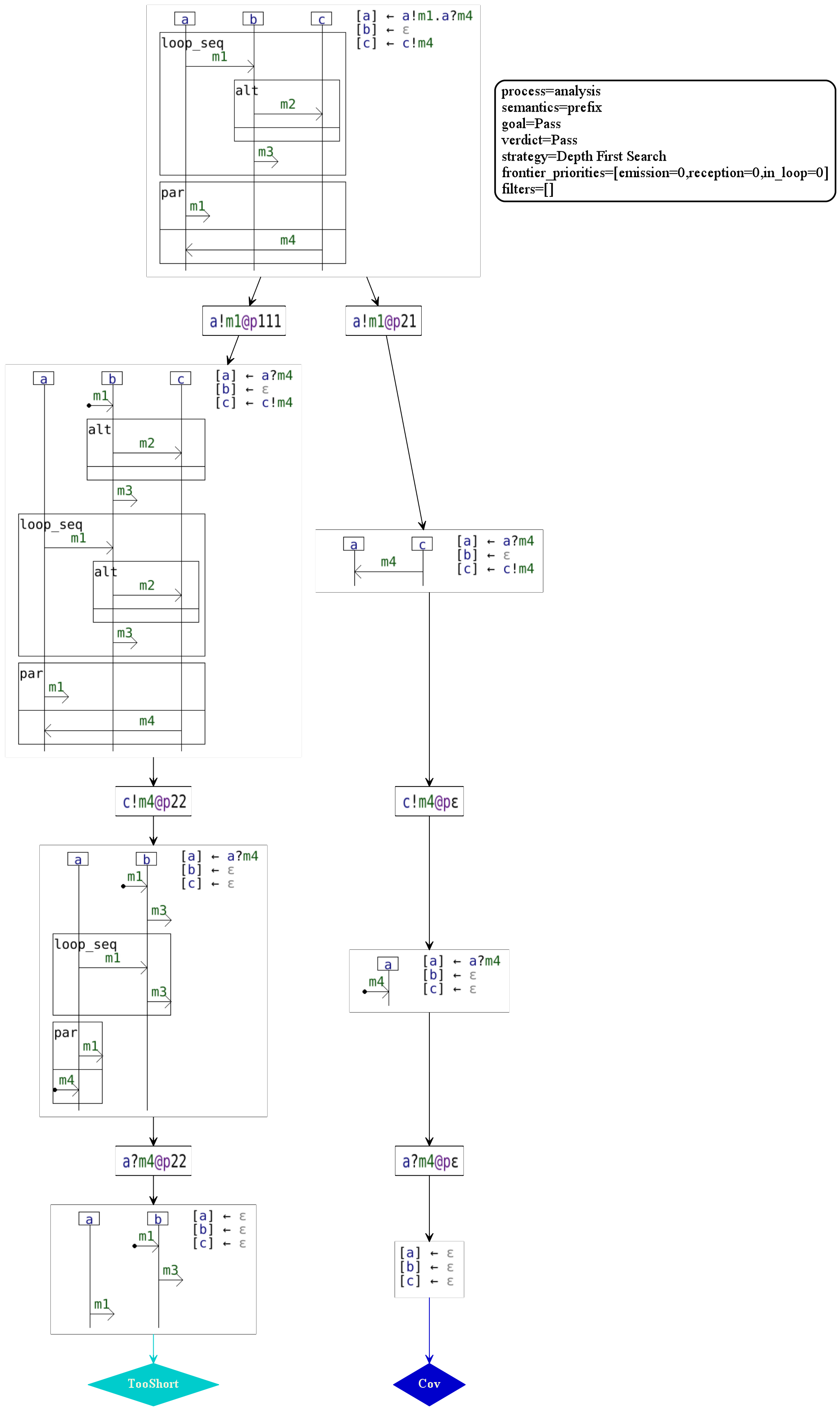
We can specify that we want algorithmic treatments of this ".hsf" file to be logged with the "`loggers`" attribute. In this build only a "`graphic`" logger exists. It will create an image file (with the same name as the ".hsf" file) describing the treatment. The generation of this image requires the graphviz tool to be installed (`https://www.graphviz.org/download/`), and the "dot" command to be in the "PATH" environment variable. The output of the graphic logger can either be a .png file (specified with "`graphic=png`") or a .svg file (specified with "`graphic=svg`"), which requires cairo to be installed.

As explained earlier, we will use the "`prefix`" semantics to analyze the multi-trace.

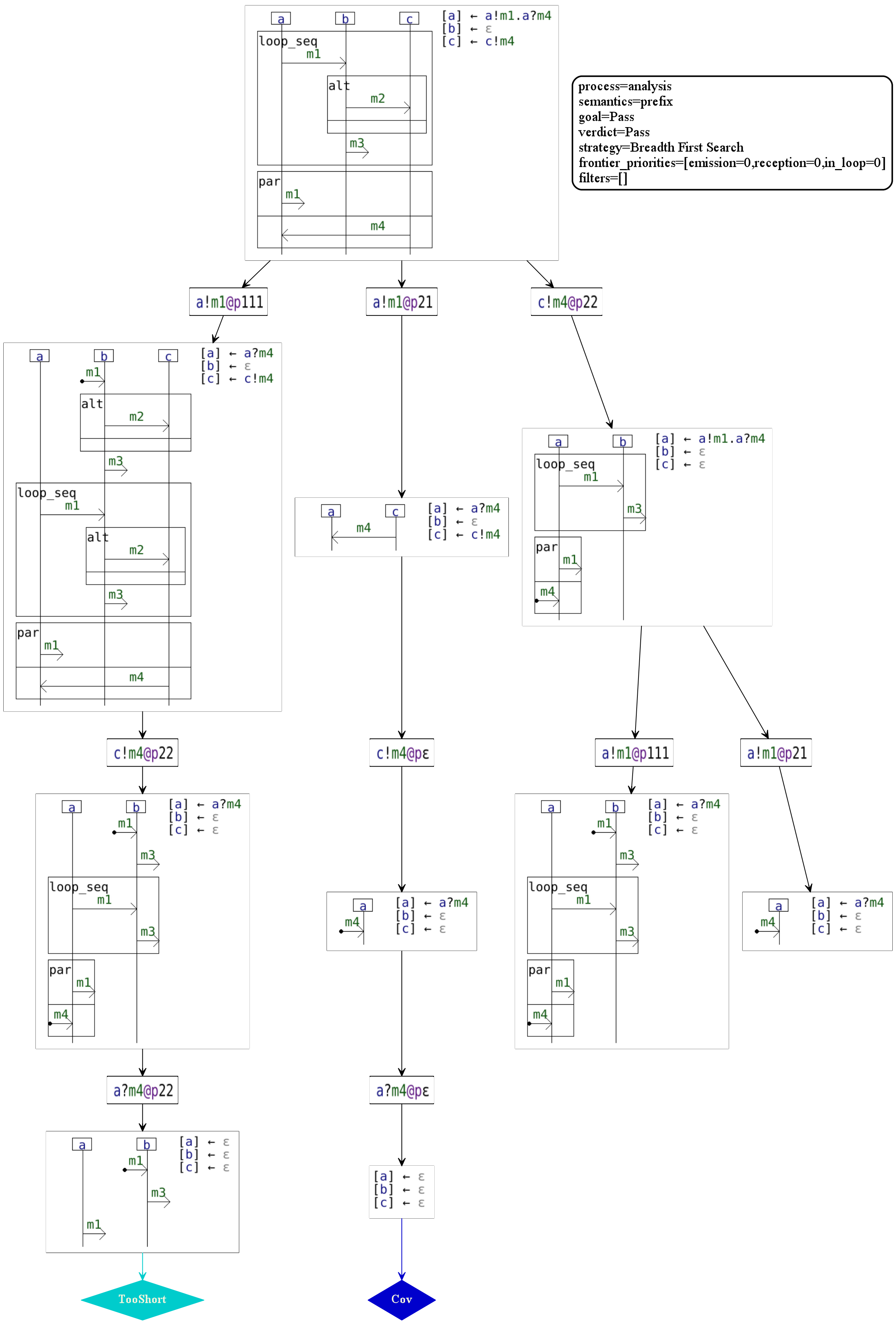
A search strategy: either Breadth First Search (BFS) or Depth First Search (DFS) can be specified using the "`strategy`" option. Indeed, for any given (interaction,multi-trace) couple, several matches may be evaluated, leading to several other couples (interaction,multi-trace). We can then explore those child nodes and their children using any classical search heuristic.

We can also specify the verdict which will be the goal of the analysis, meaning that the analysis will stop once a verdict that is greater or equal to the goal is found. This is done using the "`goal`" option. For instance, if we set the goal to "*Pass*" then, the analysis will stop either when a "*Cov*" is found, or when all paths have been exploited. If we set the goal "$WeakPass$", it will then suffice to find either a "*Cov*" or a "$TooShort$".

The options specified here allowed us, in the case of this example, to quickly find a path that consumed the entire multi-trace and we did not need to explore further executions of the initial interaction model. For instance, you can see on the image below (generated by the "`graphic`" logger) that we did not explore the branch starting with the execution and consumption of "$c!m4$".

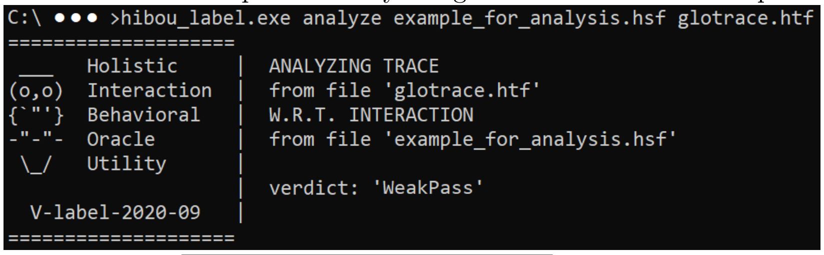

process=analysis
semantics=prefix
goal=Pass
verdict=Pass
strategy=Depth First Search
frontier_priorities=[emission=0,reception=0,in_loop=0]
filters=[]

Let us note that we would have had the following if we had used the Breadth First Search strategy:

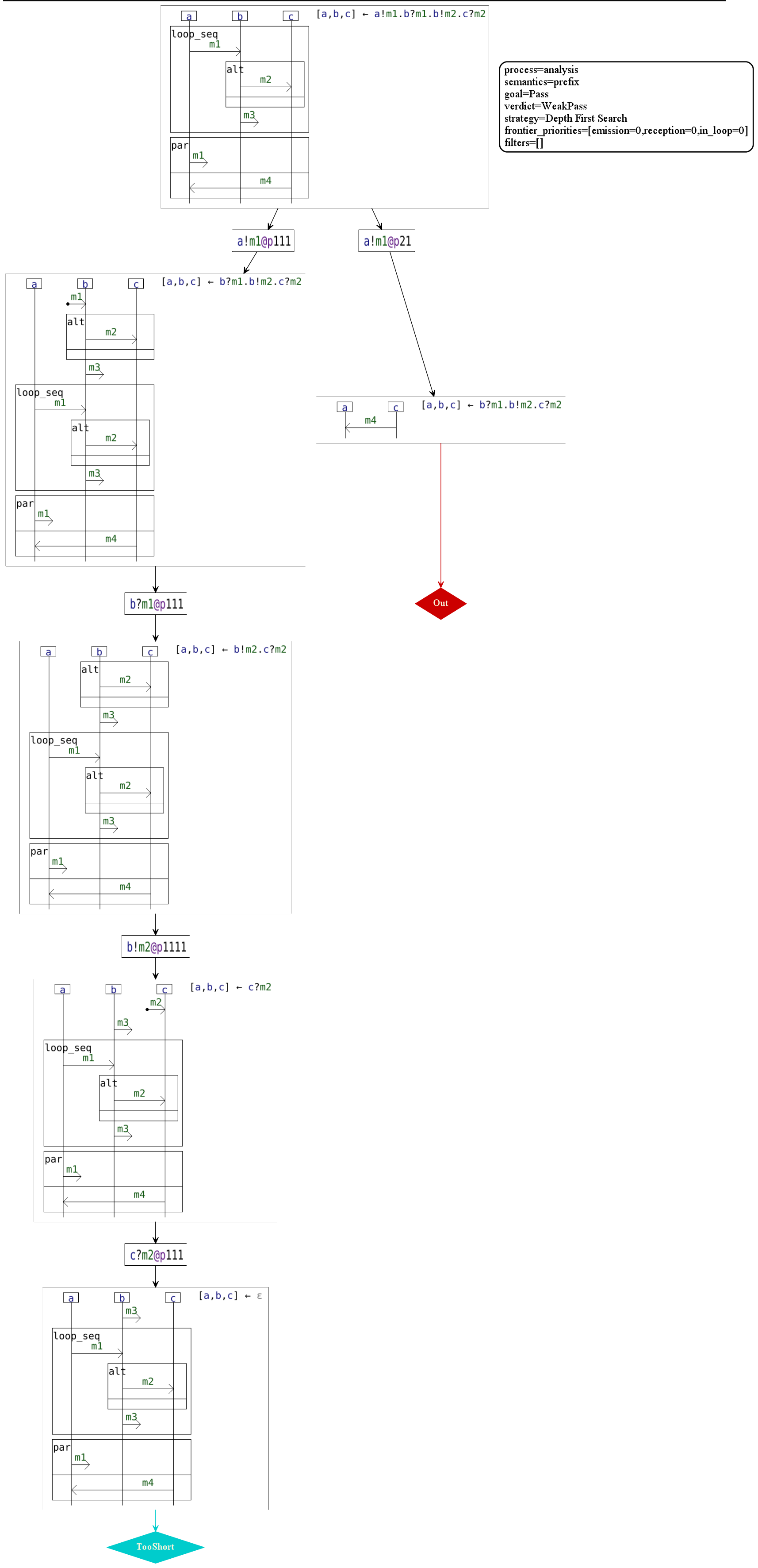


process=analysis
semantics=prefix
goal=Pass
verdict=Pass
strategy=Breadth First Search
frontier_priorities=[emission=0,reception=0,in_loop=0]
filters=[]

Example 2 Below is another example, this time of the analysis of a global trace, and yielding the "*WeakPass*" verdict. You can also reproduce it by using the files from the "examples" folder.

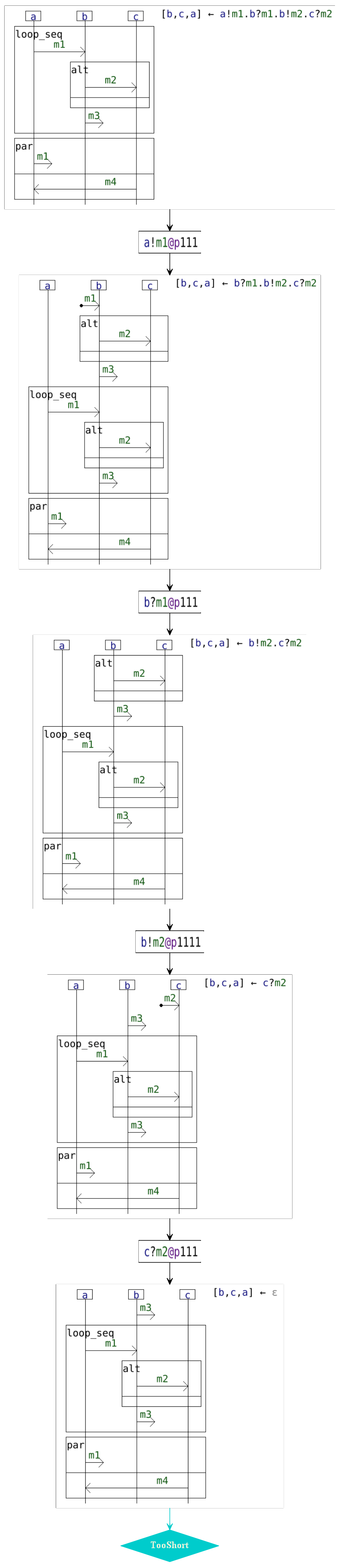


```
C:\ ••• >hibou_label.exe analyze example_for_analysis.hsf glotrace.htf
====================
 ___    Holistic      |  ANALYZING TRACE
(o,o)   Interaction   |  from file 'glotrace.htf'
{`"'}   Behavioral    |  W.R.T. INTERACTION
-"-"-   Oracle        |  from file 'example_for_analysis.hsf'
 \_/    Utility       |
                      |  verdict: 'WeakPass'
  V-label-2020-09     |
====================
```



```
process=analysis
semantics=prefix
goal=Pass
verdict=WeakPass
strategy=Depth First Search
frontier_priorities=[emission=0,reception=0,in_loop=0]
filters=[]
```

Let us note that, if we change the goal to "*WeakPass*" in the options, the branch on the right is not explored in the analysis because the analysis stops once the goal verdict (or a verdict that is stronger) is reached.



process=analysis
semantics=prefix
goal=WeakPass
verdict=WeakPass
strategy=Depth First Search
frontier_priorities=[emission=0,reception=0,in_loop=0]
filters=[]

Example 3 The child nodes of a given couple (interaction,multi-trace) correspond to the execution of a frontier action of the interaction which match a head action of a trace component of the multi-trace. The default order in which those matches are exploited is that of the lexicographic order of the positions of the matching frontier actions within the interaction.

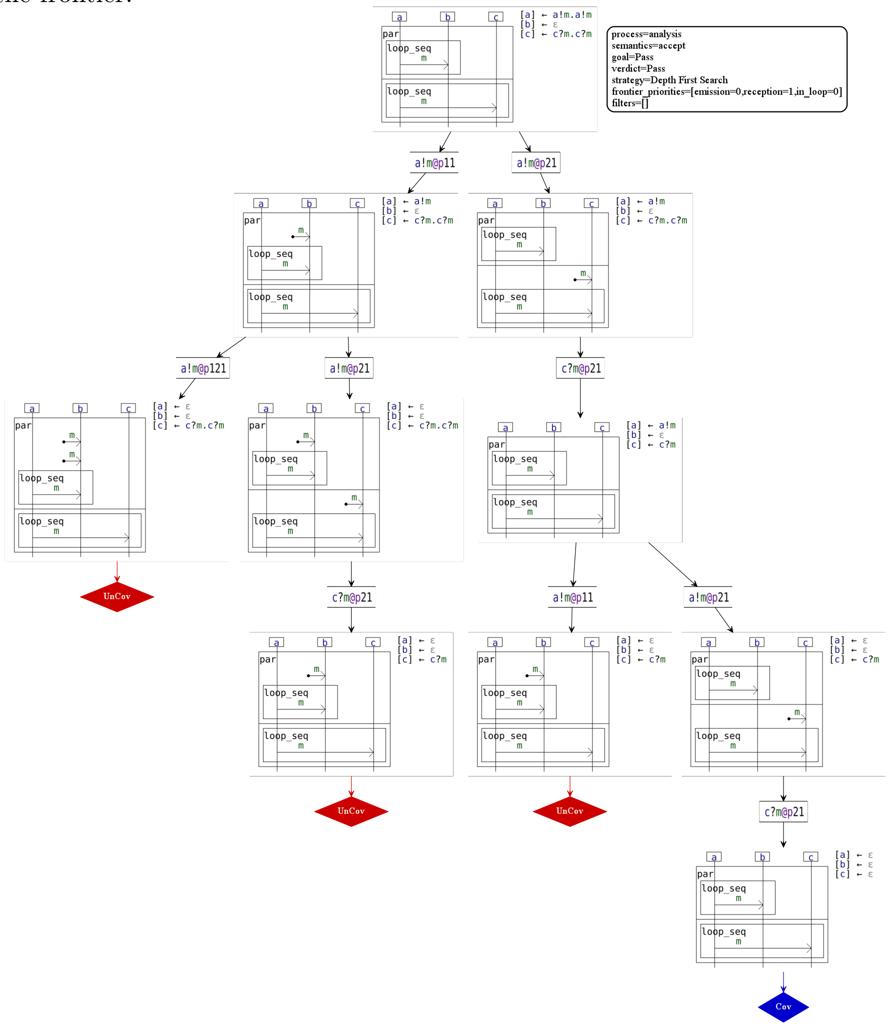
For instance, if there are matches at positions 11, 12, 21 and 22 of the interaction, then the child nodes will be explored in that order.

However, we can reorganize the order with which the matches (and therefore frontier) are explored. To do so, we use the "`frontier_priorities`" option, for instance, as follows:

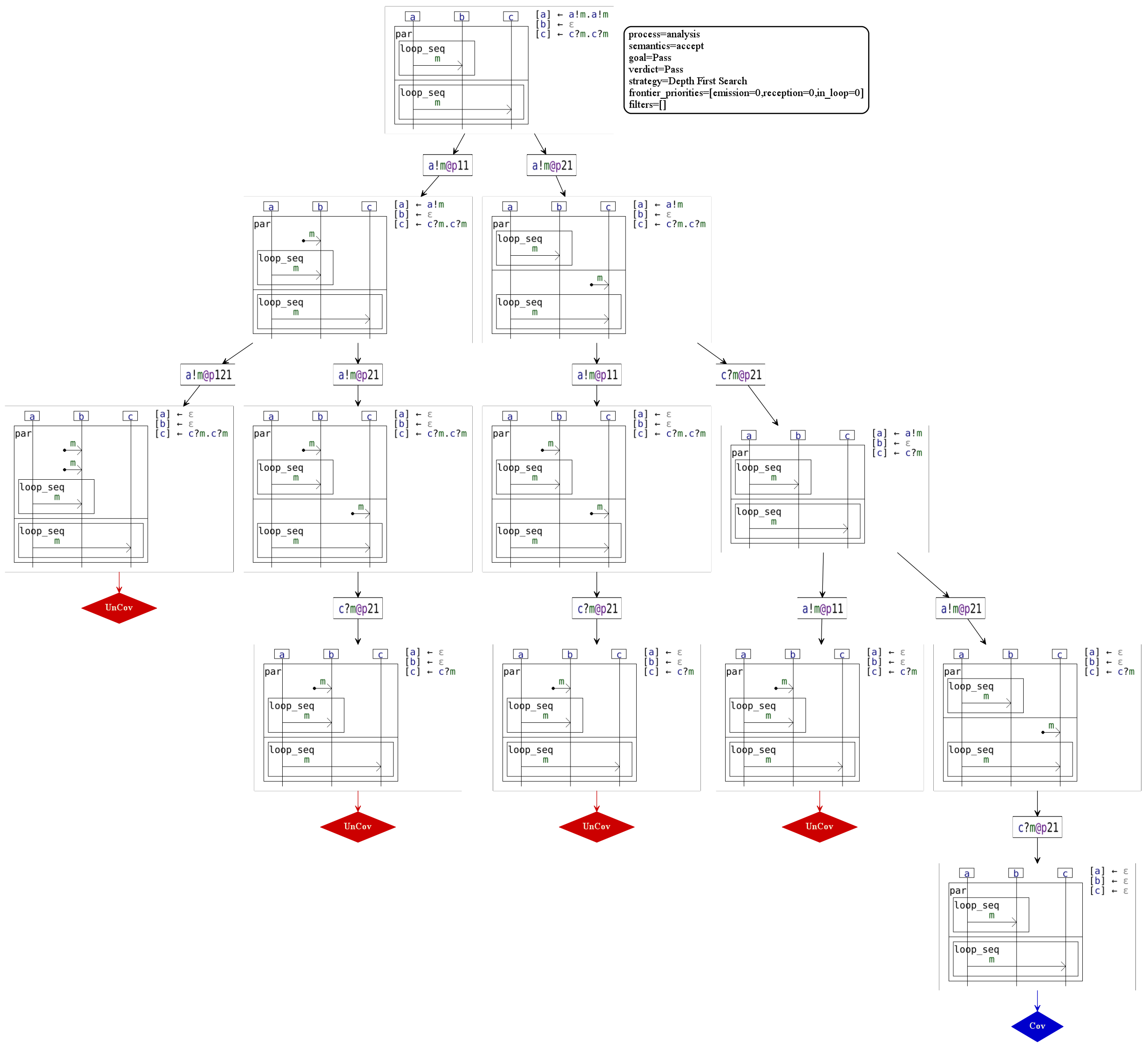
```
@analyze_option{
    semantics = accept;
    strategy = DFS;
    loggers = [graphic=svg];
    goal = Pass;
    frontier_priorities = [reception=1]
}
```

Here we prioritize the evaluation of receptions found in the frontier. For instance if the matches are at positions 11 and 12 but the action at position 11 is an emission whereas the action at position 12 is a reception, then, with those options, the match at position 12 will be exploited before the one at position 11.

Below is illustrated the analysis of a multi-trace using a DFS strategy while prioritizing the evaluation of receptions in the frontier.



If we had not prioritized receptions, we would have had the following (still using DFS):


process=analysis
semantics=accept
goal=Pass
verdict=Pass
strategy=Depth First Search
frontier_priorities=[emission=0,reception=0,in_loop=0]
filters=[]

Prioritizing the evaluation of certain frontier actions may be advantageous (so as to find more quickly a path that consumes the multi-trace) in certain cases. In this case, the model contains a repeating pattern (with a loop), which beginning can, at each iteration be interpreted differently in the model. Here, prioritizing receptions (in this case, it would be the same as de-prioritizing actions in loops with "`frontier_priorities = [loop=-1]`"), makes so that the analysis prefers to "finish" the evaluation of an instance of a loop before instantiating another one.

The gain in "performances" can be assessed using the number of nodes necessary to obtain the "*Pass*" verdict (i.e. finding a "*Cov*" node). Below is a table giving this number of nodes for respectively 1, 2, 3 and 4 repetition of the behavior $(a!m, c?m)$ and for cases where no action are prioritized (first row) and receptions are prioritized (second row).

	1	**2**	**3**	**4**
no	4	12	39	145
`reception=1`	4	10	27	89

The model file ("example_priority.hsf") and trace file ("example_priority_2.htf") for this example are also in the "examples" folder.

4.4 Explore

The "explore" command of HIBOU can generate execution trees which illustrate the semantics of a given interaction model.

Below is given an example exploration that you can obtain by typing "`hibou_label.exe explore example_for_exploration_1.hsf`" with the files from "examples" folder.

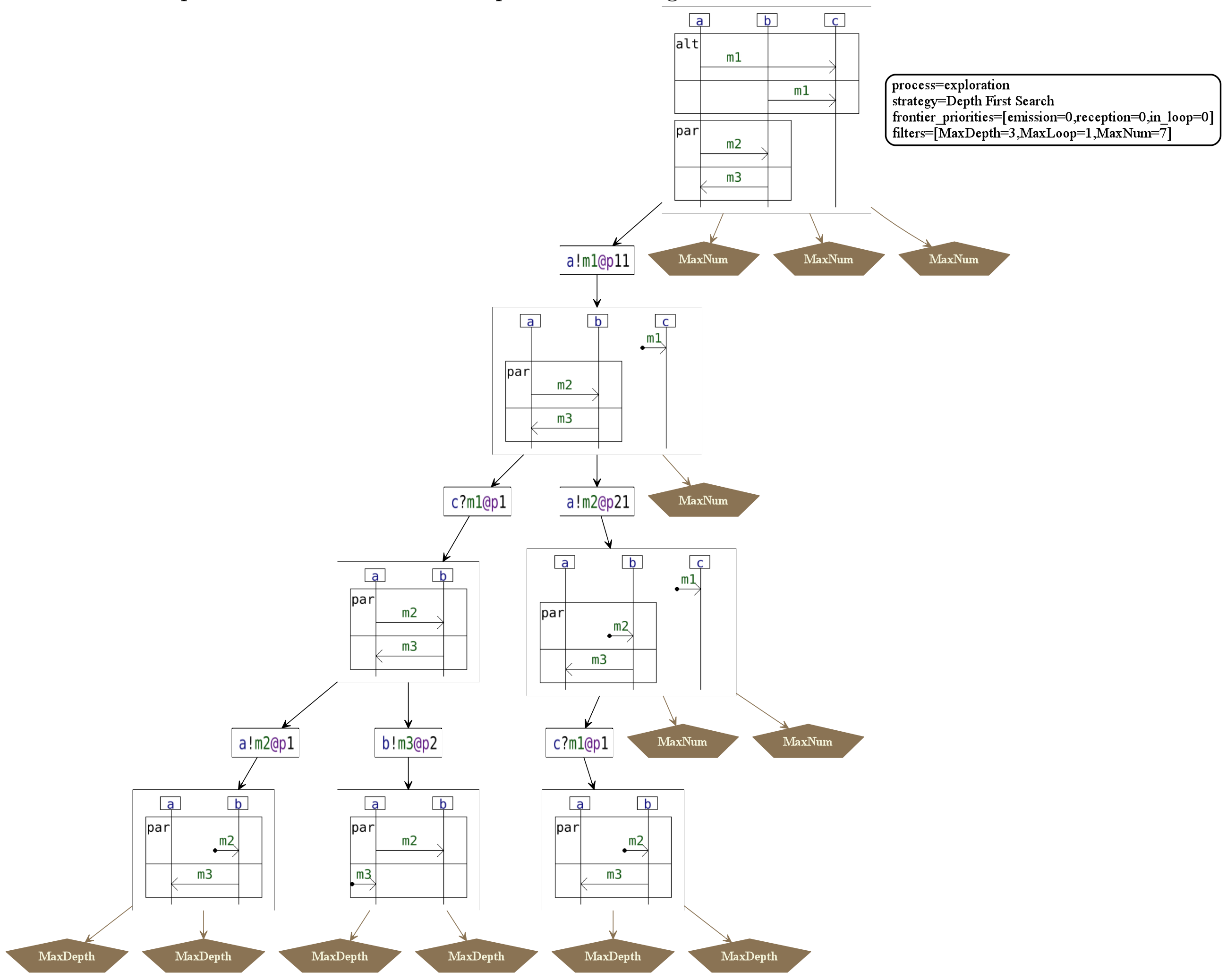
Here we used the following options for the exploration:

```
@explore_option{
    strategy = DFS;
    loggers = [graphic=svg];
    pre_filters = [ max_depth = 3,
                    max_loop_depth = 1,
                    max_node_number = 7 ]
}
```

As you can see, we can specify a number of filters that will limit the exploration of graphs in algorithmic treatments in different ways:

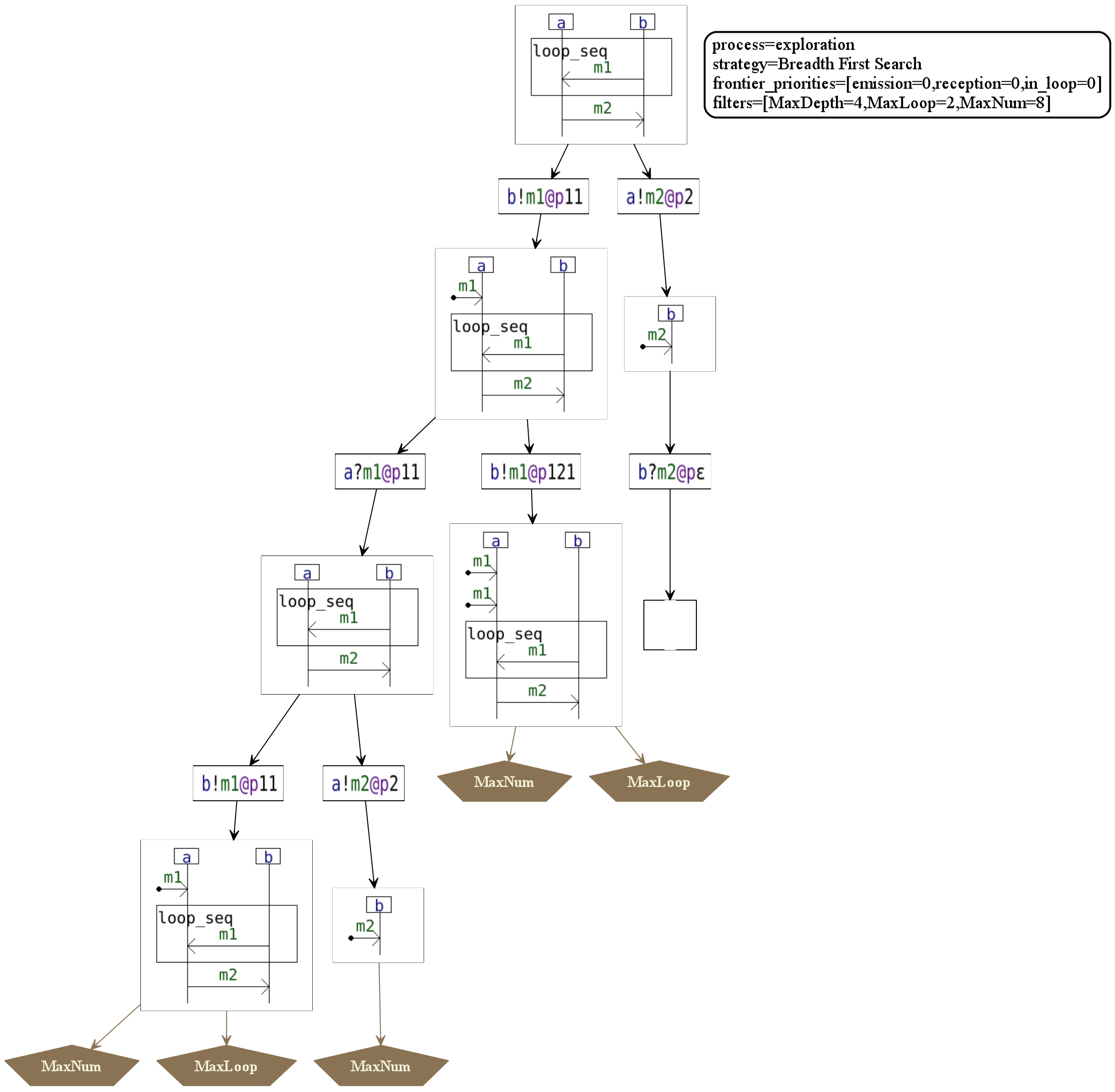
- "`max_depth`" limits the depth of the explored graph
- "`max_loop_depth`" limits the cumulative number of loop instances that can be unfolded in a given execution
- "`max_node_number`" limits the number of nodes in the explored graph

Although you can also specify those filters for the "analyze" command, it is not recommended, given that it might prevent the consumption of the multi-trace and produce a wrong verdict.



process=exploration
strategy=Depth First Search
frontier_priorities=[emission=0,reception=0,in_loop=0]
filters=[MaxDepth=3,MaxLoop=1,MaxNum=7]
alt
par
m1
m2
m3
a
b
c
a!m1@p11
c?m1@p1
a!m2@p21
a!m2@p1
b!m3@p2
MaxNum
MaxDepth

And here is a second example that you can obtain by typing "`hibou_label.exe explore example_for_exploration_2.hsf`" with the files from "examples" folder.



```
process=exploration
strategy=Breadth First Search
frontier_priorities=[emission=0,reception=0,in_loop=0]
filters=[MaxDepth=4,MaxLoop=2,MaxNum=8]
```

5 Requirements / Dependencies

You can directly use the provided binary "`hibou_label.exe`".

So as to generate the images of the graphs, you will need to have graphviz installed on your system. Graphviz is available at `https://www.graphviz.org/download/`. The "dot" command provided by Graphviz must be in your "PATH" environment variable.

6 Examples

All the examples in this README are provided in the "examples" directory as well as the commands used to generate the images above.