Pseudo-merge prototype manual

Mathieu Sassolas

December 1, 2009

1 Foreword

This document presents a prototype tool that implements in Ocaml the algorithm of *pseudo-merge* introduced in [1]. All terms related to this approach marked with "†" are defined in this document. The tool's aim was more to help us compute pseudo-merges† of MTSs† than for real users to actually merge real-world models. Thus both the data structure and the programs were seeking implementation rapidity rather than performance. In that sense, no graphical user interface was developed either. The interface to the use is discussed in Section 3.

2 Installing

2.1 Requirements

The tool operates through command line, so such an interface, along with basic tools (such as `make`) is needed. It was tested on GNU/Linux and MacOS, and should work on Windows through CygWin at least. Since the prototype is written in Ocaml [2], this program is needed to compile and run it. In addition, one might want to use this prototype in conjunction with MTSA [3, 4].

2.2 Compiling

Simply decompress the archive containing the source and run `make compile`. This operation will build two programs. The first is PseudoMerge which implements the pseudo-merge algorithm. The second is FspTranslator which translates an FSP model into a model in our internal format. The latter is meant to have models in the internal format in order to use them in the interactive interface of OCaml.

3 Input/Output syntax(es)

3.1 Input format

The input to the programs are MTSs in FSP-like specifications. For example, the MTS of Figure 1(a) is specified by the FSP model of Figure 1(b). This syntax is the one given by MTSA in its "Transition" tab. Therefore it is possible to use all the power of MTSA to generate models from more abstract specifications (parallel composition, fluents...).

Each model should be in a separate file to be used by `PseudoMerge`, while `FspTranslator` reads models from standard input. For example:

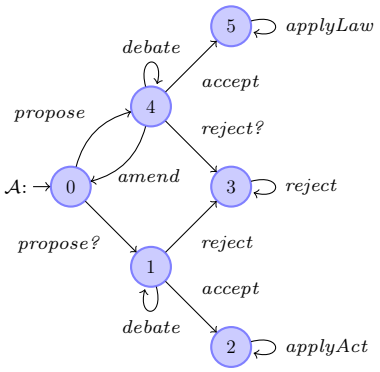
```
./PseudoMerge LawMakingBlue.trans LawMakingRed.trans

./FspTranslator < LawMakingBlue.trans
```

Files `LawMakingBlue.trans` and `LawMakingRed.trans` are provided as examples in the archive.

3.2 Output format

The output format of the `FspTranslator` is an OCaml variable definition. Therefore, they are more intended to be used in Ocaml. For example, the command



(a)

```
Process:
BlueModel
States:
6
Transitions:
BlueModel = Q0,
Q0 = (propose? -> Q1
  |propose -> Q4),
Q1 = (debate -> Q1
  |accept -> Q2
  |reject -> Q3),
Q2 = (applyAct -> Q2),
Q3 = (reject -> Q3),
Q4 = (amend -> Q0
  |reject? -> Q3
  |debate -> Q4
  |accept -> Q5),
Q5 = (applyLaw -> Q5).
```

(b)

Figure 1: A MTS model and its FSP syntax

```
cat LawMakingBlue.trans LawMakingRed.trans
                                    | ../Heuristics/FspTranslator > LawMaking.ml
```

will create a file containing definitions such as

```
let blueModel = {
  alphabet = [|"propose";"debate";"accept";"reject";"applyAct";"amend";"applyLaw"|];
  states = 6;
  transitions = [|
  (* On action 'propose'. *)
    [|[|False;Maybe;False;False;True ;False|];
      [|False;False;False;False;False;False|];
      [|False;False;False;False; ...
    .
    .
    .
};;
```

From these models, or directly with `PseudoMerge`, a FSP specification readable to MTSA can be produced.

```
BlueModel__RedModel__    = (propose -> BlueModel__RedModel__3
                           |propose -> BlueModel__RedModel__2
                           |propose -> BlueModel__RedModel__1),
BlueModel__RedModel__1   = (amend_1 -> ...
.
.
.
||PseudoMerge__BlueModel__RedModel = (BlueModel__RedModel__).
```

Since MTSA produces graphical representation, one can observe a pseudo-merge graphically rather than textually.

Models in the internal representation can also be printed in other formats such as a pgf/Tikz that can be interpreted by LaTeX, although states are aligned and almost certainly require work on layout before the model can be readable.

Another use of `PseudoMerge` is to find a distinguishing property† from a given boundary disagreement transition†. This part is experimental and may not always work for theoretical reasons [1]. The transitions must be specified by ``(2,accept_1,6)'' (the quotes are necessary for

the sake of the shell) where the numbers indicate the state of the pseudo-merge. The transition is suffixed with `_1` to denote that it is required† in the first model fed to the program and prohibited† in the second one. Hence one should first run the program without specifying a transition in order to determine what state numbers have been attributed by the algorithm. Unfortunately, the graphical rendering of MTSA does not keep the order in which the states are specified (since they are not actually states for the FSP specification). The result will be a propositional μ-calculus† property in plain text and LaTeX(user-defined macro `\senext` produces $\langle . \rangle$ while `\sanext` produces $[.]$).

```
/*  Distinguishing mu-formula
<propose>(<accept><applyLaw>t ^ <amend>t)
\senext{propose} (\senext{accept} \senext{applyLaw} \mathbf{t}
                                                \wedge \senext{amend} \mathbf{t})
*/
```

3.3 Ocaml interface

Models produced by `FspTranslator` and the ones directly specified in Ocaml can, as evoked above, be directly used (*i.e.* interpreted, pseudo-merged, "pretty"-printed, ...) through the various functions defined in the back-end of the program, namely the file `mts.ml`. The use of (almost) each function is explained before its code. For example, one might want to define and display the Tikz code for a pseudo-merge of models BlueModel and RedModel. The corresponding Ocaml code is (`consist` is a consistency relation computed beforehand, it is likely to be an empty list).

```
let pseudoMerge = pseudo_mergeMtsGen true "_1" "_2" blueModel redModel consist []
in
  printMtsTikz true pseudoMerge;;
```

This example produces a model in which each state of the pseudo-merge is named after the states it came from. Although more powerful, this use of the prototype is rather complex, and is recommended only in such limited cases.

4 Support and bug reports

Although the prototype is not being developed any further, I welcome any questions and bug reports: `mathieu.sassolas@lip6.fr`

References

[1] Mathieu Sassolas. "Exploring Inconsistencies between Modal Transition Systems: Internship Report". Technical report, ENS-Cachan, 2008. `http://pagesperso-systeme.lip6.fr/Mathieu.Sassolas/recherche/papers/ReportM2.pdf`.

[2] OCaml. http://caml.inria.fr/index.en.html.

[3] Nicolas D'Ippolito, Dario Fishbein, Marsha Chechik, and Sebastian Uchitel. "MTSA: The Modal Transition System Analyzer". In *Proceedings of International Conference on Automated Software Engineering (ASE'08)*, pages 475–476, September 2008.

[4] MTSA. `http://www.lafhis.dc.uba.ar/~suchitel/MTSA.html`.