The `sem-sfe` manual

Alessandro Flori

December 15, 2023

Contents

1 Installation

You should first have a working installation of Rust and Cargo, 1.73 and above. This project has not been tested with versions of Rust below 1.73.

Download this project, https://github.com/strang3nt/sem-lmc.git, go into its root folder and execute `cargo run -r` from terminal. The compiled executable should be located in `sem-lmc/target/release`.

2 Usage

The compiled binary is a command line interface. In the following we list and provide an explanation for all the commands and options. The following listing is a correct command line invocation:

```
sem-lmc-cli [OPTIONS] <COMMAND>
```

where `[OPTION]` is a list of flags and `<COMMAND>` is the name of the type of input we are going to feed to the tool.

Options:

-n or –normalize If enabled, the underlying system of fixpoint equations is normalized during the preprocessing phase.

-e or –explain A flag that makes the program print useful information to stdout: the underlying system of fixpoint equations, and the symbolic existential-moves, before and after composition.

2.1 Debug command

The debug has the following structure:

```
sem-sfe-cli [OPTIONS] debug <ARITY>\
<FIX_SYSTEM> <BASIS> <MOVES_SYSTEM> <ELEMENT_OF_BASIS> <INDEX>
```

`<ARITY>`

:A path to a file containing definitions of functions. The file must be formatted as follows: each line contains a string of characters and an integer number. The string represents the name of a function, which is going to be used in the system of fixpoint equations. The integer represents the arity of the function. The names and and or can be declared, but will be ignored.

<FIX_SYSTEM> A path to a file containing the definition of a system of fixed point equations. A function must be an either an and or or function, or it must be specified in the arity file. We are going to give a precise grammar specification in Input file grammar specification.

<BASIS> A path to a file containing all the elements of the basis. Each line must contain a string, which is an element of the basis.

<MOVES_SYSTEM> A path to a file containing the symbolic ∃-moves for the system of fixpoint equations. There must be a symbolic ∃-move for all possible combinations of functions introduced in the file, and basis elements introduced in the file. We give the grammar specification in Input file grammar specification.

<ELEMENT_OF_BASIS> The element of the basis which we want to verify is part of the solution of the system of fixpoint equations.

<INDEX> A number representing the equation, and thus the variable which is going to be substituted by the element of the basis specified. 2.1.1 Input file grammar specification In the following we give the grammar, in EBNF form, for systems of fixpoint equations and symbolic ∃-moves.

⟨*EqList*⟩ ::= ⟨*Eq*⟩ ⟨*EqList*⟩ '`;`' | ⟨*Eq*⟩ '`;`'

⟨*Eq*⟩ ::= ⟨*Id*⟩ '`=max`' ⟨*ExpEq*⟩ | ⟨*Id*⟩ '`=min`' ⟨*ExpEq*⟩

⟨*ExpEq*⟩ ::= ⟨*OrExpEq*⟩

⟨*Atom*⟩ ::= ⟨*Id*⟩ | '`(`' ⟨*ExpEq*⟩ '`)`' | ⟨*CustomExpEq*⟩

⟨*AndExpEq*⟩ ::= ⟨*Atom*⟩ ('`and`' ⟨*Atom*⟩)*

⟨*OrExpEq*⟩ ::= ⟨*AndExpEq*⟩ ('`or`' ⟨*AndExpEq*⟩)*

⟨*CustomExpEq*⟩ ::= ⟨*Op*⟩ '`(`' ⟨*ExpEq*⟩ ('`,`' ⟨*ExpEq*⟩)* '`)`'

⟨*Id*⟩ ::= (a C-style identifier)

⟨*Op*⟩ ::= (any ASCII string)

Notice that the syntactic category AndExpEq has a higher precedence than OrExpEq, this way we enforce the precedence of the operator and over or. Tokens *Id* and *Op* are strings, the latter represents the name of an operator provided by the user. If the goal is to parse μ-calculus formulae, a possible definition for OP would be $Op \in \{diamond, box\}$.

3 The design of `sem-sfe`

3.1 High-level structure

The tool is divided into multiple modules, namely:

1. `sem-sfe-algorithm`,
2. `sem-sfe-cli`,
3. `sem-sfe-pg`,
4. `sem-sfe-mu-ald`.

Module 1. is the core of the project, it contains the local algorithm for verifying solutions for systems of fixpoint equations. Module 2. is the command line interface. Modules 3. and 4. are interfaces to the local algorithm, they take as input some specification language and some verification logic, and they translate such input in a system of fixpoint equations and generate the symbolic ∃-moves.

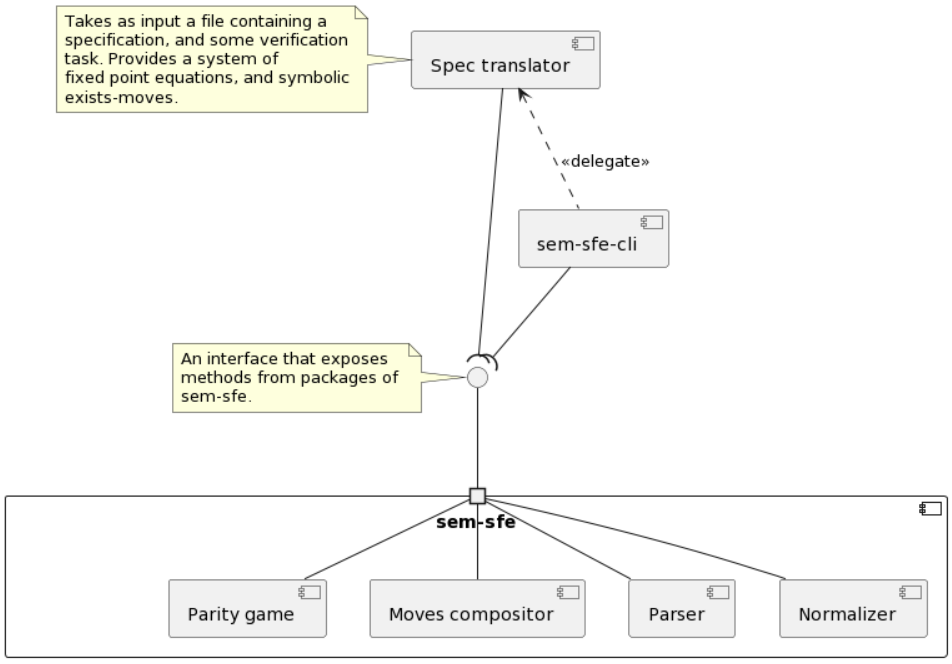

Figure 1: A diagram that represents the design of `sem-sfe`

The diagram 1 represents how the various modules of `sem-sfe` are related. In the diagram, a "Spec translator" represents both

`sem-sfe-pg` and `sem-sfe-mu-ald`. From the diagram we understand that `sem-sfe-algorithm` offers an interface, represented by the ball notation, which is accessed by every other module. The Spec translator module is used by `sem-sfe-cli`: the former is able to take as input a specification file and some verification logic, and provides to the latter a system of fixpoint equations and the symbolic $\exists$-moves, making it possible to run the verification task via the local algorithm.

3.2 The local algorithm module

3.3 How to contribute

4 The mu-calculus module

mu-ald

:

4.1 Aldebaran format

4.2 μ-calculus formulae

4.3 Output

4.4 Tutorial

This is a brief tutorial that provides a few examples. In the following we suppose to be in the terminal emulator, in the path: `sem-sfe-cli/target/release`. The project should be already compiled for release. The repository contains the files we are going to use, under the folder `sem-sfe-cli/tests`.

4.4.1 Parity games

The command:

```
./sem-sfe-cli pg -g ../../tests/parity_games/test_03.gm -n Antarctica
```

will parse the file below, in PGSolver format:

```
parity 4;
0 6 1 4,2 "Africa";
4 7 1 0 "Antarctica";
1 5 1 2,3 "America";
```

```
3 6 0 4,2 "Australia";
2 8 0 3,1,0,4 "Asia";
```

and ask whether if the existential player can win from vertex `Antarctica`.

4.4.2 μ-calculus

5 Parity game module

pg

:

5.1 PGSOLVER format

5.2 Output

6 Comparison and benchmarks