The `sem-sfe` manual

Alessandro Flori

December 15, 2023

Contents

1 The design of `sem-sfe`

1.1 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.

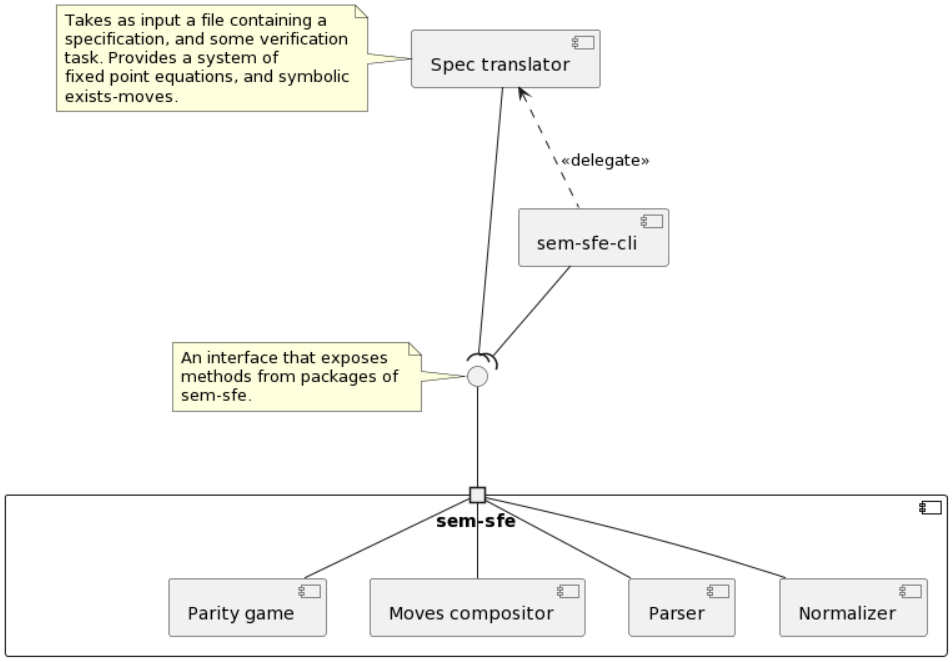

Figure 1: A diagram that represents the design of `sem-sfe`

The diagram 1 represents how the various modules of `sem-sfe` are related. In the diagram, "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 ∃-moves, making it possible to run the verification task via the local algorithm.

1.2 The local algorithm module

1.3 How to contribute

2 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.

To compile this project simply download it from the repository, and run `cargo build -r` from the terminal emulator. The compiled executable should be located in `sem-sfe/target/release`.

3 Usage

This application is a command line interface. An invocation of `sem-sfe` looks like this:

```
sem-sfe-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.

There are 2 possible options, which can be enabled:

-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..

A `<COMMAND>` string is one of the following: `debug`, `pg`, `mu-ald`, followed by their respective inputs. We are going to introduce all these commands in the next sections.

3.1 The `debug` command

The debug command 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 section Input 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 section Input 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.

3.1.1 Input grammar specification

We now give the grammar, in EBNF form, for systems of fixpoint equations and symbolic ∃-moves.

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

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

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

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

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

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

⟨*Id*⟩ ::= (a C-style identifier)

⟨*Op*⟩ ::= (any ASCII string)

The grammar above represents a system of fixpoint equations. 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 $Op \in \{diamond, box\}$.

The following EBNF grammar describes a list of symbolic $\exists$-moves:

$\langle SymMovList \rangle$ ::= $\langle SymMovEq \rangle$ $\langle SymMovList \rangle$ ';' | $\langle SymMovEq \rangle$ ';'

$\langle SymMovEq \rangle$ ::= 'phi' '(' $\langle Id \rangle$ ')' '(' $\langle Num \rangle$ ')' '=' $\langle Disjunction \rangle$

$\langle Conjunction \rangle$ ::= $\langle Atom \rangle$ ('and' $\langle Atom \rangle$)*

$\langle Disjunction \rangle$ ::= $\langle Conjunction \rangle$ ('or' $\langle Conjunction \rangle$)*

$\langle Atom \rangle$ ::= '[' $\langle Id \rangle$ ',' $\langle Num \rangle$ ']' | 'true' | 'false'
| '(' $\langle Disjunction \rangle$ ')'

$\langle Id \rangle$ ::= (a C-style identifier)

$\langle Num \rangle$ ::= $\mathbb{N}$

To parse both grammars we used a parser libray called Chumsky. Chumsky is based on parser combinators, which is a parsing technique that allows for easy to mantain code, and unlike parser generators, no unnecessary boilerplate. The downside of parser combinator is that they usually have a limited support for left recursion, which is why both grammars were built to avoid left-recursion. Indirect left recursion is permitted, but in a limited way.

3.2 The `mu-ald` command

The `mu-ald` command calls the `sem-sfe-mu-ald` module. It produces a fixpoint system and a list of symbolic $\exists$-moves from the given labelled transition system, and μ-calculus formula.

```
sem-sfe-cli [OPTIONS] mu-ald <LTS_ALD> <MU_CALC_FORMULA> <STATE>
```

`<LTS_ALD>` A path to a file describing a labelled transition system in the Aldebaran format, from the CADP toolset. The following link contains a description of the grammar: https://www.mcrl2.org/web/user_manual/tools/lts.html.

`<MU_CALC_FORMULA>` A path to a file containing a μ-calculus formula. The grammar is described in section Mu-calculus formulae.

<STATE> A string which represents a state. Since the Aldebaran specification uses natural numbers as nodes' names, the state must be a number as well. We want to verify whether if it satisfies the property described by the μ-calculus formula.

3.2.1 Mu-calculus formulae

We want to parse the following syntax:

$$\varphi ::= tt \mid ff \mid x \mid \varphi \vee \varphi \mid \varphi \wedge \varphi \mid \mu x.\varphi \mid \nu x.\varphi \mid \langle a \rangle \varphi \mid [a]\varphi$$

With $a \in Act$ and $x \in PVar$. For the same reasons as in section The `debug` command, we designed a grammar that avoids, as much as possible, left recursion. The following EBNF grammar describes a μ-calculus formula.

$\langle Atom \rangle$ ::= '`tt`' | '`ff`' | '`(`' $\langle Disjunction \rangle$ '`)`'
| '`<`' $\langle Label \rangle$ '`>`' $\langle Disjunction \rangle$
| '`[`' $\langle Label \rangle$ '`]`' $\langle Disjunction \rangle$
| '`mu`' $\langle Id \rangle$ '`.`' $\langle Disjunction \rangle$
| '`nu`' $\langle Id \rangle$ '`.`' $\langle Disjunction \rangle$

$\langle Conjunction \rangle$ ::= $\langle Atom \rangle$ ('`&&`' $\langle Atom \rangle$)*

$\langle Disjuction \rangle$ ::= $\langle Conjunction \rangle$ ('`||`' $\langle Conjunction \rangle$)*

$\langle Label \rangle$ ::= '`true`' | $\langle Id \rangle$

$\langle Id \rangle$ ::= (a C-style identifier)

3.3 The `pg` command

The `pg` command uses the `sem-sfe-pg` module, to build a system of fixed point equations and the symbolic $\exists$-moves from a parity game, and verify whether if the given node is winning for player $\exists$ (or player 0, or player Even).

This is a typical command for the `pg` command:

```
sem-sfe-cli [OPTIONS] pg <GAME_PATH> <NODE>
```

<GAME_PATH> A path to a file containing a PGSolver file specification.

<NODE> A string which must refer to the name of the node, if specified in the input file, or to the id of a node.

3.4 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`.

3.4.1 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`.

3.4.2 μ-calculus

4 Comparison and benchmarks