The LCSFE manual

Alessandro Flori

November 26, 2023

Contents

1 Structure of the project

The tool is divided into multiple modules, namely:

- `lcsfe-algorithm`,
- `lcsfe-cli`,
- `lcsfe-common`,
- `lcsfe-pg`,
- `lcsfe-mu-ald`.

Module `lcsfe-algorithm` is the core of the project, it contains an implementation of the local algorithm for verifying solutions of systems of fixpoint equations, over complete lattices. Module `lcsfe-cli` is a command line interface. Modules `lcsfe-pg` and `lcsfe-mu-ald` both use the local algorithm. Modules `lcsfe-pg` and `lcsfe-mu-ald` both use the local algorithm. They take as input some specification language and some verification logic. Then, they translate this input to a system of fixpoint equations, and generate the correct symbolic ∃-moves for their respective operators, after which they call the local algorithm to solve the verification problem, and the output is passed to `lcsfe-cli` to be printed. Module `lcsfe-common` exposes a common interface that `lcsfe-pg` and `lcsfe-mu-ald` use to provide their results to the command line interface module, it avoids circular dependency.

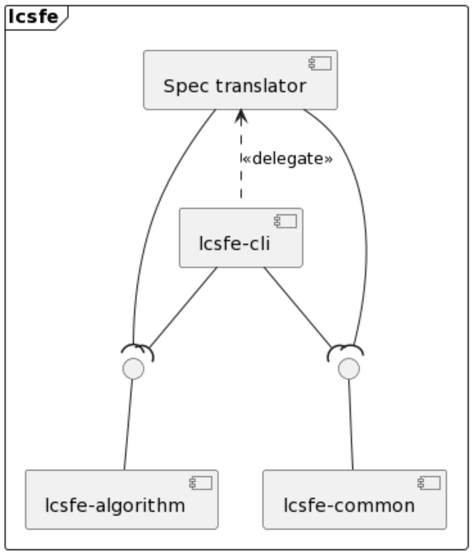

Figure 1: An informal component diagram of `LCSFE`.

Figure 1 represents how the various modules of `LCSFE` are related. In the diagram, Spec translator represents both `lcsfe-pg` and `lcsfe-mu-ald`. From the diagram we understand that `lcsfe-algorithm` offers an interface, represented by the ball notation, which is accessed by every other module. The `lcsfe-common` crate exposes a trait, represented in the diagram as an interface, via the ball notation. The goal of this trait is to uniform the results computed by Spec translator, so that `lcsfe-cli` can easily access and print them via the same common interface. Spec translator module is used by `lcsfe-cli`: the former takes as input a specification file and some verification logic, and provides to the latter the results of the computation.

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 download it from this repository, and run `cargo build -r` from the terminal emulator. The compiled executable should be located in `lcsfe/target/release`.

3 Usage

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

```
lcsfe-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 composed symbolic $\exists$-moves.

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

3.1 The `debug` command

The debug command has the following structure:

```
lcsfe-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 a natural number. The string represents the name of a function, which is going to be used in the system of fixpoint equations. The natural number 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 give a precise grammar specification in section Input grammar specification.

`<BASIS>` A path to a file containing all the elements of the basis. Each new 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 we want to check is above, with respect to some ordering, the basis element.

3.1.1 Input grammar specification

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

$$
\begin{array}{rcl}
\langle \mathit{eq\_list} \rangle & ::= & \langle \mathit{eq} \rangle \ \langle \mathit{eq\_list} \rangle \ \texttt{;} \mid \langle \mathit{eq} \rangle \ \texttt{;} \\
\langle \mathit{eq} \rangle & ::= & \langle \mathit{id} \rangle \ \texttt{=max} \ \langle \mathit{or\_exp\_eq} \rangle \mid \langle \mathit{id} \rangle \ \texttt{=min} \ \langle \mathit{or\_exp\_eq} \rangle \\
\langle \mathit{atom} \rangle & ::= & \langle \mathit{id} \rangle \mid \texttt{(} \ \langle \mathit{or\_exp\_eq} \rangle \ \texttt{)} \mid \langle \mathit{custom\_exp\_eq} \rangle \\
\langle \mathit{and\_exp\_eq} \rangle & ::= & \langle \mathit{atom} \rangle \ (\texttt{and} \ \langle \mathit{atom} \rangle)^* \\
\langle \mathit{or\_exp\_eq} \rangle & ::= & \langle \mathit{and\_exp\_eq} \rangle \ (\texttt{or} \ \langle \mathit{and\_exp\_eq} \rangle)^* \\
\langle \mathit{custom\_exp\_eq} \rangle & ::= & \langle \mathit{op} \rangle \ \texttt{(} \ \langle \mathit{or\_exp\_eq} \rangle \ (\texttt{,} \ \langle \mathit{or\_exp\_eq} \rangle)^* \ \texttt{)} \\
\langle \mathit{id} \rangle & ::= & \texttt{"} \ (\text{ a C-style identifier }) \ \texttt{"} \\
\langle \mathit{op} \rangle & ::= & \texttt{"} \ (\text{ any ASCII string }) \ \texttt{"}
\end{array}
$$

The grammar above represents a system of fixpoint equations. Notice that the syntactic category *and_exp_eq* has a higher precedence than *or_exp_eq*, this way we enforce the precedence of the operator $\wedge$ over $\vee$. 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, *op* would accept for example strings such as “diamond”, or “box”. Note that all operators are expressed in terms of a function, except for `and` and `or`, which are conveniently already provided, and are infix. A C-style identifier respects the following regex pattern `[a-zA-Z_][a-zA-Z0-9_]*`.

$$
\begin{aligned}
\langle \textit{sym\_mov\_list} \rangle &::= \langle \textit{sym\_mov\_eq} \rangle\ \langle \textit{sym\_mov\_list} \rangle\ \texttt{;} \mid \langle \textit{sym\_mov\_eq} \rangle\ \texttt{;} \\
\langle \textit{sym\_mov\_eq} \rangle &::= \texttt{phi (} \langle \textit{id} \rangle \texttt{) (} \langle \textit{num} \rangle \texttt{) = } \langle \textit{disjunction} \rangle \\
\langle \textit{conjunction} \rangle &::= \langle \textit{atom} \rangle\ (\texttt{and}\ \langle \textit{atom} \rangle)^* \\
\langle \textit{disjunction} \rangle &::= \langle \textit{conjunction} \rangle\ (\texttt{or}\ \langle \textit{conjunction} \rangle)^* \\
\langle \textit{atom} \rangle &::= \texttt{[} \langle \textit{id} \rangle \texttt{ , } \langle \textit{num} \rangle \texttt{]} \mid \texttt{true} \mid \texttt{false} \mid \texttt{(} \langle \textit{disjunction} \rangle \texttt{)} \\
\langle \textit{id} \rangle &::= \texttt{"}\ (\text{ a C-style identifier })\ \texttt{"} \\
\langle \textit{num} \rangle &::= \mathbb{N}
\end{aligned}
$$

The grammar above represents the symbolic $\exists$ moves for some operators. Note that, similarly to what we did for the grammar of systems of fixpoint equations, the conjunction operator has a greater precedence than the disjunction operator.

We now give the grammar of a basis: it is simply a list of strings, separated by the new-line character `\n`.

$$
\begin{aligned}
\langle \textit{basis} \rangle &::= \langle \textit{basis\_elem} \rangle\ \texttt{\textbackslash n}\ \langle \textit{basis} \rangle \mid \langle \textit{basis\_elem} \rangle \\
\langle \textit{basis\_elem} \rangle &::= \texttt{"}\ (\text{ any ASCII string })\ \texttt{"}
\end{aligned}
$$

Follows the grammar specification of a file containing the name of the operators and their arity.

$$\begin{aligned}
\langle \mathit{arity} \rangle &::= \langle \mathit{op\_name} \rangle\ \langle \mathit{num} \rangle\ \texttt{\textbackslash n}\ \langle \mathit{arity} \rangle \mid \langle \mathit{op\_name} \rangle\ \langle \mathit{num} \rangle \\
\langle \mathit{op\_name} \rangle &::= \texttt{"}\ (\text{ a C-style identifier })\ \texttt{"} \\
\langle \mathit{num} \rangle &::= \mathbb{N}
\end{aligned}$$

3.2 The `mu-ald` command

The `mu-ald` command calls the `lcsfe-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.

```
lcsfe-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.

$$\begin{aligned}
&A ::= a \mid \mathit{true} \mid \neg a \\
&\varphi ::= \boldsymbol{t} \mid \boldsymbol{f} \mid x \mid \varphi \vee \varphi \mid \varphi \wedge \varphi \mid \mu x.\varphi \mid \nu x.\varphi \mid \langle A \rangle \varphi \mid [A]\varphi
\end{aligned}$$

With $a \in \mathit{Act}$ and $x \in \mathit{PVar}$. We designed a grammar that avoids, as much as possible, left recursion. The following EBNF grammar describes a μ-calculus formula.

$$
\begin{array}{rcl}
\langle \mathit{atom} \rangle & ::= & \texttt{tt} \mid \texttt{ff} \mid \texttt{(}\ \langle \mathit{mu\_calc} \rangle\ \texttt{)} \mid \langle \mathit{id} \rangle \\
\langle \mathit{modal\_op} \rangle & ::= & \texttt{<}\ \langle \mathit{label} \rangle\ \texttt{>}\ \langle \mathit{atom} \rangle \mid\ \texttt{[}\ \langle \mathit{label} \rangle\ \texttt{]}\ \langle \mathit{atom} \rangle \\
\langle \mathit{conjunction} \rangle & ::= & \langle \mathit{modal\_op} \rangle\ (\texttt{\&\&}\ \langle \mathit{modal\_op} \rangle) \\
\langle \mathit{disjunction} \rangle & ::= & \langle \mathit{conjunction} \rangle\ (\texttt{||}\ \langle \mathit{conjunction} \rangle) \\
\langle \mathit{fix\_op} \rangle & ::= & \texttt{mu}\ \langle \mathit{id} \rangle\ \texttt{.}\ \langle \mathit{disjunction} \rangle \mid\ \texttt{nu}\ \langle \mathit{id} \rangle\ \texttt{.}\ \langle \mathit{disjunction} \rangle \\
\langle \mathit{mu\_calc} \rangle & ::= & \langle \mathit{fix\_op} \rangle \mid \langle \mathit{disjunction} \rangle \\
\langle \mathit{label} \rangle & ::= & \texttt{true} \mid \langle \mathit{id} \rangle \mid \texttt{!}\ \langle \mathit{id} \rangle \\
\langle \mathit{id} \rangle & ::= & \texttt{"}\ (\text{ a C-style identifier })\ \texttt{"}
\end{array}
$$

Moreover, we designed this grammar to respect some standard conventions: modal operators $[a]$ and $\langle a \rangle$ bind stronger than $\vee, \wedge$, and the fixpoint operators capture everything after the `.` character. The consequence is that a formula $\mu x(([a]x) \vee \nu y(\langle a \rangle y \wedge \boldsymbol{f}))$ can be written as $\mu x.[a]x \vee \nu y.\langle a \rangle y \wedge \boldsymbol{f}$, minimizing the use of parenthesis. Whenever we wish to add to a modal operator anything other than the syntactic categories $\boldsymbol{t}$, $\boldsymbol{f}$ or $x \in PVar$, parenthesis must be used, this is due to the inherent limitations of the type of parser we used. This is expressed by the rule $\langle \mathit{atom} \rangle$.

3.3 The pg command

The `pg` command uses the `lcsfe-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:

```
lcsfe-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 Examples and performance considerations

In this section we show three examples of executions of `LCSFE`. We solve the same three examples with Oink and mCRL2. We use them to discuss the performance of our tool. All tests are performed on the same machine: a laptop powered by an AMD Ryzen 5 5500 processor and 8 gigabytes of RAM, with the Linux kernel 6.5.11.

We use the following parity game, the same as in in Figure **??**.

```
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";
```

We want to know whether player ∃ wins from node `Antarctica`, to do that we run the following command.

```
> cargo run -r -- pg tests/parity_games/test_03.gm Antarctica
```

File `test_03.gm` contains the game specification.. The command `cargo run -r` compiles and run `LCSFE` in release mode, which creates an optimised executable. The compilation might take a few seconds. Then, aside from the compilation messages, the output is the following.

```
Preprocessing took: 0.000022963 sec.
Solving the verification task took: 0.000010881 sec.
Result: Player 1 wins from vertex Antarctica
```

Before running the local algorithm, a preprocessing phase takes place. The preprocessing phase encompasses extracting the fixpoint system from the specification, generating, and composing the symbolic ∃-moves. In the case of parity games the preprocessing phase consists in extracting the system of boolean fixpoint equations and composing the moves, there are only two operators in a system of boolean fixpoint equations, $\wedge$, $\vee$ and we provide symbolic ∃-moves for both by default. In the case of `lcsfe-mu-ald` the preprocessing phase is comprised of extracting the system of fixpoint equations from the μ-calculus formula, generating the symbolic moves for each operator appearing in the formula, instantiated to the labelled transition system provided as input, and composing them to symbolic ∃-moves.

We solve the same parity game, on the same machine, with Oink, via the following command.

```
> oink tests/parity_games/test_03.gm -p
```

The output of the command is shown below.

```
[    0.00] parity game with 5 nodes and 11 edges.
[    0.00] parity game reindexed
[    0.00] parity game renumbered (4 priorities)
[    0.00] no self-loops removed.
[    0.00] 2 trivial cycles removed.
[    0.00] preprocessing took 0.000017 sec.
[    0.00] solved by preprocessor.
[    0.00] total solving time: 0.000033 sec.
[    0.00] current memory usage: 4.62 MB
[    0.00] peak memory usage: 4.75 MB
[    0.00] won by even: America Asia Australia
[    0.00] won by odd: Africa Antarctica
```

In this very simple example `LCSFE` performance are on par with Oink's, even thought Oink finds the global solution, instead of just a winner from a node.

The next two examples come from the mCRL2 tutorial, which can be found at the the following link: https://www.mcrl2.org/web/user_manual/tutorial. We want to solve two problems. The first problem we want to solve is "The Rope Bridge": we want to know wheter four adventurers can cross a bridge in 17 minutes. We have the following constraints: no more than two persons can cross the bridge at once, a flashlight needs to be carried by one of them every crossing. They have only one flashlight. The four adventurers are not all equally skilled: crossing the bridge takes them 1, 2, 5 and 10 minutes, respectively. A pair of adventurers cross the bridge in an amount of time equal to that of the slowest of the two adventurers. We define the following μ-calculus formula: $\mu X.\boldsymbol{t} \vee \langle true\rangle(\langle report(17)\rangle x)$.

In order to use a mCRL2 specification in `LCSFE`, we convert it to a labelled transition system in Aldebaran format, using the tool `ltsconvert`, from mCRL2's toolset. Then we run our tool. The resulting transition system has 102 states, and 177 transitions.

```
> cargo run -r -- mu-ald tests/example_mucalc/bridge-referee.aut \
  tests/example_mucalc/receive-17 0
```

We pass as input to the command line interface the Aldebaran specification file, `bridge-referee.aut`, and the file containing the μ-calculus formula. We want to perform local model checking from state 0 of the labelled transition system. Follows the output of the command.

```
Preprocessing took: 0.02513744 sec.
Solving the verification task took: 0.000013575 sec.
```

```
Result: The property is satisfied from state 0
```

To do the same with mCRL2, we run the following commands. We first need to translate the mCRL2 specification the to `.lps`, which is a file format used internally by mCRL2.

```
> mcrl22lps bridge-referee.mcrl2 bridge.lps
```

The command below takes a μ-calculus formula, in `formula_A-final.mcf`, and the file we just generated `bridge.lps`, and builds a kind of system of boolean fixpoint equations, called parameterised boolean equation system. This process took 0.017395 seconds to finish.

```
> lps2pbes --formula=formula_A-final.mcf bridge.lps bridge.pbes --timings
```

Finally, we solve the model checking task, via the following command.

```
> pbes2bool -rjittyc bridge.pbes --timings
```

The output is the following.

```
true
- tool: pbes2bool
  timing:
    instantiation: 0.009156
    solving: 0.000032
    total: 0.038423
```

Notice how `LCSFE` performs roughly the same as mCRL2.

We now describe the next specification: "Gossips". There are five agents, each have an information that must be shared to all of them. Agents share information by making phone calls, and each time they share all of their secrets. We want to check whether this system is deadlock free. To do so, we use the following μ-calculus formula: $\varphi = \nu X.\Diamond tt \wedge \Box X$. The result of the conversion of the mCRL2 specification to Aldebaran format is a labelled transition system with 9152 states and 183041 transitions.

```
> cargo run -r -- mu-ald tests/example_mucalc/gossips.aut \
  tests/example_mucalc/deadlock-liveness 0
```

We pass as input to the command line interface the Aldebaran specification file, `gossips.aut`, and the file containing the μ-calculus formula. We want to perform local model checking from state 0 of the labelled transition system.

```
Preprocessing took: 0.16171992 sec.
```

```
Solving the verification task took: 5.695183 sec.
Result: The property is satisfied from state 0
```

We run the same commands as before, until we reach the following output.

```
true
    - tool: pbes2bool
      timing:
        instantiation: 1.397916
        solving: 0.002444
        total: 1.424587
```

We see that in this case mCRL2 is much faster. Moreover, during the execution of `LCSFE` we experienced a stack overflow: due to the recursive nature of the local algorithm we employ, the recursive calls filled a stack of 8 megabytes of size. In order to solve this verification stack, we incremented the size of the stack.