CoCoSim
Automated analysis and compilation framework for Simulink/Stateflow

NASA CoCo-Team
contact (cocosim@lists.nasa.gov)

Feb 2020

Disclaimers

THE ACCURACY AND QUALITY OF THE RESULTS OF RUNNING COCOSIM DIRECTLY CORRESPONDS TO THE QUALITY AND ACCURACY OF THE MODEL AND THE REQUIREMENTS GIVEN AS INPUTS TO COCOSIM. IF THE MODELS AND REQUIREMENTS ARE INCORRECTLY CAPTURED OR INCORRECTLY INPUT INTO COCOSIM, THE RESULTS CANNOT BE RELIED UPON TO GENERATE OR ERROR CHECK SOFTWARE BEING DEVELOPED. SIMPLY STATED, THE RESULTS OF COCOSIM ARE ONLY AS GOOD AS THE INPUTS GIVEN TO COCOSIM.

No Warranty: THE SUBJECT SOFTWARE IS PROVIDED "AS IS" WITHOUT ANY WARRANTY OF ANY KIND, EITHER EXPRESSED, IMPLIED, OR STATUTORY, INCLUDING, BUT NOT LIMITED TO, ANY WARRANTY THAT THE SUBJECT SOFTWARE WILL CONFORM TO SPECIFICATIONS, ANY IMPLIED WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE, OR FREEDOM FROM INFRINGEMENT, ANY WARRANTY THAT THE SUBJECT SOFTWARE WILL BE ERROR FREE, OR ANY WARRANTY THAT DOCUMENTATION, IF PROVIDED, WILL CONFORM TO THE SUBJECT SOFTWARE. THIS AGREEMENT DOES NOT, IN ANY MANNER, CONSTITUTE AN ENDORSEMENT BY GOVERNMENT AGENCY OR ANY PRIOR RECIPIENT OF ANY RESULTS, RESULTING DESIGNS, HARDWARE, SOFTWARE PRODUCTS OR ANY OTHER APPLICATIONS RESULTING FROM USE OF THE SUBJECT SOFTWARE. FURTHER, GOVERNMENT AGENCY DISCLAIMS ALL WARRANTIES AND LIABILITIES REGARDING THIRD-PARTY SOFTWARE, IF PRESENT IN THE ORIGINAL SOFTWARE, AND DISTRIBUTES IT "AS IS."

Waiver and Indemnity: RECIPIENT AGREES TO WAIVE ANY AND ALL CLAIMS AGAINST THE UNITED STATES GOVERNMENT, ITS CONTRACTORS AND SUBCONTRACTORS, AS WELL AS ANY PRIOR RECIPIENT. IF RECIPIENT'S USE OF THE SUBJECT SOFTWARE RESULTS IN ANY LIABILITIES, DEMANDS, DAMAGES, EXPENSES OR LOSSES ARISING FROM SUCH USE, INCLUDING ANY DAMAGES FROM PRODUCTS BASED ON, OR RESULTING FROM, RECIPIENT'S USE OF THE SUBJECT SOFTWARE, RECIPIENT SHALL INDEMNIFY AND HOLD HARMLESS THE UNITED STATES GOVERNMENT, ITS CONTRACTORS AND SUBCONTRACTORS, AS WELL AS ANY PRIOR RECIPIENT, TO THE EXTENT PERMITTED BY LAW. RECIPIENT'S SOLE REMEDY FOR ANY SUCH MATTER SHALL BE THE IMMEDIATE, UNILATERAL TERMINATION OF THIS AGREEMENT.

Contents

Abstract. This document presents the CoCoSim framework, an open-source platform to gather various tools performing verification and validation of Simulink and Stateflow models. The document first advocates for the formalization of model requirements as model components through the use of synchronous observers. Once these requirements are available different methods could be applied through the CoCoSim tool: simulation of requirements, generation of test cases, or formal verification.

Chapter 1

CoCoSim

Notice: The accuracy and quality of the results of running CoCoSim directly corresponds to the quality and accuracy of the model and the requirements given as inputs to CoCoSim. If the models and requirements are incorrectly captured or incorrectly input into CoCoSim, the results cannot be relied upon to generate or error check software being developed. Simply stated, the results of CoCoSim are only as good as the inputs given to CoCoSim.

CoCoSim is an automated analysis and code generation framework for Simulink and Stateflow models. Specifically, CoCoSim can be used to verify automatically user-supplied safety requirements. Moreover, CoCoSim can be used to generate C and/or Rust code and support test-cases generation. CoCoSim uses Lustre as its intermediate language. It is structured as a hub easing the integration of formal methods in reactive systems development. CoCoSim is currently under development. We welcome any feedback and bug report.

1.1 Installation

1.1.1 Dependencies

- MATLAB(c) and Simulink version **R2017** or newer (CoCoSim has been well tested in R2017b and R2019b versions)
- Lustrec[3] tool is a modular compiler of Lustre code into C and Horn Clauses. CoCoSim uses lustrec in many features: Test case generation, requirements importation...
- Kind2[1] is a multi-engine, parallel, SMT-based automatic model checker for safety properties of Lustre programs.

CoCoSim uses the following external libraries:

- CoCoSim standard libraries from https://github.com/coco-team/cocoSim2
- Simulink/Matlab selected toolboxes from https://github.com/hbourbouh/cocosim-external-libs

1.1.1.1 installation

Using latest stable version
We recommend this option. You can download the latest stable version of CoCoSim source code from https://github.com/NASA-SW-VnV/CoCoSim/releases.

CoCoSim is a Matlab Toolbox, in the zip we include the external libraries and binaries for Lustrec and Kind2.

You still need to run the install script to make sure binaries are working on your machine. If not remove them and run the script again.

In your terminal, go first to `scripts` folder in `cocosim` and run the `install_cocosim` script.

```
> cd CoCoSim/scripts
> ./install_cocosim
```

The script may require some dependencies, please install them and run the script again.

Use developer version

1. Clone the cocosim repository in your local machine.

   ```
   git clone https://github.com/NASA-SW-VnV/CoCoSim.git
   ```

2. Download External Matlab Libraries:

   - Open your Matlab and navigate to `CoCoSim/scripts`. Run `install_cocosim_lib(true)` from the Matlab Command window.
     Function `install_cocosim_lib.m` is responsible of copying all required external libraries to the right destination in our repository. It needs `git` to clones external repositories from github and copy some of their code on the right place on CoCoSim. Read the function `install_cocosim_lib.m` to know what are the external libraries are copied to cocosim2 to do it manually in case the function failed for internet connection or `git` issues.
   - Nnavigate back to `CoCoSim` then run `start_cocosim` from the Matlab Command window.

3. In your terminal, go first to `scripts` folder in `cocosim` and run the `install_cocosim` script.

   ```
   > cd CoCoSim/scripts
   > ./install_cocosim
   ```

Basic dependencies. `install_cocosim` script assumes that the operating system provides: `bash`, `basename`, `dirname`, `mkdir`, `touch`, `sed`, `date`, `cat`, `rm`, `mv`, `cp`, `ln`, `find`, `tee`, `patch`, `tar`, `gzip`, `gunzip`, `xz`, `make`, `install`, `git` as well as the following tools `autoconf`, `automake`, `aclocal`, `pkg-config`.

The script detects the missing dependencies that should be installed by the user. If the above script failed to install the tools. You may install them in the following paths.
If you have linux machin, change `osx` by `linux`.

KIND2 binary: `CoCoSim/tools/verfiers/osx/bin/kind2`
Z3 binary: `CoCoSim/tools/verfiers/osx/bin/z3`
LUSTREC binary: `CoCoSim/tools/verfiers/osx/bin/lustrec`
LUSTRET binary: `CoCoSim/tools/verfiers/osx/bin/lustret`
LUCTREC_INCLUDE_DIR: `CoCoSim/tools/verfiers/osx/include/lustrec`

If you want to customize these paths go to `cocosim/tools/tools_config.m` and change the values of variables KIND2, Z3, LUSTREC, LUSTRET, LUCTREC_INCLUDE_DIR to your preferences.

1.2 Run CoCoSim

1.2.1 Requirements for input models

In order to use CoCoSim, the model should compile within Simulink, ie. it should be usable for a simulation. Using Simulink Simulator engine is the best way to ensure that the model can be compiled. If the Simulink model refers to constants, these constants should be loaded into the Matlab workspace or Model workspace.

1.2.2 Overview of CoCoSim in practice

To start using CoCoSim follow these steps:

1. Launch Matlab
2. Make sure folder 'CoCoSim/' is added to your path
3. Call in Matlab command window "'start_cocosim"'
4. Open your Simulink model. Make sure you can simulate the model and all constants are defined in Matlab or Model workspace.
5. In your Simulink model go to Tools/CoCoSim

The following is some of the things you can do with through CoCoSim menu:

Check Compatibility

Check compatibility produces the list of unsupported blocks or options of some blocks. Please note, the checking compatibility does not guarantee that the model is fully supported but it detects the blocks/options we already know we do not support. You may have other error messages that a specific block is not supported later on the compilation step. For example, Stateflow States and Transitions actions will not be checked for compatibility until the translation step. Errors will be raised explaining what is not supported.

In addition the compatibility is performed only for Verification, a model can be supported for proving properties but not for C code generation. Since model checkers (such as Kind2) support abstractions, a block can be abstracted for verification but we can not produce C code for that specific block.

Prove Properties

Prove properties will start the Verification back-end. Depending on your Preferences you set in *tools -> CoCoSim -> Preferences*, a Lustre code is generated and the model checker Kind2 is called. The results of the model checker is reported back to the user as an HTML report in addition to the possibility to simulate the Counterexample if the property is falsified.

The model should have first the requirements attached to it using CoCoSim Specification Library or Simulink Design Verification Library. See section 2 for how to specify requirements at the Simulink model.

CoCoSim Preferences

In *tools -> CoCoSim -> Preferences* you can choose different settings:

- **Simulink To Lustre compiler**: Currently two compilers are available, NASA compiler and IOWA compiler. We recommend using NASA compiler, it supports more Simulink blocks and it was well tested on a large set of regression tests.
- **NASA compiler preferences**: If you choose NASA compiler in the previous preference. The compiler comes with different parameters, we explain some of them here:
  - *Abstract Integer machine types (int8, int16..) by Z ([-oo, +oo])*: If checked, Simulink integer machine types signals will be considered all as mathematical integers. Use this option if you want to prove high level properties without taking into account Integer overflow. By default, it's unchecked.
  - *Use more precise abstraction for mathematical functions (sqrt, cos, ...).*: Use this option to use Lookup tables to abstract unsupported mathematical functions. By default, a simple abstraction is used, for example *cos* function is abstracted by $[-1, 1]$ interval. Having more precise abstract may make the verification harder and take longer time to finish.

  - *Skip compatibility check*: If checked, the compatibility check step will be ignored. This is useful if the model is big and the compatibility is already checked previously by the user.
  - *Abstract unsupported blocks for verification (Kind2)*: This option helps abstract blocks that are not supported in CoCoSim by a black box. The abstraction is useful to enable verification for unsupported models but it may produce nonsensical counterexamples. They are only useful if the properties were proven to be valid. The user can give more precise abstractions for the unsupported blocks by attaching contracts to them.
  - *Skip pre-processing (not recommended):* Only useful if you are running CoCoSim in an already pre-processed model and the pre-processing takes too much time.
  - *Skip defected pre-processing blocks (recommended):* when the option is checked, CoCoSim stops after every pre-processing library call and simulates the model to make sure that particular pre-processing call did not brake the model. In case the model is not executable anymore, CoCoSim restores the saved version of the model just before the library call.
  - *Skip Lustre code optimization:* Enable this option if the optimization takes too much time or if you want a more readable Lustre code (Lustre equation for each Simulink block).
  - *Force Lustre code Generation (i.e., ignore errors):* If enables, the check compatibility will be ignored and a Lustre code will be generated even if the model is not supported. This may cause a generation of an invalid Lustre code, but it could be useful for having a partial Lustre code that can be used by the user.
  - *Skip Stateflow parser check (not recommended):* The Statflow to Lustre compiler checks for each State/Transition action that we parsed the Matlab expression correctly. It should be checked, if the model contains too much Stateflow actions and the user does not want to go over the actions every time CoCoSim compiles the model.

- **Verification Backend:** Currently only Kind2 is supported.
- **Kind2 Preferences:**
  - *Compositional Analysis:* If checked, contracts will be used as abstraction of the Subsystems attached to.
  - *Kind2 Binary:* Only local binary is supported for NASA compiler.
- **Lustrec binary:** Only *Local* option is supported for the moment. Support for docker is in progress.
- **Verification timeout:** This is the model checker timeout.
- **Verbose level:** 0 to display less messages in the Matlab Command window, 3 to display everything.
- **Reset preferences:** to reset preferences to their default values.

Chapter 2

Formalize requirements as Simulink components

A major step when considering formal verification of models or software is the need to specify requirements in a formal fashion. Formalization of requirements means expressing specification, which is usually defined in large documents using natural languages, as computer-processable elements. Depending on the context, on the kind of specifications or on objects of studies, these elements can be logical predicates, physical measures associated to resources, or even other programs.

A strong benefit of synchronous languages such as Simulink, Scade or Lustre is the possibility to rely on regular model constructs to specify these requirements as model components. These are called synchronous observers.

In this chapter, we provide some insights regarding their definition and their use to support verification and validation activities. We present two means of formalizing requirements, the first is using Simulink Verification Library that comes with SLDV toolbox. The second is using CoCoSim own Specification library using the notion of assume-guarantee contracts. The last section gives the difference in the semantic of the two libraries, we restrict the user to not combine both libraries blocks and only use one of them.

2.1 Simulink Verification Library

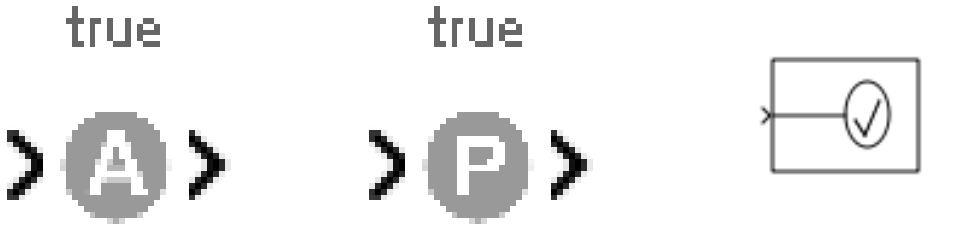
Simulink Design Verifier[1] provides blocks for requirements modeling[2]. CoCoSim supports the blocks in Fig. 2.1 to specify requirements. We explain here the use for each block:

- *Proof Assumption* block is used to specify assumptions about your model. For example, constraining signal values when proving model properties.
- *Proof Objective* block is used to specify properties to prove. It defines objectives that signals must satisfy when proving model properties.

[1] `https://www.mathworks.com/products/simulink-design-verifier.html`
[2] `https://www.mathworks.com/help/sldv/modeling-requirements-for-verification.html`

- *Assertion* block checks whether signal is zero. This block is a special case of *Proof Objective* block where the input signal should be non-zero.

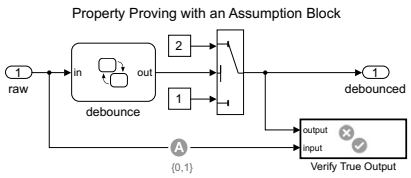
The example in Fig. 2.2 demonstrates the use of the *Proof Assumption* block and the *Assertion* block. The property is specified inside the Verification Subsystem which makes the model separated from requirements, which is a good practice we recommend.



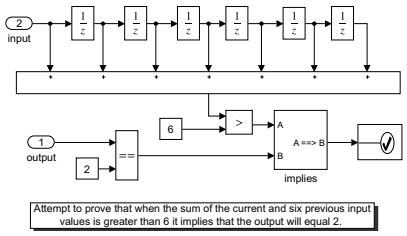
(c) Assertion block

(a) Proof Assumption block (b) Proof Objective block

Figure 2.1: Simulink Design Verification blocks



(a) Verification subsystem is linked to an Assumption block assuming signal values of input raw is 0 or 1.



(b) The content of the verification subsystem "Verify True Output"

Figure 2.2: Property proving with Assumption block.

2.2 CoCoSim Specification Library

To specify requirements, the user can use the specification library provided by IOWA university to express assume/guarantee contracts. A detailed explanation of the library can be found in : `https://github.com/coco-team/cocoSim2/blob/master/doc/specificationLibrary.md`. Also we recommend watching the following video explaining how to create contracts from scratch `https://coco-team.github.io/cocosim/videos/2_contracts_simulink.mp4`.

Figure 2.4 gives the main blocks used inside a contract to specify requirements. Keep in mind, a contract is attached to a specific Subsystem, both inputs and outputs of the Subsystem are linked to the contract (see Fig. 2.3).

- **Assume** block is used to add properties about the inputs of your Subsystem. It's a boolean function of the inputs of the Subsystem. For example constraining an input to an interval of values, or assuming an input is positive. A contract can have more than one assumption.
- **Guarantee** block is used to add properties about the outputs of the Subsystem. It's a boolean function of the inputs/outputs of the Subsystem. A contract can have more than one guarantee.
- **Mode** block is always attached to two other blocks:
  - **Require** block is the local assumption for the specific mode. It's the condition that activated that mode. For example, a Stopwatch is in reset mode, if the reset button is on. Therefore, the require condition of resetting mode is reset input to be true.
  - **Ensure** block is the local guarantee of that specific mode. It's the properties that should always hold when the Mode is active (when Require is positive). For example, in reset Mode the Stopwatch output should be zero.

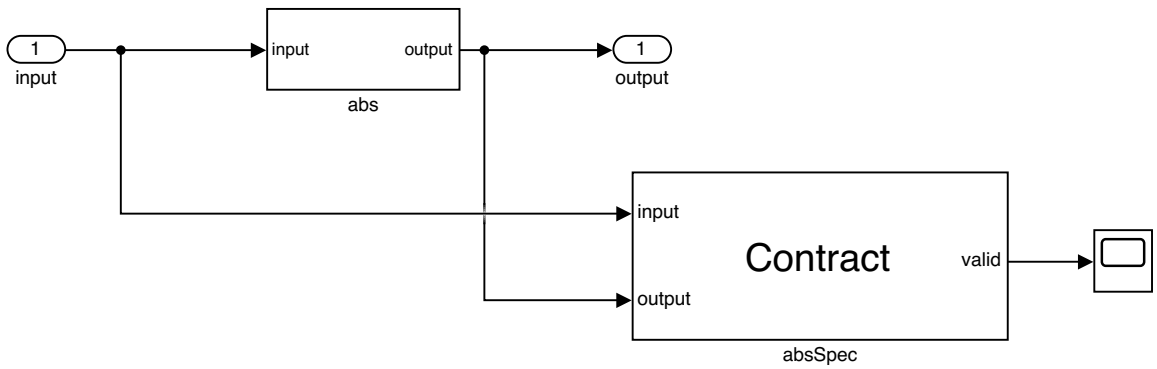


Figure 2.3: CoCoSim Contract attached to a Subsystem.

Note: the contract is not aware of what the Subsystem is connected to unless you specify this in the assumption. For example, In Fig. 2.3, if the Inport block "*input*" is a Constant block instead of an Inport, and the user wants to prove the properties in the contract for that specific constant, The user needs to explicitly add the assumption inside the contract that assumes the input of Subsystem "*abs*" is constant.

2.3 Difference between CoCoSim contracts and SLDV verification blocks

Both CoCoSim contracts and SLDV specification blocks provide means for specifying the environment or pre-conditions of the model (Assumptions about inputs for CoCoSim contracts and Proof Assumption block over internal signals for Simulink

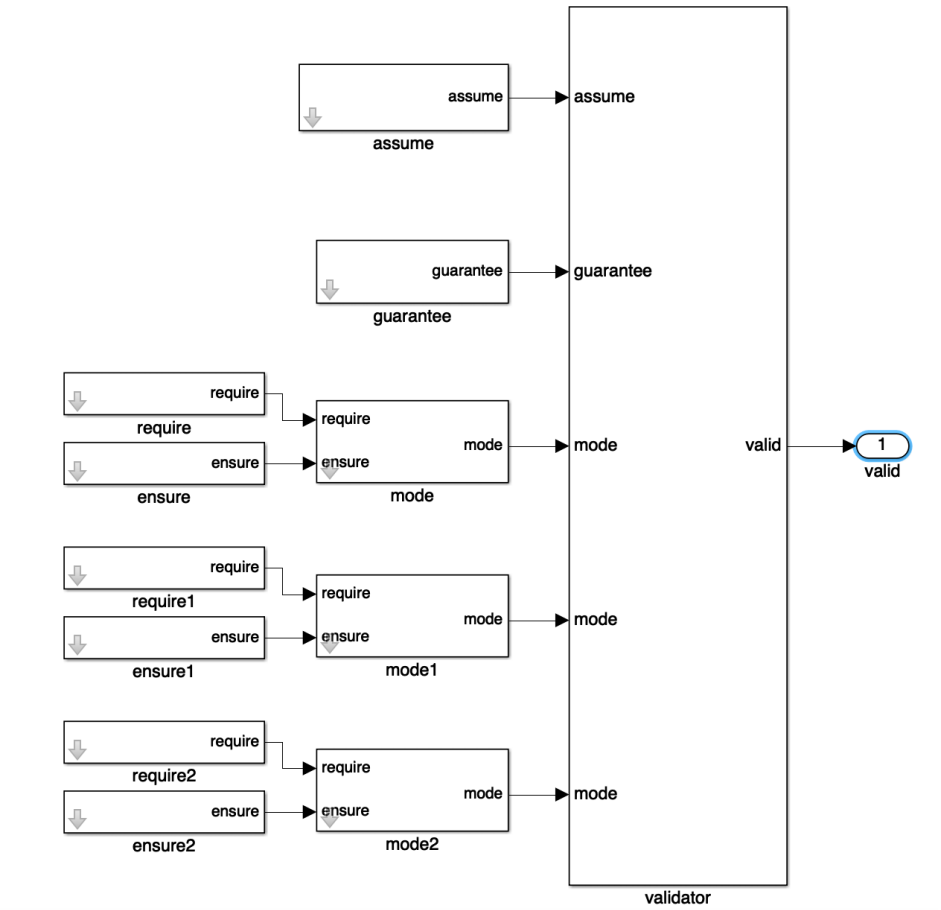


Figure 2.4: The content of CoCoSim contract

specification library). In addition, both libraries provides means to express the properties that need to be verified, they consist of the post-conditions and the guarantees of the System.

The main difference is that a CoCoSim contract is attached to a specific Subsystem, it provides the pre-conditions that should be respected for that specific Subsystem and express the properties that are guaranteed to hold if the pre-conditions always hold. It allows CoCoSim and Kind2 to reason locally instead of the whole model, it also allows to prove properties compositionally.

Whereas SLDV specification library is used over signals and reason about the whole Simulink model.

Chapter 3

Overview of the platform

In this chapter we give an overview of the platform and explain the computations performed in the three main phases of the tool and provides a description of each phase features.

3.1 CoCoSim struture

CoCoSim is strutured as a compiler, sequencing a serie of translation steps leading, eventually to either the production of source code, or to the call to a verification tool. By design, each phase is highly parametrizable through an API and could then be used for different purposes depending on the customization.

The Figure 3.1 outlines the different steps.

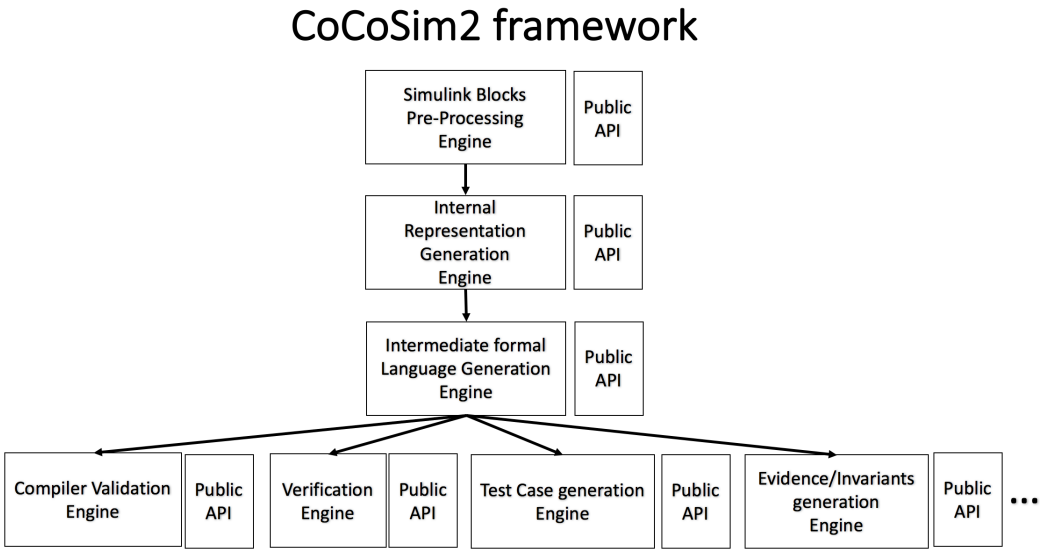


Figure 3.1: CoCoSim framework

Front-End. CoCoSim is a toolbox that can be called directly from the Matlab Simulink environment (similar to Simulink Design Verifier). CoCoSim can be used

either for code generation (e.g. C/Rust and/or Lustre), for test-case generation or for property verification (More back-ends are in progress). The front-end performs a bunch of pre-processing (i.e. lowering of different Simulink blocks into basic blocks, see section 3.2.1) and optimizations. The second step of the front-end is generating an intermediate representation of the Simulink model. This intermediate representation can be exported as a Json file, see section 3.2.2.

Middle-End: Compiler from Simulink to formal language. This step translates modularly the pre-processed Simulink model and generates a formal language. The main target is currently Lustre models while other outputs from the internal representation could be produced. Moreover, it performs book-keeping of the different Simulink/Stateflow constructs (e.g. signal name, block type etc ..). This allows to trace Simulink/Stateflow constructs in the resulting Lustre programs.

Back-End: CoCoSim features. CoCoSim back-ends provide most of CoCoSim features. From specifying properties graphically to verifying these properties or generating test-cases. Some features require the execution of the complete chain, from pre-processing to Lustre generation, ie. verification or test-case generation tools. Some others remain at Simulink level, ie. tools supporting the definition of requirements. The current features of CoCoSim can be found in Section 3.4.

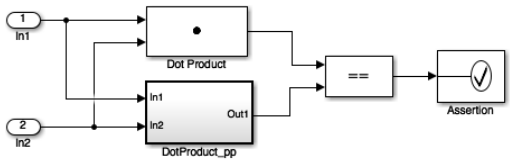
3.2 Front-End

The front-end phase is splitted into two parts. A first one performs model-to-model transformation, generating a preprocessed Simulink model. Typically it substitute some blocks by an equivalent version relying on simpler definitions. The second part produces an intermediate representation of the pre-processed model.

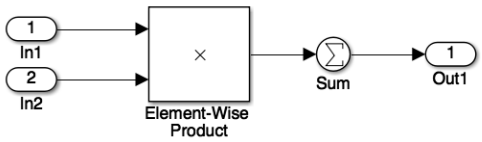
3.2.1 Preprocessing blocks

Pre-processing engine is an independent library in CoCoSim that takes a Simulink model as an input and produces a simpler, yet equivalent version of that model. Simulink provides many pre-defined block libraries (e.g., *Integrator*, *Saturation*, *Dot Product*, *Transfer Function*) that help the user build the model faster. These predefined block libraries can be expressed using basic blocks such as Product and Sum. For instance, Fig. 3.2 illustrates the simplification of the *Dot Product* block by a combination of an element-wise product of two vectors and the sum of the elements of the resulted vector.

The pre-processing engine consists of a set of libraries performing a simplification of an atomic block. Each library is a MATLAB function defined in a separate file that takes as an input the name of the model; it performs transformations on it using Simulink API functions such as *find_ system*, *add_ block*, *delete_ block*, *get_ param* and *set_ param*. CoCoSim provides a configuration file where the user can change the order of libraries execution. The order is important when a transformation of a block is resulting in a new block that needs to be handled by another library. New libraries



(a) Dot Product block and its substituted version that their equivalence can be checked using SLDV



(b) The content of DotProduct_pp Subsystem in Fig. 3.2a

Figure 3.2: Example of simplifying Dot Product Simulink block.

can also be defined by the user by adding the MATLAB function file and setting the order of execution in the configuration file.

Moreover, CoCoSim provides means of verifying the equivalence between the original model and the pre-processed model. For example, in Fig. 3.2a, we proved the equivalence between *Dot Product* and its simplified version using Simulink Design Verifier (SLDV).

The main objective of this pre-processing step is to reduce the number of blocks that need to be translated to the intermediate formal language (currently Lustre). Therefore, we mainly pre-process atomic blocks into basic blocks that will be supported later by the translator.

3.2.1.1 Pre-Processing configuration

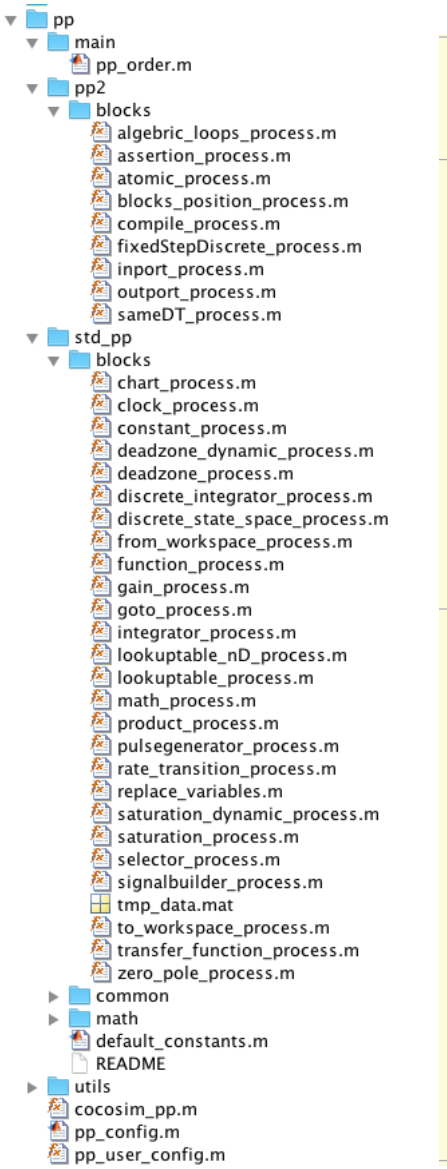
CoCoSim pre-proccessing performs a sequence of model-to-model transformation, following a given order. The structure of CoCoSim architecture is highly generic and the pre-processing can be a combination of multiple preprocessing librariries, with a specific order.

The default libraries are coming from two sources: one defined by CMU and IOWA university and one developed within NASA. Both libraries are complimentary and perform different pre-processing.

3.2.1.1.1 Configure pre-processing order programmatically The function call order is specified in `main/pp_order.m`. The pre-processing tree is presented in Fig. 3.3a.

The `std_pp` and `pp2` folders are two libraries that offer some pre-processing functions. `std_pp` refers to the standard library from CMU while `pp2` is from NASA. The file `main/pp_order.m` (Fig. 3.3b) defines which functions have to be executed and their order. `pp_handled_blocks` and `pp_unhandled_blocks` are variables defining accepted and rejected blocks. Functions are defined thanks to their relative path to the pre-processing folder. The user can give an absolute path to other functions not exist in CoCoSim source code.

The map `pp_order_map` defines a priority for each set of functions. Priority -1 is associated to ignored functions. Priority 0 is the highest priority and functions are run by the ascending order of priority. Regular expressions can be used. For example, one can give priority 3 to all functions in folder 'pp2/blocks' using:



(a) Pre-Processing tree

```
%% TODO: add imported libraries paths
% In our case "main" library import both "std_pp" and "pp2" libraries
addpath(genpath(fullfile(config_path, 'std_pp')));
addpath(genpath(fullfile(config_path, 'pp2')));


%% TODO: add blocks to be pre-processed or to be ignored
% Here are the functions to be called (or to be ignored) in the
% pre-processing.
% examples:
% -To add all supported blocks in `std_pp`, add 'std_pp/blocks/*.m'
% -To add all supported blocks in `pp2` except `atomic_process.m`.
%   Add 'pp2/blocks/*.m' to pp_handled_blocks and
%   Add 'pp2/blocks/atomic_process.m' to pp_unhandled_blocks
% -To impose a specific order of functions calls see above.

% add both std_pp and pp2
pp_handled_blocks = {'std_pp/blocks/*.m',...
    'pp2/blocks/*.m'};
% To not call atomic_process we may add it to the following list, or
% give it an order -1 in pp_order_map (see next TODO).
pp_unhandled_blocks = {'pp2/blocks/atomic_process.m',...
    'pp2/blocks/compile_process.m',...
    'pp2/blocks/blocks_position_process.m',...
    'std_pp/blocks/product_process.m'};
%compile process is called in the end of cocosim_pp.


%% TODO: define orders
% pp_order_map is a Map with keys define the priority and Values define
% functions list
pp_order_map = containers.Map('KeyType', 'int32', 'ValueType', 'any');

% -1 means not to call
pp_order_map(-1) = {'pp2/blocks/atomic_process.m'};
% 0 means all this functions will be called first.
pp_order_map(0) = {'pp2/blocks/inport_process.m', ...
    'pp2/blocks/outport_process.m'};

pp_order_map(1) = {'std_pp/blocks/goto_process.m'};
pp_order_map(2) = {'pp2/blocks/blocks_position_process.m'};
% '*.m' means all std_pp functions have the same priority 1,
% if a function already defined it will keep its highest priority.
pp_order_map(3) = {'std_pp/blocks/*.m', ...
    'pp2/blocks/*.m'};


pp_order_map(4) = {'pp2/blocks/algebric_loops_process.m', ...
    'pp2/blocks/fixedStepDiscrete_process.m'};

pp_order_map(5) = {'pp2/blocks/compile_process.m'};

[ordered_pp_functions, priority_pp_map]  = ...
    PP_Utils.order_pp_functions(pp_order_map, pp_handled_blocks, ...
    pp_unhandled_blocks);
```

(b) Pre-Processing configuration file

Figure 3.3: Pre-Processing parametrization.

```
pp_order_map(3) = {'pp2/blocks/*.m'};
```

3.2.1.1.2 GUI-order configuration A configuration GUI (Fig. 3.4) helps the user to define the order of functions and adding new functions. It can be called using the function `pp_user_config` in Matlab command line.

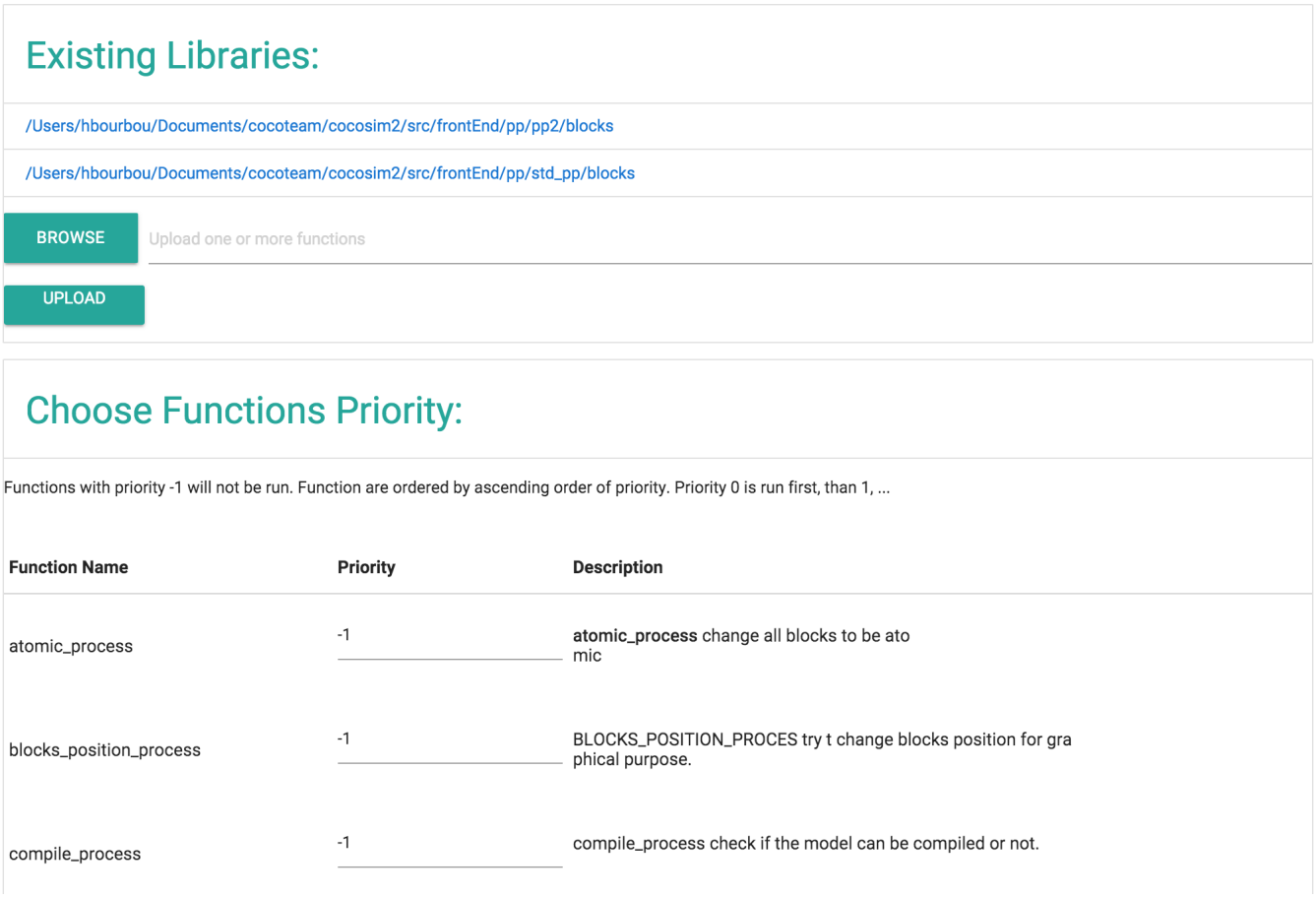


Figure 3.4: Pre-Processing user configuration interface

3.2.1.2 Extending Pre-Processing Libraries

The user can define more pre-processing libraries. The simplest way is to add the new functions in one of the folders `pp/pp2` or `pp/str_pp`. Any function added to the previous folders will be executed unless given priority -1. The user can also define his personal folder. In that case, the user should follow the configuration steps in section 3.2.1.1.

3.2.2 Internal Representation of Simulink/Stateflow

The internal representation (IR) allows to export the simplified model into a hierarchical tree-based structure. We present here the structure and the syntax of its fields. The structure is specific to the input model. The internal representation is an independent

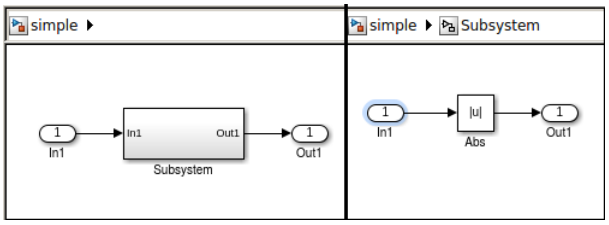


Figure 3.5: Absolute value subsystem.

library, it provides a hierarchical representation of the whole model even for the blocks that may not be supported later on the translation.

3.2.2.1 Simulink IR

The function `cocosim_IR` produces the IR associated to a given Simulink model. The function's signature is as follow:

```
function [ir_struct, all_blocks, subsyst_blocks, handle_struct_map] =
              cocosim_IR( simulink_model_path, df_export, output_dir )
```

This function takes one mandatory parameter – the model – and two optional ones. The option `df_export` is disabled by default. When set to `true` the call produces a json file containing the resulting IR structure. The third parameter `output_dir` specifies the path where the json file is saved. By default the model path is used.

The function returns the IR (a struct in matlab), the list of all blocks present in the model, the list of all subsystems or blocks treated as subsystems in the model, and a map of block's handles (IDs) associated with the struct of the block in the IR. This last one simplifies IR accesses for the `get_struct` function.

Here is the description of the structure of the IR:

```
IR = {"meta" : META, "model_name" : {SUBS_IR}}
META = {"date" : date, "file_path" : model_path}
SUBS_IR = "Content" : {BLOCKS_IR}
BLOCKS_IR =  "block_formated_name" : {PROPERTIES, SUBS_IR}, BLOCKS_IR
PROPERTIES = PropertyName : value, PROPERTIES
```

Since field of struct cannot have spaces or line breaks, the block's names are first formated to have correct field names. The original one is contained in the `Origin_path` property in the IR.

The first component of the IR is the meta data. It contains informations such as date of creation of the IR or model path. The model representation is rather straightforward, each block is defined as its set of properties and values. The Figure 3.5 presents a simple Simulink model computing the absolute value of an input flow and Figure 3.6 its IR.

3.2.2.2 Stateflow IR

A stateflow chart is represented as a Program with the following attributes:

```
SFCHART = {
      "name" : chart_name,
      "origin_path" : Simulink_path_to_chart,
```

```
{
    "meta": {
        "date": "07-Sep-2017",
        "file_path": "./simple"
    },
    "simple": {
        "Content": {
            "In1": {
                "BlockType": "Inport",
                "Handle": 1964.000244140625,
                "Name": "In1",
                "Origin_path": "simple/In1",
                "Path": "simple/In1",
                "Port": "1"
            },
            "Out1": {
                "BlockType": "Outport",
                "Handle": 1966.000244140625,
                "Name": "Out1",
                "Origin_path": "simple/Out1",
                "Path": "simple/Out1",
                "Port": "1"
            },
            "Subsystem": {
                "BlockType": "SubSystem",
                "Content": {
                    "Abs": {
                        "BlockType": "Abs",
                        "Handle": 1972.000244140625,
                        "Name": "Abs",
                        "Origin_path": "simple/Subsystem/Abs",
                        "Path": "simple/Subsystem/Abs"
                    },
                    "In1": {
                        "BlockType": "Inport",
                        "Handle": 1959.00048828125,
                        "Name": "In1",
                        "Origin_path": "simple/Subsystem/In1",
                        "Path": "simple/Subsystem/In1",
                        "Port": "1"
                    },
                    "Out1": {
                        "BlockType": "Outport",
                        "Handle": 1960.000244140625,
                        "Name": "Out1",
                        "Origin_path": "simple/Subsystem/Out1",
                        "Path": "simple/Subsystem/Out1",
                        "Port": "1"
                    }
                },
                "Handle": 1958.000244140625,
                "IsSubsystemVirtual": "on",
                "Mask": "off",
                "MaskType": "",
                "Name": "Subsystem",
                "Origin_path": "simple/Subsystem",
                "Path": "simple/Subsystem",
                "SFBlockType": "NONE",
                "ShowPortLabels": "FromPortIcon"
            }
        }
    }
}
```


Figure 3.6: Absolute value subsystem intermediate representation.

```
        "states": [STATE*],
        "junctions":[JUNCTION*],
        "sffunctions":[SFCHART*],
        "data":[DATA*]
}

STATE = {
        "path" : path,
        "state_actions" : {
                    "entry_act":entry_act,
                    "during_act":during_act,
                    "exit_act":exit_act,
        },
        "outer_trans": [TRANSITION*],
        "inner_trans": [TRANSITION*],
        "internal_composition": COMPOSITION
}

JUNCTION = {
        "path" : path,
        "type" : 'CONNECTIVE' | 'HISTORY',
        "outer_trans": [TRANSITION*]
}

TRANSITION = {
        "id": id,
        "event": event,
        "condition" : condition,
        "condition_act": condition_act,
        "transition_act": transition_act,
        "dest": {
                    "type": 'State' | 'Junction',
                    "name": dest_name
        }
}

COMPOSITION = {
        "type": 'EXCLUSIVE_OR'| 'PARALLEL_AND',
        "tinit": [TRANSITION*],
        "substates": [substates_names list]
}

DATA = {
        "name": name,
        "scope": 'Local'| 'Constant'| 'Parameter'| 'Input'| 'Output' ,
        "datatype": DataType, /*Type of data as determined by Simulink*/
        "port": port_number, /*Port index number for this data */
        "initial_value": initial_value,
        "array_size": array_size, /*Size of data as determined by
           Simulink.*/
}
```

- origin_path: the Simulink path to the Stateflow chart.
- name: The name of the Stateflow chart.

- states: List of chart's states, a state is represented by:
  - path: the full path to the state.
  - state_actions: State actions (entry, exit and during actions).
  - outer_trans: List of outer transition of the state. Each transition is represented by:
    - id: a unique ID of the transition.
    - event: Sting containing the name of the event.
    - condition: String containing the condition that triggers the transition.
    - condition_act: String containing the condition actions.
    - transition_act: String containing the transition actions.
    - dest: The destination of the transition. Which is a structure containing:
      - type: 'State' or 'Junction'.
      - name: The path to destination.
  - inner_trans: List of inner transition of the state.
  - internal_composition: Is the composition of the state. It has the following attributes:
    - type: even 'EXCLUSIVE_OR' or 'Parallel_AND'
    - tinit: List of default transitions of the state.
    - substates: List of sub-states names of the current state.
- junctions: List of junctions. Junction is defined by its path, type and the outer transition from this junction.
- sffunctions: List of Stateflow functions. A Stateflow function is a special chart, it contains the same attributes that defines a chart.
- data: List of all chart data variables. A data variable is defined by a name, scope (local, input, parameter or output), a datatype (int8, int16 ...) and its initial value when defined.

3.2.2.3 Matlab IR

Matlab code can also be exported in a Json format. The Matlab grammar in ANTLR4 format and java source code handling this grammar can be found in `cocosim2/src/IR/matlab_IR/EM` The user can define a new transformation from Matlab AST to another language or format. Following the existing example of transforming Matlab to Json format may help. Fig. 3.7 shows an example of a simple Matlab function and its equivalent in Json format.

Note that the subset of Matlab expressions accepted of extremelly limited and cannot involve sophisticated Matlab functions.

```
function z = f(x, y)
z = x + y;
end
```


```
{
    "functions": [
        {
            "type": "function",
            "name": "f",
            "return_params": [
                "z"
            ],
            "input_params": [
                "x",
                "y"
            ],
            "statements": [
                {
                    "type": "assignment",
                    "operator": "=",
                    "leftExp": {
                        "type": "ID",
                        "name": "z"
                    },
                    "rightExp": {
                        "type": "plus_minus",
                        "operator": "+",
                        "leftExp": {
                            "type": "ID",
                            "name": "x"
                        },
                        "rightExp": {
                            "type": "ID",
                            "name": "y"
                        }
                    }
                }
            ]
        }
    ]
}
```


Figure 3.7: Internal representation of a simple Matlab code in Json format.

3.3 Middle-End: generation of Lustre models

The middle end phase translate the produced internal representation into a target formal language. We chose the Lustre synchronous language since it shares similarities with the discrete subsytem of Simulink. In addition, it is an academic language which syntax and semantics have been widely studied. Furthermore multiple tools addressing its analysis are available.

3.3.1 Lustre compiler

CoCoSim translate discrete-time Simulink to Lustre code. The main goal is to preserve Simulink semantics. The compilation from Simulink to Lustre is hierarchical block-by-block compilation. Every Subsystem (treated as atomic) will be translated in a Lustre node.

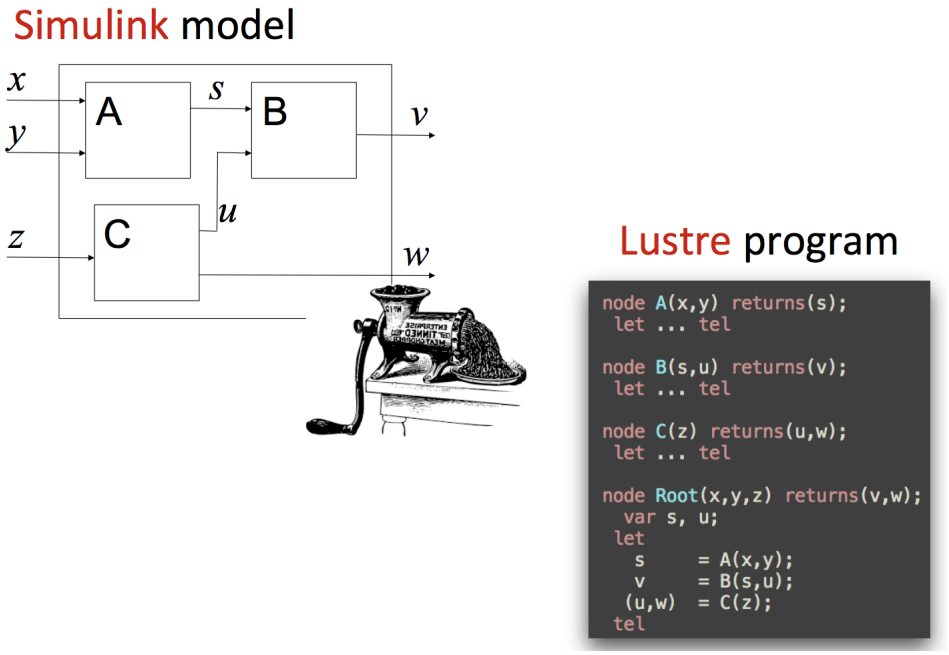
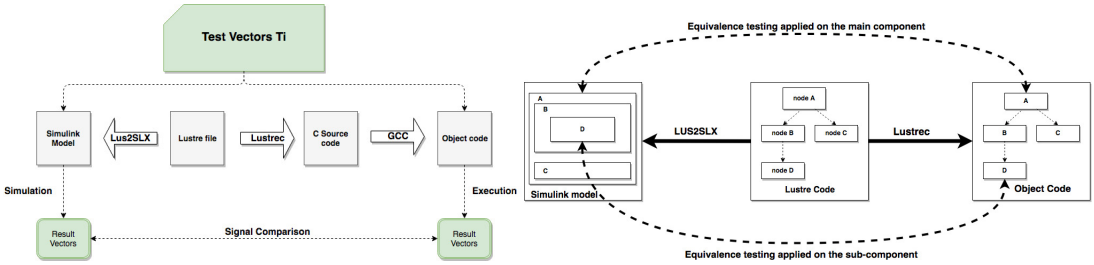


Figure 3.8: Bottom-up compilation.

The **Atomic** feature of a subsystem can be specified at Simulink level. While a regular subsystem acts as a virtual bound around a set of of simulink block, an **atomic** block shall produce a dedicated function, preserving the subsytem hierarchy in the final code. The use of atomic blocks has advantages and drawbacks. **Preserving the structure greatly supports verification and validation**. It is more easy to perform code review since the compilation is more tractable. Furthermore the verification activities are eased. Unit tests can be performed or modular analyses. On the negative side the memory footprint is larger and there are less opportunities to improve or optimize the code.



(a) (b)

Figure 3.9: (a) The equivalence testing approach. (b) Equivalence testing applied on the main component as well as the sub-components

CoCoSim provides links to multiple compilers. The main one is *LustreC* [3, 2], a compiler developped at Onera and IRIT but a new one supporting advanced contracts is being developed by IOWA university. Another compiler supporting multi-periodic subsystems is also under development.

3.3.2 Validation of CoCoSim compiler

Validating the translation from Lustre to Simulink is a mandatory task to increase confidence in the compilation process. We proposed the combination of multiple verification and validation methods for that purpose.

There are two type of validation, Translator validation and translation validation. The first class is based on validating the compiler for all possible executions. The second is interested on one individual translation. The choice of CoCoSim is to focus on the second approach in which we validate the translation after each execution. This validation is integrated in the process of development and performed automatically [4].

In the following sections we present the different methods we used to validate Lustre compiler. In the following we rely on a feature of LustreC allowing to generate a Simulink model from a Lustre input. This feature, combined with the capability of CoCoSim to generate Lustre model from a Simulink is used to obtain two models in Simulink or in Lustre of the same input and compare them. Note that CoCoSim and LustreC are completely independent and do not share code.

3.3.2.1 Equivalence testing

Equivalence testing is based on numeric equivalence between the target and the source. Figure 3.9 illustrates the equivalence testing approach. In this case the target is a Simulink model that can be simulated and executed. Our source language is a Lustre code that can be compiled to C code using LustreC and then can be executed as a C binary. This approach is fully automated and the signal comparison between the to result vectors (see fig 3.9a) is based on an epsilon given by the user (by default 10^{-15} (ie. `1e-15`).

This approach can be applied also for validating Lustre sub-nodes and not only the main node. Figure 3.9b illustrates the use of equivalence testing on subcomponents, relying on the Lustre-to-Simulink translation capbilities. This is possible thanks to hierarchical preserving during the compilation, every node in Lustre is translated back

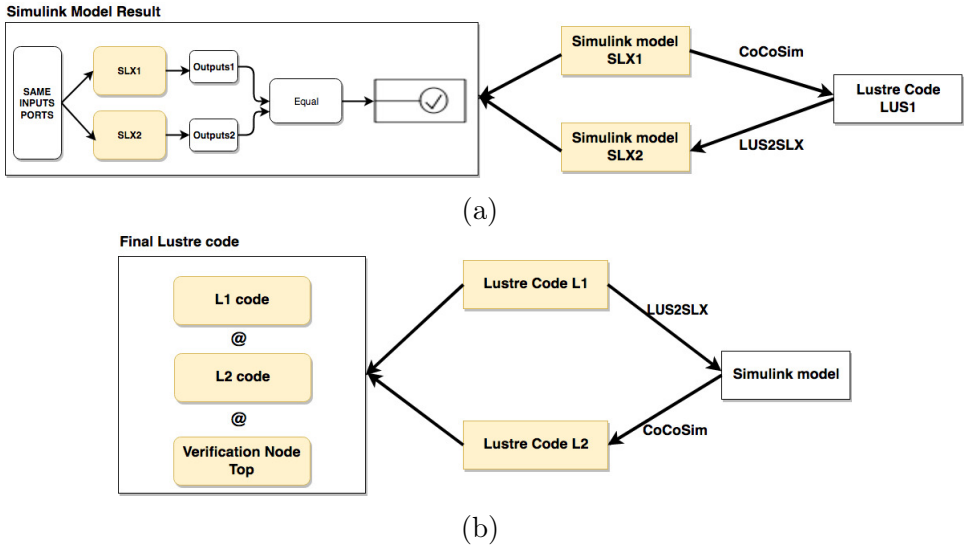


Figure 3.10: (a) The equivalence checking applied on Simulink level. (b) Equivalence checking applied on Lustre level

as a subsystem in Simulink. In case of an invalid translation of the main model, validating the subcomponents supports the identification of the subsystem or node causing the validation error.

Test vectors are then used to stress the equivalence of the two available representations. These tests can either be user-provided or generated through different means (as described in the Back-End chapter): random tests, coverage-based synthesis, mutation-based synthesis.

3.3.2.2 Equivalent checking

Equivalence checking is a much stronger result since it exhaustively guarantees that both models are equivalent. It proves that the source model and the target are equivalent for all possible inputs. We used different encodings of the problem: a monolothic global and a compositional local one.

Depending on the availability of tools, the verification can be performed at

- either at Simulink level between the initial Simulink and the re-interpretation of the generated Lustre model as a new Simulink model;
- or at Lustre level, between the first generated Lustre model, and its interpretation of Simulink model, compiled again in Lustre.

Monolithic Verification. Monolithic verification is based on verifying a property P on the entire system and in all its components. In order to increase our confidence in both CoCoSim and LustreC tools, we followed the approach described in Figure 3.10.

In a first pattern, presented in Figure 3.10a, a Simulink model $SLX1$ is translated to its equivalent Lustre code using CoCoSim. This latter is translated back to a new Simulink model $SLX2$. Both tools use different approaches in the translation.

Therefore both $SLX1$ and $SLX2$ have the same hierarchical design but use different Simulink blocks. $SLX2$ should be semantically equivalent to $SLX1$.

We combine both $SLX1$ and $SLX2$ in a new Simulink model in order to prove that for the same inputs both $SLX1$ and $SLX2$ return the same outputs. We use Simulink Design Verifier to prove the equivalence.

In a second pattern, presented in Figure 3.10b, Lustre code $L1$ is translated to its equivalent Simulink model $SLX1$ using LustreC tool. Then $SLX1$ is translated to Lustre code $L2$ using CoCoSim. Now we could combine both $L1$ and $L2$ in the same verification file and create a top verification node as follows:

```
node top(inputs) returns (OK:bool)
var outputs1, outputs2;
let
  outputs1 = L1(inputs);
  outputs2 = L2(inputs);
  OK = outputs1 = output2;
  --!Property OK=true;
tel
```

We can use any model-checker (such as Kind2 and Zustre) based on Lustre to verify the equivalence.

Compositional Verification. In Compositional verification we check one component at a time. Each sub-componant is abstracted by a stub since only the local equations are evaluated. Thanks to the traceability of both tools LustreC and CoCoSim, we can map every node in $L1$ to $L2$ (not from $L2$ to $L1$ since we have additional nodes in $L2$ introduced by CoCoSim): each node in $L1$ has been translated to a subsystem in the Simulink model $SLX1$ and then back to another Lustre node in $L2$.

Each node in $L1$ is associated to a contract specifying that the node should be equivalent to its associated version in $L2$. Let's assume there is a node called $L1\_A$ in $L1$ and its equivalent in $L2$ is called $L2\_A$. We generate the following contract, using CoCoSpec specification language supported by Kind2.

```
node L1_A(inputs) returns (outputs);
(*@contract guarantee outputs = L2_A (inputs); *)
let
   --BODY
tel
```

The modular and compositional analysis of such a file with Kind2 allows to validate most functions while remaining unproven ones have to be checked with other methods such as equivalence tests. In case of failure the model-checker returns a candidate trace violated the property.

3.3.3 Supported Blocks

In addition to blocks supported by pre-processing and the ones considered as masked subsystems with supported content, CoCoSim supports more than 100 blocks. Table 3.1 lists the unsupported blocks in the standard Simulink libraries. The table does not include other toolboxes such as the Aerodefence Simulink toolbox that may have masked blocks that are supported (most Quaternion manipulation blocks are masked blocks that are supported by the tool). Also some blocks are supported partially in CoCoSim based on the block configuration or block features (See Table. 3.2).

Library	# supp. Blocks	% supp. Blocks	Unsupported blocks
Discontinuities	11/12	91%	Backlash
Discrete	19/21	90%	Discrete PID Controller, Discrete PID Controller (2DOF)
Logic & Bit Operations.	18/19	95%	Extract Bits
Lookup Tables.	9/9	100%	
Math Operations.	31/37	83%	Algebraic Constraint, Complex to Magnitude-Angle, Complex to Real-Imag, Find, Magnitude-Angle to Complex, Real-Imag to Complex
Model Verification	11/11	100%	
Ports & Subsystems.	29/31	93%	While Iterator Subsystem, While Iterator
Signal Attributes.	13/14	93%	Unit Conversion
Signal Routing.	13/25	52%	Data Store Memory, Data Store Read, Data Store Write, Environment Controller, Goto Tag Visibility, Index Vector, State Reader, State Writer, Variant Source, Variant Sink, Manual Variant Source, Manual Variant Sink
Sinks.	9/9	100%	Visualization blocks are ignored
Sources.	15/26	57%	Band-Limited White Noise, Counter Free-Running, Counter Limited, From File, From Spreadsheet, Repeating Sequence, Repeating Sequence Interpolated, Repeating Sequence Stair, Signal Editor, Signal Generator, Waveform Generator
User-Defined Functions.	2/15	13%	Argument Inport, Argument Outport, Event Listener, Function Caller, Initialize Function, Interpreted MATLAB Function, Level-2 MATLAB S-Function, MATLAB System, Reset Function, S-Function, S-Function Builder, Simulink Function, Terminate Function

Table 3.1: Simulink blocks that are not supported. The list is not exaustive, as other Simulink blocks are partially supported depending on block parameters (See Table. 3.2).

Block Name Name	Unsupported Options
Trigonometry	Operator 'cos + jsin' is not supported.
Switch	Allow different data input sizes option is not supported.
Relational Operator	Operator "isInf", "isNaN", "isFinite" are not supported.
Merge	- Allow unequal port widths is not supported. - We support only Merge blocks that are connected to conditionally-executed subsystem.
Function Call Generator	Number of iterations > 1 is not supported.
For Iterator	- External iteration limit source is not supported. - External increment is not supported. - Action Ports inside a For Iterator block should have "States when execution is resumed" option set to "reset". - Outports in a conditionally executed Subsystem inside a For Iterator block should have "Output when disabled" set to "reset". - Memory blocks are only allowed in the first level of the For Iterator Subsystem.
Discrete Pulse Generator	Option "Use external signal" is not supported.
Demux	Bus selection mode should be off.
Selector	"Starting and ending indices (port)" option is not supported.
Multiport Switch	- "Specify indices" option is not supported. - Allow different data input sizes is not supported.
Lookup Table blocks	- More than 7 dimensions interpolation is not supported. - "Intermediate results Data Type" option should be set to double or single.
From Workspace	"Cyclic repetition" option is not supported.
Delay	When Delay length > 1, the initial condition should be scalar.
Concatenate	Concatenate dimension > 2 is not supported.
Assignment	- OutputInitialize set to 'Specify size for each dimension in table' is not supported, - IndexOptionArray set to 'Starting and ending indices (port)' is not supported.

Table 3.2: Simulink blocks that are partially supported.

3.4 Back-End

Back-ends cover the main capabilities of CoCoSim framework. Most back-end rely on the intermediate representation expressed as a Lustre model to perform analyzes or compilation of that Lustre models and provide back information at Simulink level.

For the moment CoCoSim is offering the following features: verification engines (cf. Sect. 3.4.1), code generation (cf. Sect. 3.4.2), invariant generation (cf. Sect. 3.4.3), test-case generation and coverage evaluation (cf. Sect. 3.4.4), Support of specification, including Lustre to Simulink translation (cf. Sect. 3.4.5), compilation validation (cf. Sect. 3.4.6), and code analysis (cf. Sect. 3.4.7).

As mentioned in the introduction, the goal of the CoCoSim framework is to ease the application of formal methods for Simulink-based systems. The backends are introduced and linked to the platform in a very generic way. While CoCoSim is built mainly around a specified set of tools, additional ones can be easily locally linked or even distributed as extensions.

3.4.1 Verification

Takes as input the formal language generated at the Middle-end step (e.g. Lustre code) and performs safety analysis of the provided property using one of model-checkers: Zustre or Kind2. More model-checkers can be integrated in the tool. The result of the safety analysis is reported back to the Simulink environment. In case the property supplied is falsified, CoCoSim provides means to simulate the counterexample trace in the Matlab environment. Otherwise, in case of success, proof evidence expressed as generated invariant can also be propagated back at model-level as synchronous observers.

3.4.2 Code Generation

CoCoSim generates C, Rust or Lustre code from Simulink model. This compilation process is validated by equivalence testing and equivalence checking. See section 3.3.2.

3.4.3 Invariants Generation

- Using Zustre: Zustre is a PDR-based tool, while analyzing Simulink model requirements, Zustre can produce a counter-example in case of failure, but also returns a set of invariants in case of success. These invariants can be expressed as runnable evidence in the Simulink level. They are expressed as a set of local invariants that could be attached to Simulink subsystems.

- Using IKOS: We try in this work in progress to generate invariants in the Simulink level using Abstract Interpretation using IKOS tool. They will be expressed as a set of local invariants that could be attached to each Simulink subsystems.

3.4.4 Test-Case Generation

The CoCoSim framework can rely on various methods to synthesize test cases. We elaborate here on some approaches and provide elements explaining the performed computation. All of these techniques rely on a model-checker such as Kind2 or Zustre to perform bounded model-checking. A property, or, more precisely its negation, is given to the tool and the transition relation is unrolled until the property becomes invalid. The resulting so-called counter-example is a test case that activates the property, eventually. This unrolling process is called BMC: bounded model-checking.

Figure 3.11 presents the set of approaches that can be run at Lustre level and then propagated back as test case at the Simulink level.

3.4.4.1 Specification based test generation

A first natural use of BMC is to encode the contracts as targets. Interesting elements could be active states for modes, or transitions from one mode to another. One could also generate multiple test cases for each specification by adding more criteria, for example range conditions for values.

3.4.4.2 Mutation based test generation

In the following we denote by *mutant* a mutated model or mutated implementation where a single mutation has been introduced. The considered mutation does not change the control flow graph or the structure of the semantics but could either: (i) perform arithmetic, relational or boolean operator replacement; or (ii) introduce additional delay (pre operator in Lustre) – note that this could produce a program with initialization issues - or (iii) negate boolean variable or expressions; or (iv) replace constants.

Such generation of mutants has been implemented as an extension to our Lustre to C compiler. The latter can now generate a set of mutant Lustre models from the original Lustre model. Once mutants are generated and the coverage-based test suite is computed, we can evaluate the number of mutants killed by the test suite. This evaluation is performed at the binary level, once the C code has been obtained from the compilation of the mutant. In this setting, the source Lustre file acts as an oracle, i.e. a reference implementation. Any test, that shows a difference between a run of the original model compiled and a mutation of it, allows to kill this mutant.

In the litterature, mutants are mainly used to evaluate the quality of a test suite, allowing to compare test suites. In our case, the motivation is different, we aim at providing the user with a test suite related to its input model. This test suite covers the model behavior in order to show that the compiler doesn't introduce bugs. A test suite achieving a good coverage of the code but unable to kill lots of mutant would not show that the compiler did a good job. Indeed any unkilled mutant would then be as good as the initial model while in practice they are different. We have therefore to introduce new tests to kill those mutants unkilled by the existing test suite.

We rely on the BMC to compute such a trace between the two versions of the Lustre model. It may happen that the solver does not terminate or return a usable output. First the BMC engine may not be capable of generating a counter-example trace – i.e., the condition used might be an invariant. Second the difference between out, the original output, and out', the mutated one, may be unobservable. The latter is possible in mutated programs where the mutation does not impact the observed output. For example, a condition $a \vee b$ was always true because of b while the mutation was performed in the computation of a. In this case the mutation is unobservable and it is here related to dead code. This kind of mutation-based test suite reinforcement is also able to detect some of those programmming issues of the input model. In practice, our algorithm tries both to find a trace showing the different between out and out' but also to prove their equivalence. The latter case would then exhibit some issues solely related to the input model, with no relevance to our problem.

3.4.4.3 Coverage-based test generation

Usually the quality of a test suite is measured with its capability to fulfill a given coverage criteria. Depending on the criticality of the considered system the coverage criteria is more or less difficult to meet. Among the various coverage criteria, the Modified Condition/Decision Coverage (MC/DC) is recognized, with respect to testing the usefulness and influence of each model artifact, as the strongest and therefore the most costly to achieve.

CoCoSim can generate or complement a test suite to meet such a coverage. The approach is the following: each condition is expressed as a dedicated predicate; then we rely on BMC to generate a test case that activates this condition. Let us develop how one can express the MC/DC criterion as a predicate over node variables. First, we need an external procedure which can extract the decision predicates from the source code. This analysis generates a list of such conditions, eg. all boolean flow definitions.

Coverage of each decision predicate is checked in isolation, against a given global set of test cases. The principle is the following: from a decision $P(c_1, \ldots, c_n)$ where the c_i's are a set of atomic conditions over the variables $\tilde{s}$, $\tilde{in}$ and $\tilde{out}$, we have to exert the value of each condition c_i with respect to the global truth value of P, the other conditions $c_{j\neq i}$ being left untouched. Precisely, we have to find two test cases [1] for which, in the last element of the trace, c_i is respectively assigned to $False$ and $True$. Then, for each such test case, blindly changing the value of c_i should also change the global predicate value. Formally, for a given decision $P(c_1, \ldots, c_n)$, the set of predicates describing the last element of its covering traces is:

$$\left\{ \begin{array}{l} c_i \wedge (P(c_1, \ldots, c_n) \oplus P(c_1, \ldots, c_{i-1}, \neg c_i, c_{i+1}, \ldots, c_n)) \ , \\ \neg c_i \wedge (P(c_1, \ldots, c_n) \oplus P(c_1, \ldots, c_{i-1}, \neg c_i, c_{i+1}, \ldots, c_n)) \end{array} \right\}_{i \in 1..n} \tag{3.1}$$

Note that the process may not succeed for each condition since the property can be (1) unreachable or (2) undecidable to the SMT solver behind the BMC analyzer.

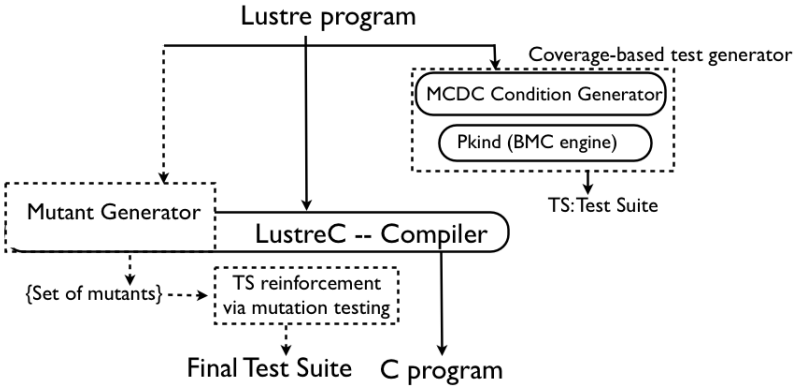


Figure 3.11: Combinaison of approaches to support test generation.

3.4.5 Lustre to Simulink

This back-end is interested in translating any Lustre code to pure Simulink. It has many uses. We use this compilation process for our compiler validation. See section

[1] In practice, a single test case may cover both cases, at different steps of the trace.

3.3.2. It can also be used to annotate Simulink models with specification written as Lustre models.

3.4.6 Compiler Validation

This back-end is interested in validating the compiler from Simulink to Lustre. See section 3.3.2.

3.4.7 Design error detection using IKOS

Detect Design Errors using abstract interpretation (IKOS). We are interested in detecting the following Design errors:

- Integer overflow.
- Division by zero.
- Dead logic.
- Out of bound array access.
- Derived Ranges of signals : (the interval approximation [a, b] of a signal x).

Bibliography

[1] Adrien Champion et al. "CoCoSpec: A Mode-Aware Contract Language for Reactive Systems". In: *SEFM'16*. 2016, pp. 347–366. ISBN: 978-3-319-41591-8. DOI: 10.1007/978-3-319-41591-8_24. URL: http://dx.doi.org/10.1007/978-3-319-41591-8_24.

[2] Pierre-Loïc Garoche, Temesghen Kahsai, and Xavier Thirioux. "Hierarchical State Machines as Modular Horn Clauses". In: *HCVS'16*. 2016, pp. 15–28. DOI: 10.4204/EPTCS.219.2. URL: http://dx.doi.org/10.4204/EPTCS.219.2.

[3] Pierre-Loïc Garoche, Temesghen Kahsai, and Xavier Thirioux. *LustreC*. URL: %5Curl%7Bhttps://github.com/coco-team/lustrec%7D.

[4] Pierre-Loïc Garoche et al. "Testing-Based Compiler Validation for Synchronous Languages". In: *NASA Formal Methods - 6th International Symposium, NFM 2014, Houston, TX, USA, April 29 - May 1, 2014. Proceedings*. 2014, pp. 246–251. DOI: 10.1007/978-3-319-06200-6_19. URL: https://doi.org/10.1007/978-3-319-06200-6_19.