CoCoSim
Automated analysis and compilation framework for Simulink/Stateflow

CoCo-Team
contact (hamza.bourbouh@nasa.gov)

May 2017

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

Need to 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 first part of the document, we provide some insights regarding their definition and their use to support verification and validation activities.

1.1 Synchronous language and Synchronous observers

1.1.1 A glance at synchronous dataflow languages

Let us first outline the specificities of synchronous languages and draw some parallel between constructs.

Synchronous languages aim at specifying the behavior of synchronous reactive systems. This model of computation describes programs that intend to be run forever, repeating the same computation regularly, at fixed time steps. Usually the time step length is not an element of the program itself but more like a meta information required for further developments such as the scheduling of multiple processes. Synchronous models assume that the processing time of functions is immediate and communication is assumed instantaneous [1]. These unreasonable hypotheses allow to separate the concerns: on the one hand, a functional description of the computation – the model –, and, on the other hand, physical constraints: the evaluation of the function body has to meet the deadlines. For example, if the program is expected to be executed 100 times a second, ie. at 100Hz, then the evaluation of the function should be performed is less than 10 ms. This constraint is the subject of dedicated analyses such as the computation of worst case execution time or the computation of bounds over network delays. It can also impact the hardware development, providing requirements in terms of computational power.

That said, the model itself can focus on functional behavior and the computations performed at each time step, regardless of these time step lengths.

Among the varieties of synchronous languages, let us focus briefly on three of them. First Matlab Simulink, produced by TheMathWorks. It is a *de facto* standard in the industry and supported by strong Matlab toolboxes, providing a large set of advanced mathematical functions. One of the strong point of Simulink is the capability, in the model, to denote both continuous and discrete components. While continuous components will have, eventually, to be discretized before being executed on the final embedded platform, they are essential ingredients of the development process of controllers. Indeed, the theory of control largely rely on continuous models to first describe the plant semantics and then design a feedback controller satisfying desirable properties of stability, robustness and performances. Simulink is also fitted with strong simulation means enabling the computation of traces of models combining discrete components and continuous ones specified with ODEs (Ordinary Differential Equations). When considering the discrete subset of Simulink, it can be used to automatically produce embedded code, accelerating code development while minimizing the introduction of bugs during the design.

Another industrial language is ANSYS Scade and its associated academic companion Lustre. While Scade resemble to the discrete subset of Simulink with similar graphical objects to describe the model components, Lustre is an equivalent yet textual language. Since the CoCoSim toolchain relies on Lustre as an intermediate language, let us provides some elements about its syntax and semantics.

1.1.1.1 Lustre [2, 8] and relationship to Simulink

Lustre is a synchronous language for modeling systems of synchronous reactive components. Lustre code consists of set of nodes transforming infinite streams of input flows as streams of output flows. A notion of symbolic "abstract" universal clock is used to model system progress. In Lustre, a node is defined as a set, ie. unordered, of stream equations, with possible local variables denoting internal flows. Regular arithmetic and comparison operators are lifted to sequences and are evaluated at each time step. If-then-else constructs are functionals and should be well typed: they build values instead of sequencing imperative statements. Ie. a valid flow equation could be `x = 3 + (if y > 0 then 4 else y);`

Stateful constructs. Temporal operator `pre`, for *previous*, enables a limited form of memory, allowing to read the value of a stream at the previous instant. It is corresponds to the unit delay operator of Simulink or to its Memory block. The arrow operator, know as *follow-by*, allows to build a stream `c -> e` as the expression `e` while specifying the first value `c`. A flow defined as `c -> pre e` will then correspond in Simulink to a Unit Delay over flow `e` with initial value `c`. A node containing such operators in its expressions is considered as *stateful*, meaning that it does not act as a pure mathematical function but is fitted with an internal state describing the current values for `pre` and whether it is in its first time step or not.

The following Lustre node describes a simple program: a node that every four computation steps activates its output signal, starting at the third step. The `reset` input reinitializes this counter.

```
node counter(reset: bool) returns (active: bool);
var a, b: bool;
let
  a = false -> (not reset and not (pre b));
  b = false -> (not reset and pre a);
  active = a and b;
tel
```

In that example the streams `a` and `b` are local ones and defined by the two first equations. Expressions within equations can also make calls to other nodes. Different occurrences of call to the same stateful node represent different instances of memories.

Hierarchy and algebraic loops. Nodes and calls form a hierarchy of nodes comparable to the notion of subsystems in Simulink. Types and clocks inference can guarantee, at compile time, that expressions and function calls respect their type constraints and properly rely on previous values to build current ones. For example the following equations are accepted and produce an *algebraic loop* error: `x = f(y); y = g(x);` The same definitions with either `f(pre y)` or `g(pre x))` would be typable and accepted by the compiler. Note that Simulink manage to solve these errors dynamically, when running simulation. The resolution of algebraic loop can either be performed by inlining nodes, or through the introduction of a `pre` construct acting as a buffer. While inlining preserves the semantics, introducing a new memory does not. It is therefore important to address these possible algebraic loop early in the process to master their resolution.

Clocks and resets. Another specific construct is the definition of clocks and clocked expressions. Clocks are defined as enumerated types, the simplest ones being boolean clocks. Expressions can then be clocked with respect to such clock values: `e when c` where `c` is a boolean clock. In this case the expression is only defined when variable `c` is positive. Clocked expressions can be gathered using `merge` operator:

```
 x = merge c (true -> e1 when c) (false -> e2 when not c);
```

Expressions associated to each clock case have to be clocked appropriately. The clocking phase of the compiler allows to check the consistency of clocks definitions and their uses. Clocks can also be used to reset a node in its initial state using the syntax `f(x) every c`. When `c` holds this instance of node `f` is restored in its initial setting with all arrow equations pointing to their left value. Recursively all callee of `f` are also reset. Again, one can see some similarities with the Enable Subsystems of Simulink.

1.1.1.2 Extensions to model automata

All of these frameworks provide extensions to express automata. In the Matlab context, Stateflow can support the definition of such automata. Stateflow is a toolbox developed by TheMathWorks that extends Simulink [11] with an environment for modeling and simulating state machines as reactive systems. A Stateflow diagram can be included in a Simulink model as one of the blocks interacting with other Simulink components using input and output signals. Stateflow is a highly complex language with no formal semantics[1]: its semantics is only described through examples on TheMathWorks website [10] without any formal definition. A Stateflow diagram has a hierarchical

[1]At least not provided as a reference by the tool provider.

structure, which can be either arranged in *parallel* in which all states are eventually executed, following a specific order; or *sequentially*, in which states are connected with transitions and only one of them can be active. The occurrence of a signal or the computation of a new time step allows the active state to evaluate transitions and can perform an unbounded number of side effects over the automaton variables. In practice the use of Stateflow in actual system has to be restricted to a limited number of construct in order to guarantee, for example, the execution time of one time step computation. The typical use is to rely on these automata to build a set of boolean flows denoting the active mode of the system. This boolean flow is then used in the regular Simulink model to drive the computation.

The Scade/Lustre approaches also propose extensions with automata. In this context, automata definition acts as a basic construct and can be mixed with classical flow definitions. Therefore the content of a node could define regular flows as well as automaton, i.e. hierarchical state machines. Each automaton state is also defined with a Lustre node which can, itself, contains regular flows and automaton. The semantics is very constrained and specifies the notion of weak or strong transitions. A single step computation can fire at most one weak and one strong transition.

1.1.2 Means of expressing the axiomatics semantics: varieties of synchronous observers

In [9], "An Axiomatic Basis for Computer Programming", HOARE defines a deductive reasoning to validate code level annotations. This paper introduces the concept of HOARE triple $\{Pre\}code\{Post\}$ as a way to express the semantics of a piece of code by specifying the postconditions ($Post$) that are guaranteed after the execution of the code, assuming that a set of preconditions (Pre) was satisfied. HOARE supports a vision in which this axiomatic semantics is used as the "ultimately definitive specification of the meaning of the language [...], leaving certain aspects undefined. [...] Axioms enable the language designer to express its general *intentions* quite simply and directly, without the mass of detail which usually accompanies algorithmic descriptions." When this pair $(Pre, Post)$ is associated to a function, it can be interpreted as a function contract. In a more general use of formal specification, the local reasoning about the function makes the assumption Pre but, when this function is called, the precondition has to be guaranteed. Otherwise the function is not fully specified and its behavior is not defined.

This idea has been naturally extended to synchronous dataflow languages with the concept of synchronous observer [7, 13, 12]. A synchronous observer encodes a predicate corresponding to the postcondition of the Hoare triple. However since the semantics is not expressed over values but flows of values, the principle of Hoare triple has to be lifted to sequences of values.

$$\{Pre(state, inputs)\}node(in, out)\{Post(state, state', in, out)\}$$

means

$$\Box\left(\bigwedge \begin{array}{l}\mathcal{H}(Pre(state, input))\\ node(state, state', in, out)\end{array} \implies Post(state, state', in, out)\right).$$

with $\mathcal{H}(p) \triangleq \{$ p has held since beginning $\}$. The operator $\mathcal{H}$ can be defined in Lustre with the node `Sofar`:

```
node Sofar (in: bool) returns (out: bool);
let
  out = in -> pre out and in;
tel
```

Such a synchronous contract is active when, at a given time step, all the inputs and internal states, up to now, have satisfied the precondition. It is valid if then the postcondition always applies.

Graphically speaking a synchronous observer is a subsystem that accesses to some internal flows and computes a boolean output. Figure 1.1 performs such a computation and verifies that a specific relationship between its two inputs is always valid.

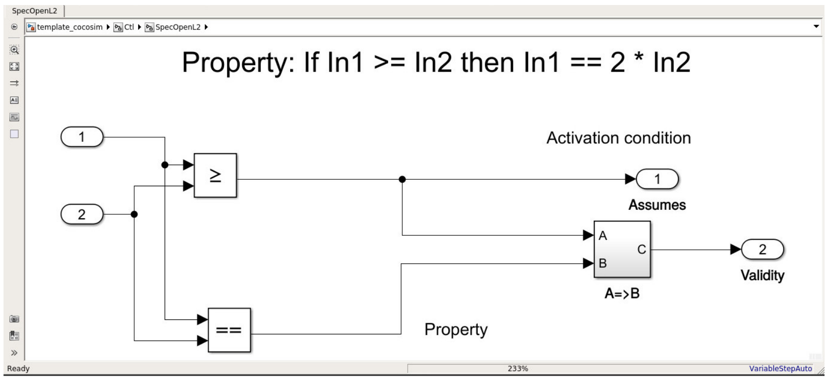


Figure 1.1: Simple synchronous observer as Simulink subsystem

In control theory we speak about an open-loop property: the property can be expressed over the controller inputs, outputs or memories without knowledge of the plant semantics. Figure 1.2 presents the association of such a synchronous observer, an open-loop property, attached to a component element.

The content of the observer itself is left free and could be as complex as required, depending on the complexity of the requirement it models. While this notion is expressive enough and is capable of capturing all kinds of requirements, it is sometimes more convenient to refine the specification by expressing hypotheses, ie. the precondition of the Hoare triples, or modes, conditional behavior depending on some conditions.

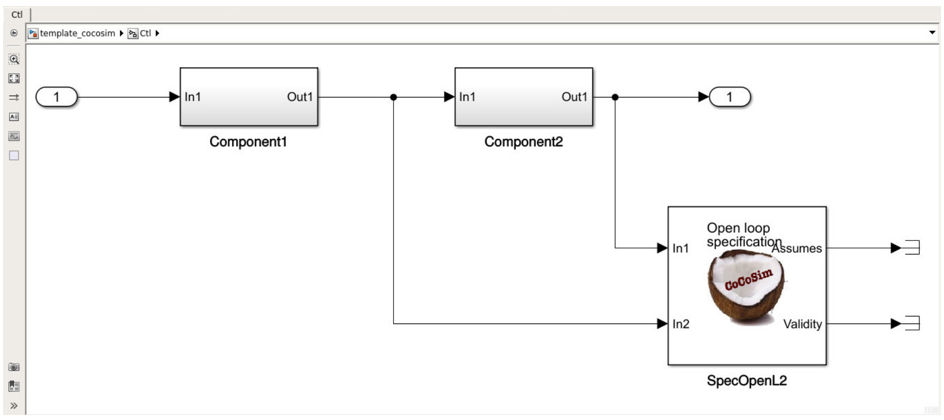
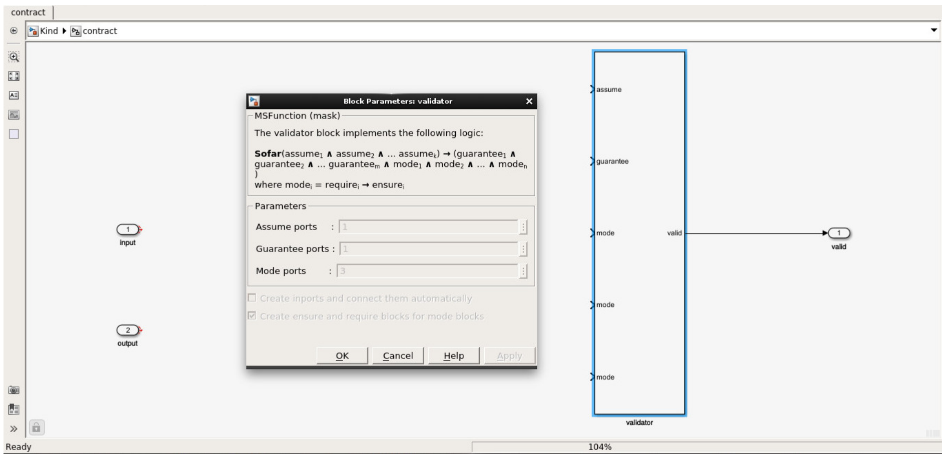


Figure 1.2: Open-loop properties in a synchronous observer

```
contract ml ( altRequest , fpaRequest , deactivate : bool ; altitude , targetAlt : real )
    returns ( altEngaged , fpaEngaged : bool ) ;
let
  var altRequested = switch(altRequest , deactivate) ;
  var fpaRequested = switch(fpaRequest , deactivate) ;
  var smallGap = abs(altitude - targetAlt) < 200.0 ;
  assume altitude >= 0.0 ;
  guarantee targetAlt >= 0.0 ;
  guarantee not altEngaged or not fpaEngaged ;
  mode guide210Alt ( require smallGap ; require altRequested; ensure altEngaged ; ) ;
  mode guide210FPA ( require smallGap ; require fpaRequested ;
                     require not altRequested; ensure fpaEngaged; ) ;
  mode guide180 ( require not smallGap ; require fpaRequested; ensure fpaEngaged; ) ;
  mode guide170 ( require not smallGap ; require altRequested ;
                  require not fpaRequested; ensure altEngaged ; ) ;
tel

node ml ( altRequest , fpaRequest , deactivate : bool ; altitude , targetAlt : real )
returns ( altEngaged , fpaEngaged : bool );
(*@contract import
     mlSpec ( altRequest , fpaRequest , deactivate : bool ; altitude , targetAlt : real )
     returns ( altEngaged , fpaEngaged : bool ); *)
let ... tel
```

Figure 1.3: Listing of CoCoSpec contracts at Lustre level using modes.



Block Parameters: validator

MSFunction (mask)

The validator block implements the following logic:

Sofar(assume$_1$ ∧ assume$_2$ ∧ ... assume$_k$) → (guarantee$_1$ ∧ guarantee$_2$ ∧ ... guarantee$_m$ ∧ mode$_1$ ∧ mode$_2$ ∧ ... ∧ mode$_n$)
where mode$_i$ = require$_i$ → ensure$_i$

Parameters

Assume ports : 1
Guarantee ports : 1
Mode ports : 3

☐ Create inports and connect them automatically
☑ Create ensure and require blocks for mode blocks

OK Cancel Help Apply

Figure 1.4: Modes as Simulink contracts

In Lustre, recent works [3] proposed a dedicated language to annotate Lustre model with a rich specification. Figure 1.3 gives an example. The node `ml` represents the mode logic of an aircraft controller, deciding whether the autopilot is active or not. Its specification is described in a `contract`. This contract can bind new variables but, more importantly, can specify the precondition `altitude >= 0.0` for that contract. Two mains postcondition are expressed as well as four different modes. Each of these modes is guarded by some conditions in the `require` expressions, while a conditional postcondition `ensure` is specified. Last, in the actual Lustre node, the contract is declared.

At Simulink level dedicated constructs, such as shown in Fig. 1.4, ease the definition of such model-based contracts.

Regarding the complexity of the synchronous observer node, it can contain any legal Simulink or Scade/Lustre content. As an example, Figure. 1.5 presents a template to

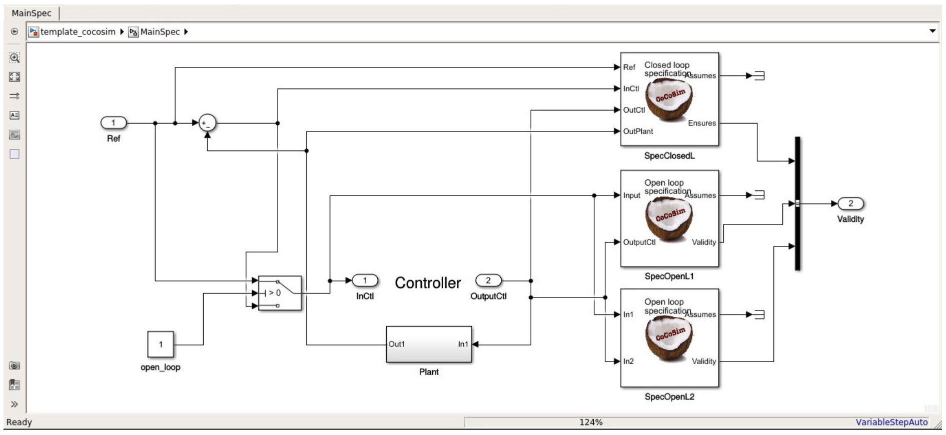


Figure 1.5: Encoding closed-loop properties in an observer

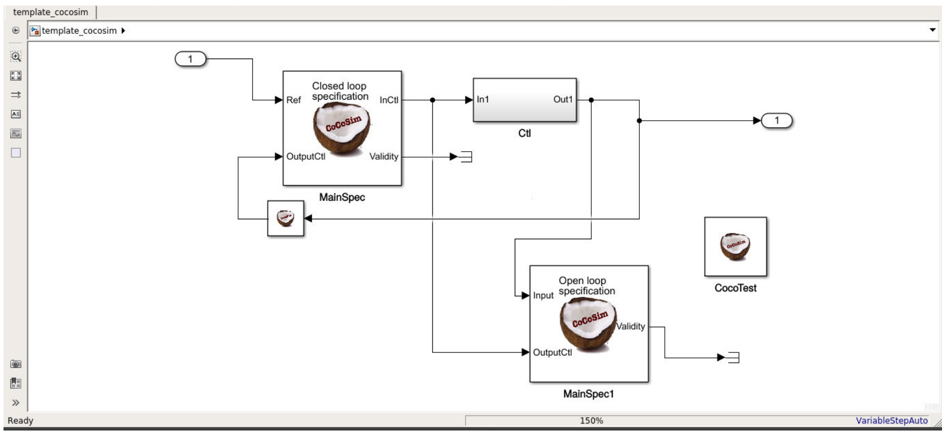


Figure 1.6: Injecting closed-loop observers as model annotations

support the expression of closed loop properties. This observer contains both the plant model and a set of closed and open-loop properties. Within that specification subsystem, observers can have access to any flows, including the plant' flows.

However, the insertion of the closed-loop specification node within a model is not as convenient that it is for an open-loop property. The open one could be defined only with probes, while the closed one needs, maybe artificially, to reconstruct a feedback loop. This is presented in Figure. 1.6. Note the occurrence of a *specification-based unit delay* to prevent the creation of a spurious algebraic loop.

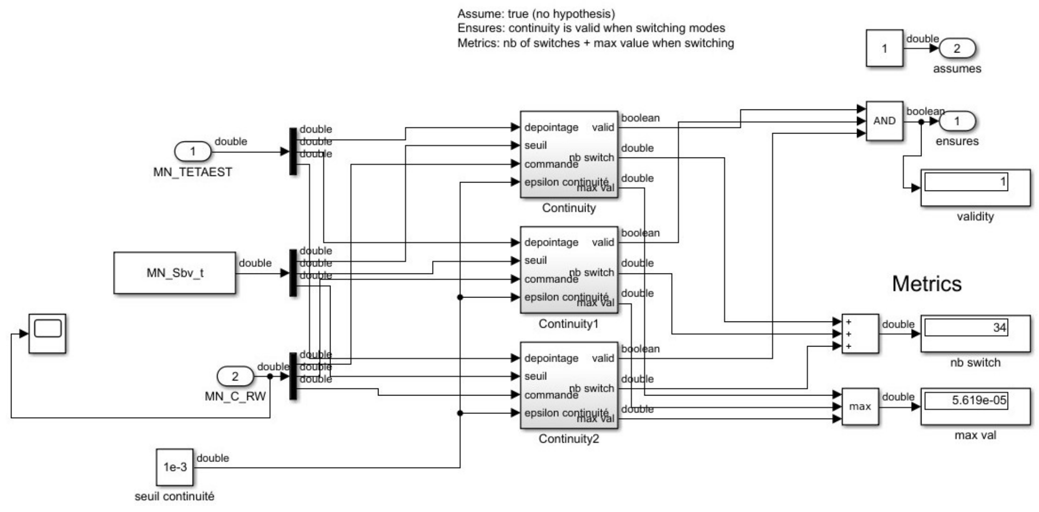


Figure 1.7: Example of a specification

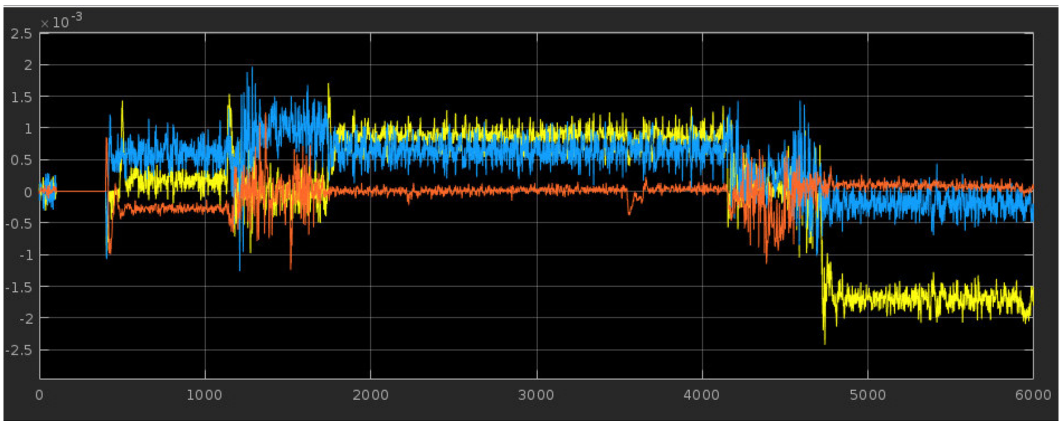


Figure 1.8: One run of the example

1.2 Synchronous observers to support V&V activities

Once the specification is formalized, as regular Simulink components, one can rely on them to support numerous verification and validation activities. Let us look at the example in Figure 1.7 to illustrate these various uses.

This observer only focuses on a very local property: depending on some conditions the controller switches between different control laws. This property ensures that the switch is continuous. However simulations performed on the whole controller leave no opportunity to evaluate the validity of this specific property. Figure 1.8 provides one of such run. While one can consider that the global behavior is acceptable, it is important to provide strong arguments for each requirement.

1.2.1 Synthesis of test oracles

Each formalized requirement acts as a test oracle. The synchronous observer defines a predicate. Therefore its boolean output corresponds to the validity of the expressed requirement.

This block is runable and can be used at various levels. As visible in Figure 1.7 additional elements could be added to the model to visualize the status of the property. In this specific simulation run, the positive value of the output shows that the property was valid during all the execution of that single test.

In addition, since our CoCoSim framework is capable of producing C code for Simulink models, the observer itself can be compiled to produce code. This opens the opportunity to produce C level or binaries implementing test oracles.

1.2.2 Computation of metrics regarding test suites

When considering a large test suite, it is important to evaluate the validity of each requirement for each test case but also to measure the coverage of the specification. It can happen, for example, that a test case does not activate a specification. The notion of modes in CoCoSpec is appropriate: one need to provide figures regarding the evaluation of each mode by a test suite.

The Figure 1.7 also provides these elements as internal flows. Each simulation will produce some numerical values denoting the activation of the property or some meaningful values. In this specific case we compute the number of mode switches, which was 34 is that run, as well as the maximal value of the discontinuity, which was $5 \cdot 10^{-5}$.

Metrics and coverage of requirements can then be automatized, either at model level, or at code level.

1.2.3 Supporting the generation of test cases

Since the property is expressed in the same language as the model it can be easily expressed in the intermediate language and, eventually in C. For certain class of specifications, eg. blocks limited to boolean and linear integer flows, satisfiability model checkers can search for sequence of inputs activating a given mode or satisfying a given condition. Synthesis of traces, ie. test case, containing real/floats values is much more challenging and requires other techniques.

1.2.4 Consistency of specification

Among the possibilities, let us mention also the evaluation or verification of the consistency of the specification. At the contract level, one can ensure that mode constraints are disjunctive or that the mode partitioning is complete, ie. that their disjunction is always valid.

One can also check the validity of assumes, requires, ensures statements or evaluate whether the expressed predicates are compatible with predicates expressed over subcomponents.

Formalizing requirements as synchronous observers is a powerful approach to support the verification and validation of reactive systems. The specification of requirements as model elements requires a minimal amount of work but produces numerous benefits. It supports existing test activities but also enables the use of formal methods.

Chapter 2

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.

This chapter is structured as follow: first we give a brief overview of the platform. Then we present the installation steps of CoCoSim and its dependencies. A third part illustrates a typical use of the framework on a Simulink model. The next three sections detail the computations performed in the three main phases of the tool and provides a description of each phase features.

2.1 Overview of the platform

2.1.1 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 2.1 outlines the different steps.

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 2.4.1) 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.

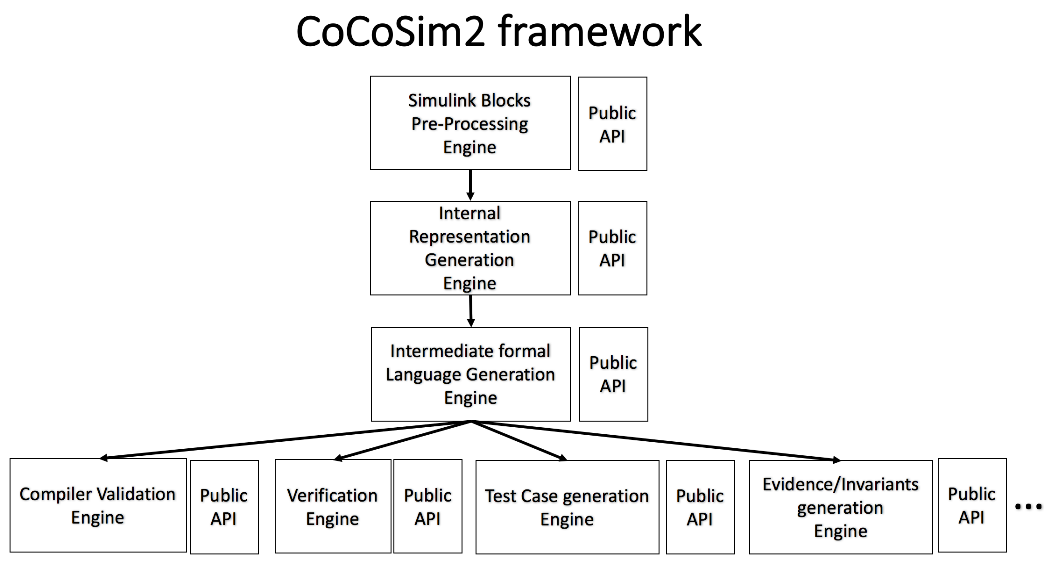

CoCoSim2 framework
Simulink Blocks Pre-Processing Engine
Public API
Internal Representation Generation Engine
Public API
Intermediate formal Language Generation Engine
Public API
Compiler Validation Engine
Public API
Verification Engine
Public API
Test Case generation Engine
Public API
Evidence/Invariants generation Engine
Public API
...

Figure 2.1: CoCoSim framework

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 The 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 2.6.

2.2 Installation

2.2.1 Install dependencies

Dependencies install can be eased with a dedicated script. It requires access to internet since it downloads sources and perform git accesses. It mainly performs two steps: first, it downloads, compiles and installs in a local folder the external tools required by CoCoSim; and then it downloads from a public repositories the standard library for each phase: front-end, middle-end, back-end.

```
$cd "PATH\_TO\_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`.

Tools dependencies. The script detects the missing tools and performs their installation. By default it installs the following ones: Zustre, Spacer, Lustrec, Kind2. They are installed in a local path, `tools/verifiers/osx` if you are Mac user or `tools/verifiers/linux` for Linux platforms.

For example, on a Mac, we obtain the following binaries:

- ZUSTRE : 'cocoSim/tools/verfiers/osx/zustre/bin/zustre'
- LUSTREC: 'cocoSim/tools/verfiers/osx/lustrec/bin/lustrec'
- KIND2 : 'cocoSim/tools/verfiers/osx/kind2/bin/kind2'

One can rely on an existing version of those tools. This can be parametrized in `tools/tools_config.m`, providing path to tools. One can also copy the binary in the `tools/verifiers` folders.

Libraries. CoCoSim distribution is provided with the core algorithm but each phase is parametrized and extensible through libraries. The standard library is available on a public github repository and involves contributions from CMU, University of Iowa, Onera and IRIT. The install script copies these libraries such as the pre-processing standard `src/pp/std_pp` and the internal representation `src/IR/std_IR`.

The dependencies can be summarized as follows:

- MATLAB(c) version **R2015a** or newer
- If you need CoCoSim for verification, at least one of the following solvers should be installed:
  - **Kind2** `http://kind2-mc.github.io/kind2/`
  - **Zustre** `https://github.com/lememta/zustre`
  - **JKind** `https://github.com/agacek/jkind` – for Windows OS users, since it is developped in Java.
  - We recommend **Kind2** and **Zustre**. **Kind2** shows more capabilities in the number of properties solved.
- If you need CoCoSim to analyze the generated C code from the Simulink model, at least on of the following solvers should be installed:
  - **IKOS** `https://ti.arc.nasa.gov/opensource/ikos/`
    - In addition to IKOS, you need to install whole **LLVM**
      - `https://github.com/travitch/whole-program-llvm`
      - Or, using pip: "pip install wllvm"
  - **SeaHorn** `http://seahorn.github.io/`
- Python2.7

Configuration. Edit `cocosim_config.m` and follow the commented instructions.

2.3 Run CoCoSim

2.3.1 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.

Some Simulink models use external user-defined libraries. These constructs are not yet supported and CoCoSim requires them to be *unlinked*. Another construct, which is supported by CoCoSim, is the Model Referencing blocks: a model could be used as sub-system, as a reference to the other model. For simulation and code generation, the referenced model effectively replaces the Model block that references it. Like libraries, referenced models allow you to use the same capability repeatedly without having to redefine it. Referenced models provide several advantages that are unavailable with subsystems and/or library blocks (See Model Referencing[1] in Simulink documentation.).

2.3.2 Overview of CoCoSim in practice

To start using CoCoSim follow these steps or watch `https://youtu.be/dcs8GOeFI9c`:

1. Launch Matlab
2. If you did not install CoCoSim via toolbox:
   - Make sure folder 'cocosim/' is added to your path
3. Call in Matlab command window "'start_cocosim"'
4. Open your Simulink model
5. Load models constants (as a .m or .mat file) into Matlab Workspace.
6. Make sure your model can be compiled
7. In your Simulink model go to Tools/CoCoSim
   - A first step checks that all model blocks could be properly processed: **"Check unsupported block"**. As a result it produces the list of unsupported blocks.
   - A second step ensures that the produced intermediate form in Lustre is compatible with the initial model: **"Compiler Validation"**
     - Some blocks can be abstracted and give different outputs, in that case check the generated Lustre code.

[1]`https://www.mathworks.com/help/simulink/model-reference.html`

- If you want to **verify your model requirements**
  (a) First *"Create Property"* to introduce a specification block.
    - a GUI helper eases the definition of the block, provided with
      * the target subsystem
      * the inputs/outputs required to express the synchronous observer
      * a unique name
      * some assumptions on inputs, if this is required.
    - once the observer is created, define its content within the subsystem.

  (b) Specified properties can be validated with the menu option *"Verify properties using ..."* when selecting one of the proposed solvers.
  (c) Make sure the selected solver is installed (e.g. Zustre, Kind2 and or JKind) and configured in `tools/tools_config.m`.
  (d) A GUI will pop up and provide feedback of verification process.
- If you want to **analyze the C code generated by Simulink Coder** of your model.
  (a) Go to *"Analyze C generated code"*
  (b) Choose one of the proposed analyzer (e.g. IKOS or SeaHorn)
  (c) make sure they are installed and configured in *config.m*.
- If you want to **View generated Invariants**
  - It only works with Zustre for the moment. Run Zustre verifier on your model.
  - Then click on *"View generated Invariants"*
- You can also **Compile your model to C or Rust** by clicking on generate code and choose the target language.

2.4 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.

2.4.1 Preprocessing blocks

The idea behind the pre-processing (as described in Fig. 2.2) is to transform a Simulink model with complex blocks to one that uses basic Simulink blocks.

For example Fig 2.3 illustrates the pre-processing of a Saturation block: such a block can, without loss of precision, be transformed into a combination of min and max operators. Fig. 2.4 proposes the expression of a `TransferFcn` block as a sub-system describing its linear system realization. This process can be applied on a discrete transfer function without loss of precision, while its application to continuous transfer function can be appropriate or not depending on the context.

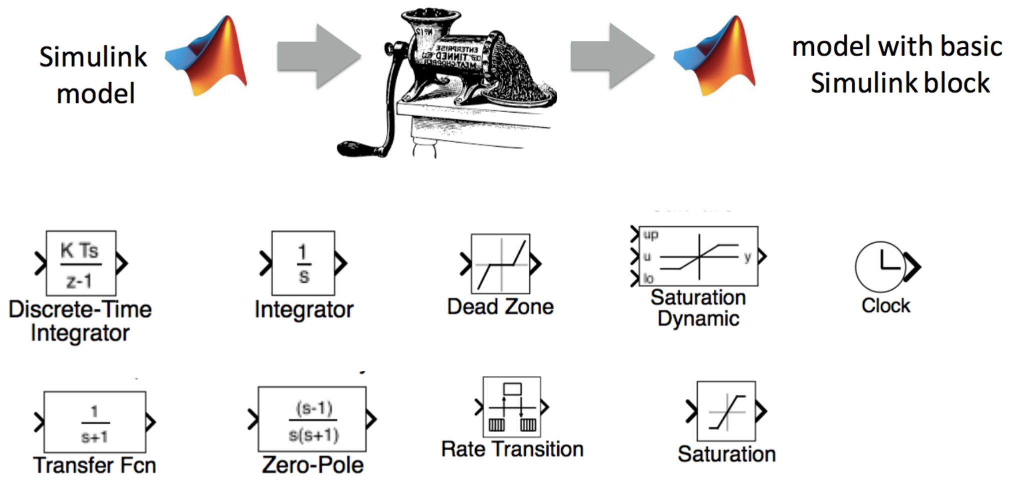


Figure 2.2: Some of pre-processing libraries.

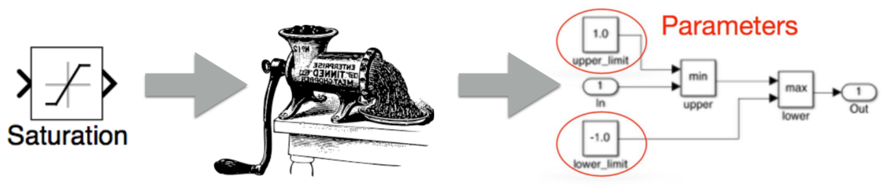
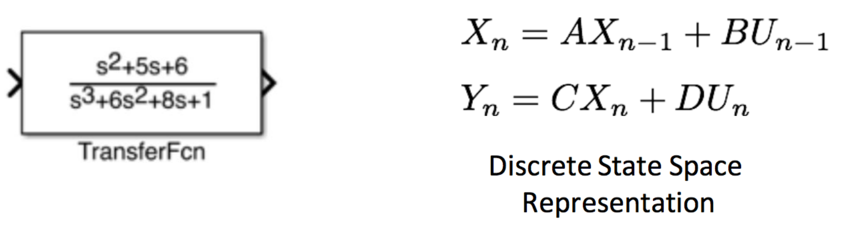
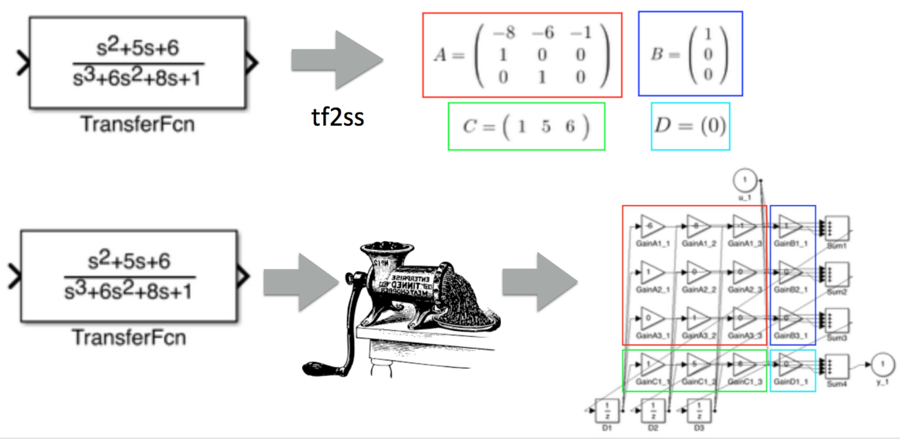


Figure 2.3: Example of simplifying the Saturation block.


s2+5s+6 / s3+6s2+8s+1
TransferFcn
$X_n = AX_{n-1} + BU_{n-1}$
$Y_n = CX_n + DU_n$
Discrete State Space Representation

(a)


s2+5s+6 / s3+6s2+8s+1
TransferFcn
tf2ss
$A = \begin{pmatrix} -8 & -6 & -1 \\ 1 & 0 & 0 \\ 0 & 1 & 0 \end{pmatrix}$
$B = \begin{pmatrix} 1 \\ 0 \\ 0 \end{pmatrix}$
$C = \begin{pmatrix} 1 & 5 & 6 \end{pmatrix}$
$D = (0)$

(b)

Figure 2.4: Example of pre-processing the TransferFcn block.

We present here the Matlab function performing the pre-processing. Then we present how the pre-processing can be tuned and extended.

2.4.1.1 Matlab Function

CoCoSim pre-processing which converts a Simulink model into a CoCoSim-friendly one is automatically called when using the complete CoCoSim framework. It could also be called independently on a given model to simplify it. The function `cocosim_pp` can be used as follows:

- `cocosim_pp('path_to_model.slx')`: the model is processed and its processed copy is produced in the file `path_to_model_PP.slx`. Note that if the given file is already a pre-processed file `*_PP.slx` the modification is performed in place.
- `cocosim_pp('path_to_model.slx','constant_filename.m')`: If the Simulink model needs the definition of constants to be fully defined, they can be provided through a `.m` file.
- `cocosim_pp('path_to_model.slx','model_constants.m','verif')` with all its paremeters or `cocosim_pp('path_to_model.slx',,'verif')` without the constant file part: a third option parameter can be used to validate the pre-processing. It generates a Simulink model containing the original model and the pre-processed model for validation.

  The function `cocosim_pp` will then generate, in addition to the pre-processed model, a new model containing both models as well as means of simulating both models outputs for the same inputs. The feature is used for non regression tests and can be used to validate a specific pre-processing.

In all cases, `cocosim_pp` needs to compile the model during its execution. It relies on the type inference of the simulation engine to discover the type of variables. Indeed most users of Simulink do not specify types but rather rely on the -1 value denoting inheritance, ie. type inference. If the pre-processing lacks the values of certain parameters, it may fail.

Matlab r2014b or newer is required because of its python capabilities. Calling `cocosim_pp` in an older version of Matlab is feasible but any call of the Python parser script will fail, while the computation is completed, errors being reported to the user. Some cases are postponed and will be handled by CoCoSim translator, such as constants, gain block, discrete integrator and Fcn blocks.

2.4.1.2 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 one developped within NASA. Both libraries are complimentary and perform different pre-processing.

2.4.1.2.1 Configure pre-processing order programmatically

The function call order is specified in `main/pp_order.m`. The pre-processing tree is presented in Fig. 2.5a.

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. 2.5b) 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:

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

2.4.1.2.2 GUI-order configuration

An configuration GUI (Fig. 2.6) 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.

2.4.1.3 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 2.4.1.2.

Existing libraries. Please refer to section A for more details.

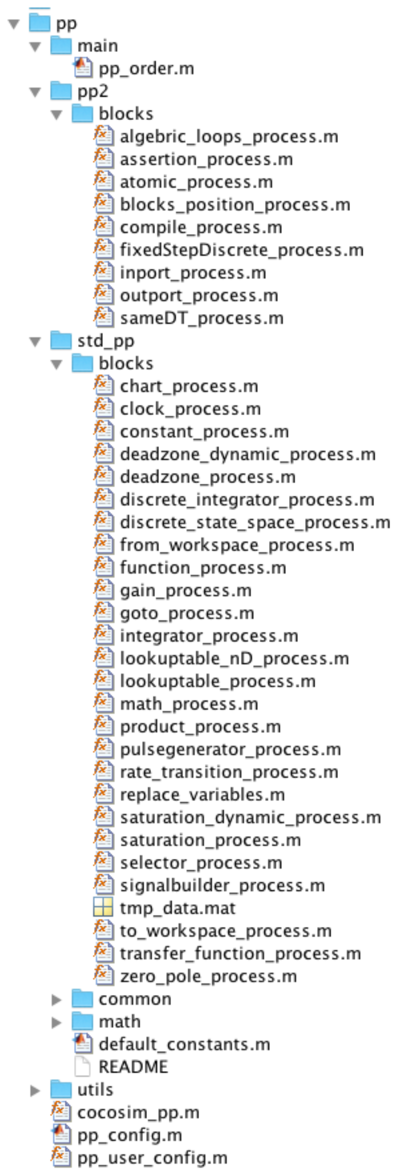
2.4.2 Internal Representation of Simulink/Stateflow

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. We handle a restricted subset of Simulink components, cf Sec. 2.4.2.1, a subset of Stateflow automata, cf Sec. 2.4.2.2 and some basic Matlab expressions, cf Sec. 2.4.2.3.

2.4.2.1 Simulink IR

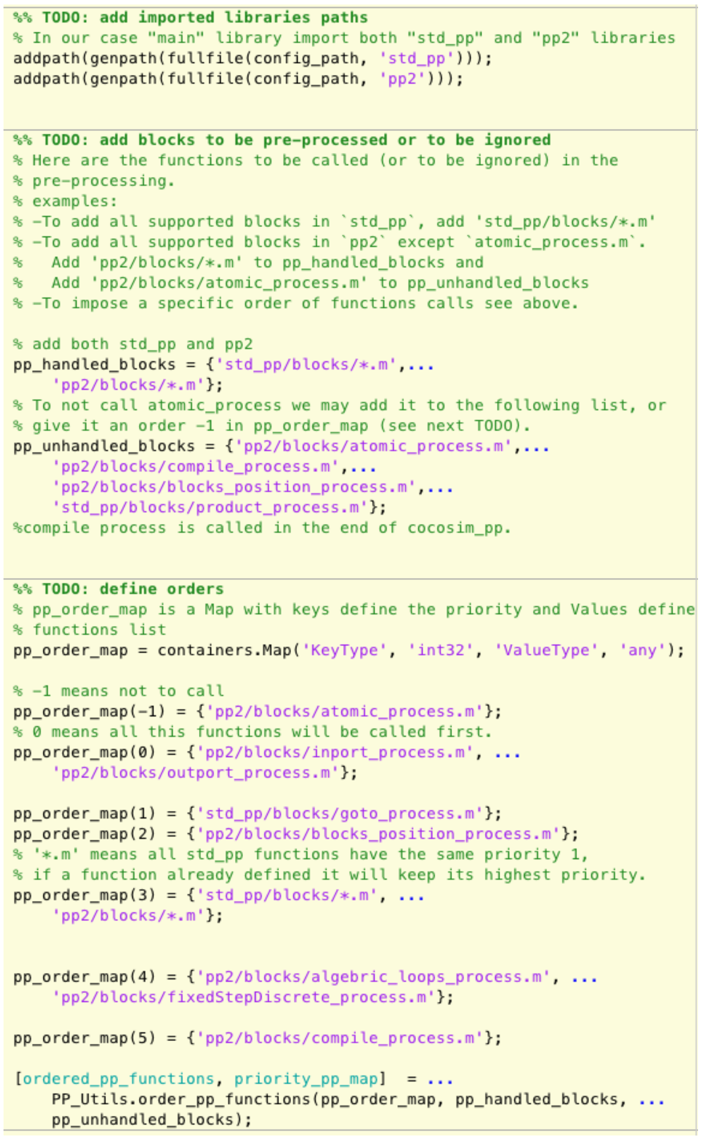
The function `cocosim_IR` produces the IR associated to a given Simulink model. The function's signature is as follow:

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



```
▼ pp
  ▼ main
      pp_order.m
  ▼ pp2
    ▼ blocks
        algebric_loops_process.m
        assertion_process.m
        atomic_process.m
        blocks_position_process.m
        compile_process.m
        fixedStepDiscrete_process.m
        inport_process.m
        outport_process.m
        sameDT_process.m
  ▼ std_pp
    ▼ blocks
        chart_process.m
        clock_process.m
        constant_process.m
        deadzone_dynamic_process.m
        deadzone_process.m
        discrete_integrator_process.m
        discrete_state_space_process.m
        from_workspace_process.m
        function_process.m
        gain_process.m
        goto_process.m
        integrator_process.m
        lookuptable_nD_process.m
        lookuptable_process.m
        math_process.m
        product_process.m
        pulsegenerator_process.m
        rate_transition_process.m
        replace_variables.m
        saturation_dynamic_process.m
        saturation_process.m
        selector_process.m
        signalbuilder_process.m
        tmp_data.mat
        to_workspace_process.m
        transfer_function_process.m
        zero_pole_process.m
    ▶ common
    ▶ math
      default_constants.m
      README
  ▶ utils
    cocosim_pp.m
    pp_config.m
    pp_user_config.m
```

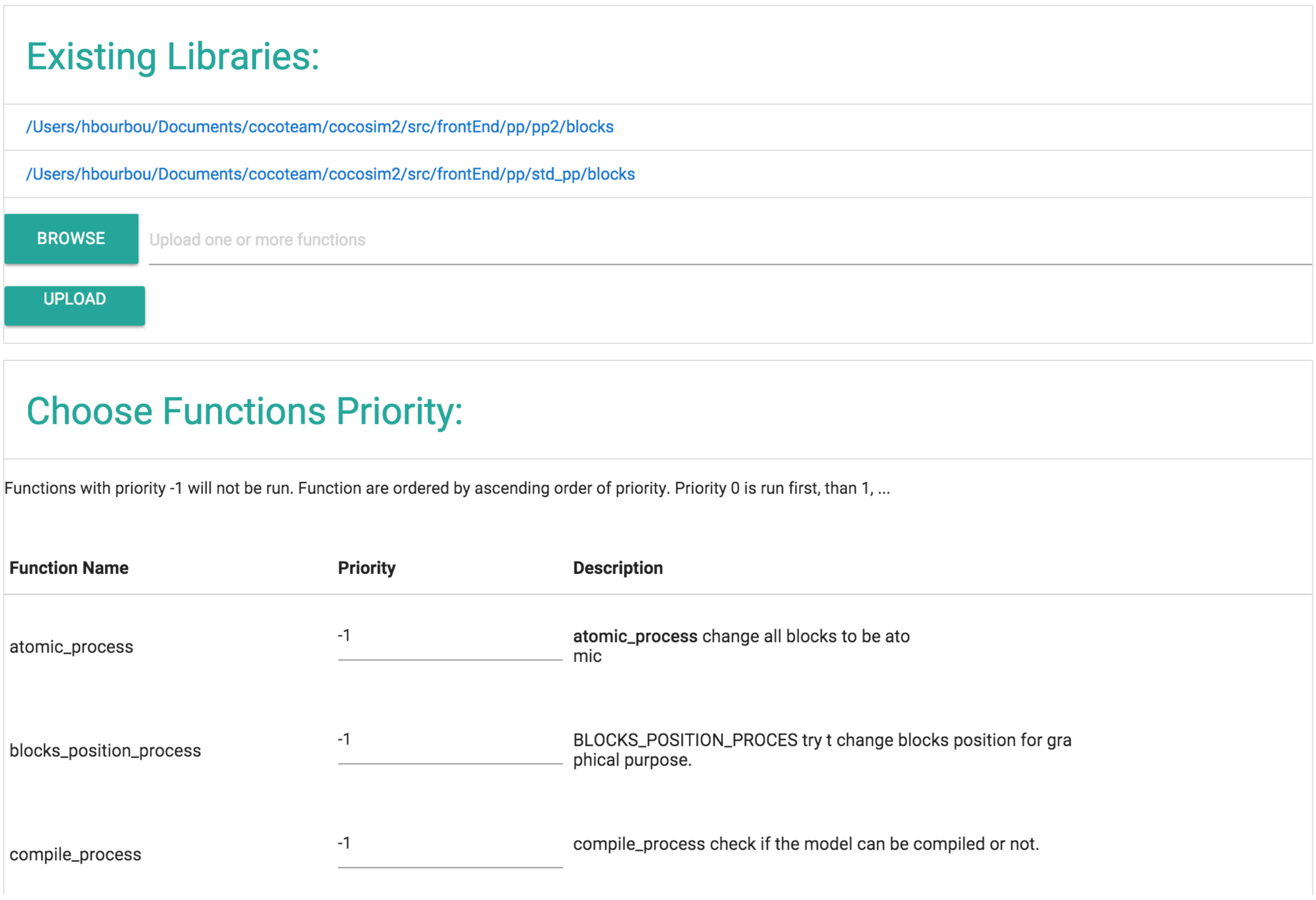
(a) Pre-Processing tree



```
%% 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 2.5: Pre-Processing parametrization.


Existing Libraries:

/Users/hbourbou/Documents/cocoteam/cocosim2/src/frontEnd/pp/pp2/blocks

/Users/hbourbou/Documents/cocoteam/cocosim2/src/frontEnd/pp/std_pp/blocks

Choose Functions Priority:

Functions with priority -1 will not be run. Function are ordered by ascending order of priority. Priority 0 is run first, than 1, ...

Function Name	Priority	Description
atomic_process	-1	**atomic_process** change all blocks to be ato mic
blocks_position_process	-1	BLOCKS_POSITION_PROCES try t change blocks position for gra phical purpose.
compile_process	-1	compile_process check if the model can be compiled or not.

Figure 2.6: Pre-Processing user configuration interface

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 2.7 presents a simple

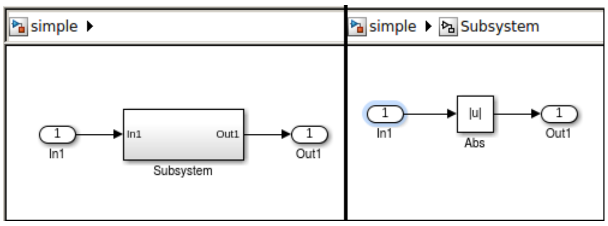


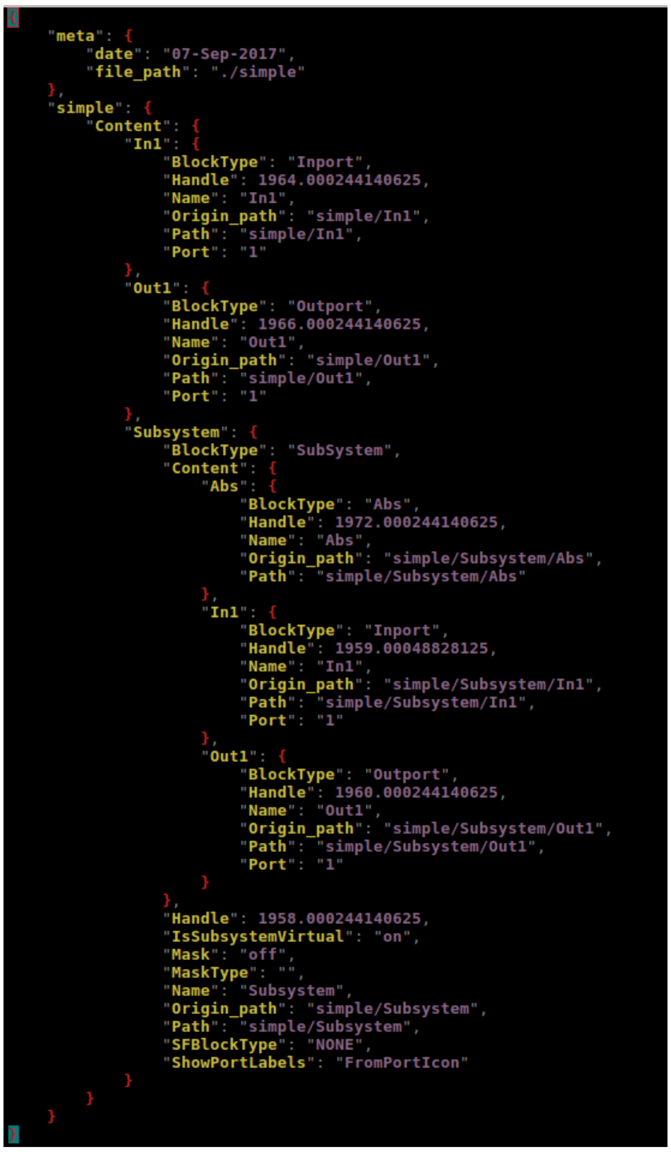
Figure 2.7: Absolute value subsystem.

Simulink model computing the absolute value of an input flow and Figure 2.8 its IR.

2.4.2.2 Stateflow IR

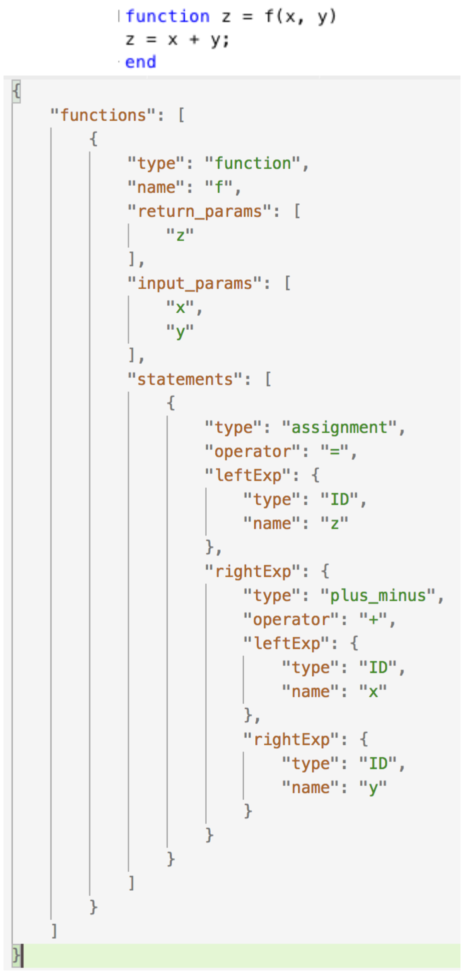
A stateflow chart is represented as a Program with the following attributes:

- 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.



```
{
    "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 2.8: Absolute value subsystem intermediate representation.



```
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 2.9: Internal representation of a simple Matlab code in Json format.

- 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.

2.4.2.3 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. 2.9 shows an example of a simple Matlab function and its equivalent in Json format.

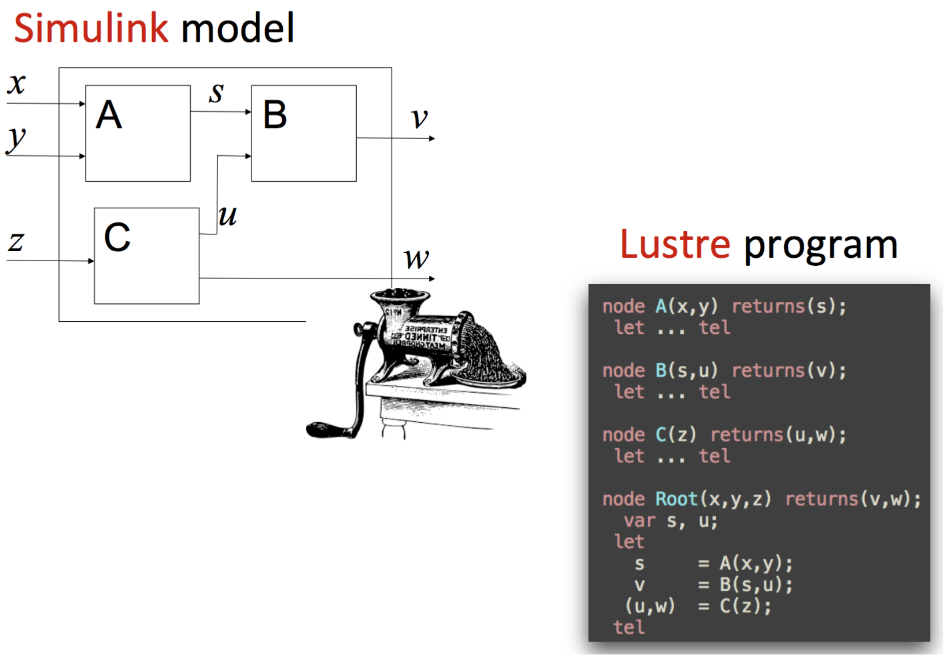
Note that the subset of Matlab expressions accepted of extremelly limited and cannot involve sophisticated Matlab functions.

2.5 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.

2.5.1 Lustre compiler

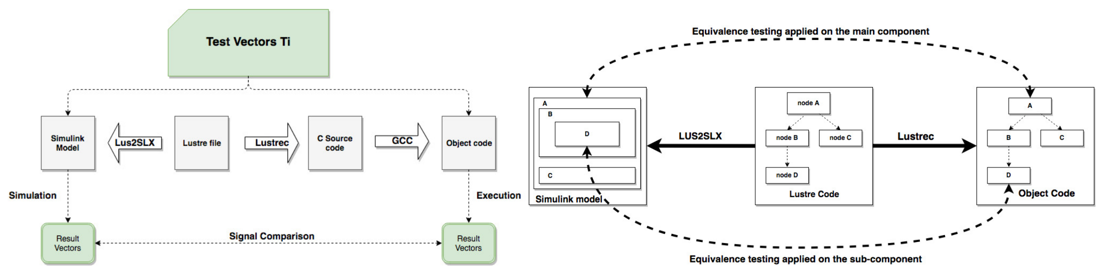
CoCoSim translate discrete-time Simulink to Lustre code. The main goal is to preserve Simulink semantics. The compilation from Simulink to Lustre is hierarchical block-by-block compilation. Every Subsystem (treated as atomic) will be translated in a Lustre node.



```
node A(x,y) returns(s);
 let ... tel

node B(s,u) returns(v);
 let ... tel

node C(z) returns(u,w);
 let ... tel

node Root(x,y,z) returns(v,w);
  var s, u;
 let
   s     = A(x,y);
   v     = B(s,u);
  (u,w)  = C(z);
 tel
```

Figure 2.10: 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 2.11: (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* [5, 4], 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.

2.5.2 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.

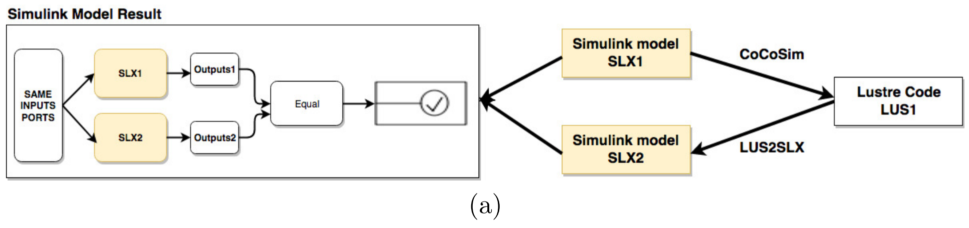
There are two type of validation, Translator validation and translation validation. The first class is based on validating the compiler for all possible executions. The second is interested on one individual translation. The choice of CoCoSim is to focus on the second approach in which we validate the translation after each execution. This validation is integrated in the process of development and performed automatically [6].

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.

2.5.2.1 Equivalence testing

Equivalence testing is based on numeric equivalence between the target and the source. Figure 2.11 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 2.11a) 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 2.11b 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



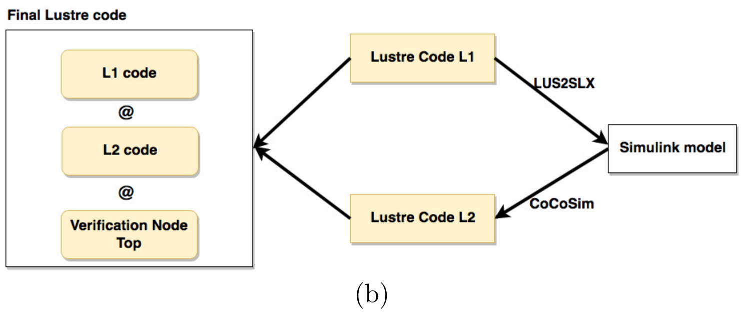


Figure 2.12: (a) The equivalence checking applied on Simulink level. (b) Equivalence checking applied on Lustre level

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.

2.5.2.2 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 2.12.

In a first pattern, presented in Figure 2.12a, 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 2.12b, 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.

2.5.3 Supported Blocks

In addition to the blocks pre-processed (see section 2.4.1) we translate the following blocks directly to Lustre code.

Discontinuities. Saturate, Dynamic Saturation,

Discrete. DiscreteIntegrator, DiscreteStateSpace, UnitDelay, Delay, Memory, DiscreteZeroPole,

Logic & Bit Operations. RelationalOperator, Compare To Constant, Compare To Zero' Bitwise Operator, Detect Change, Detect Increase, Detect Decrease, Detect Rise Positive, Detect Rise Nonnegative, Detect Fall Negative, Detect Fall Nonpositive

Lookup Tables. LookupNDDirect,

Math Operations. Gain, Abs, Product, Polyval, MinMax, Sum, Bias, Concatenate, Reshape, Trigonometry, DotProduct, Math, Sqrt, Assignment, Signum,

Ports & Subsystems. SubSystem, If, ForIterator, SwitchCase, ActionPort, TriggerPort, EnablePort, ModelReference,

Signal Attributes. DataTypeConversion, SignalSpecification, SignalConversion,

Signal Routing. Switch, MultiPortSwitch, Demux, Goto, From, Merge, Mux, BusSelector, BusCreator, BusAssignment, Selector,

Sinks. Terminator, ToWorkspace

Sources. Inport, Outport, Constant, Step, FromWorkspace,

Stateflow. Chart (see `https://ti.arc.nasa.gov/publications/42621/download/`)

User-Defined Functions. Fcn, S-Function,

Utilites. Cross Product, Create 3x3 Matrix,

2.6 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 enfines (cf. Sect. 2.6.1), code generation (cf. Sect. 2.6.2), invariant generation (cf. Sect. 2.6.3), test-case generation and coverage evaluation (cf. Sect. 2.6.4), Support of specification, including Lustre to Simulink translation (cf. Sect. 2.6.5), compilation validation (cf. Sect. 2.6.6), and code analysis (cf. Sect. 2.6.7).

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.

2.6.1 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.

2.6.2 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 2.5.2.

2.6.3 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.

2.6.4 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 2.13 presents the set of approaches that can be run at Lustre level and then propagated back as test case at the Simulink level.

2.6.4.1 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.

2.6.4.2 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.

2.6.4.3 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 [2] 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:

$$\left\{ \begin{array}{l} c_i \wedge (P(c_1, \ldots, c_n) \oplus P(c_1, \ldots, c_{i-1}, \neg c_i, c_{i+1}, \ldots, c_n)) \ , \\ \neg c_i \wedge (P(c_1, \ldots, c_n) \oplus P(c_1, \ldots, c_{i-1}, \neg c_i, c_{i+1}, \ldots, c_n)) \end{array} \right\}_{i \in 1..n} \tag{2.1}$$

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.

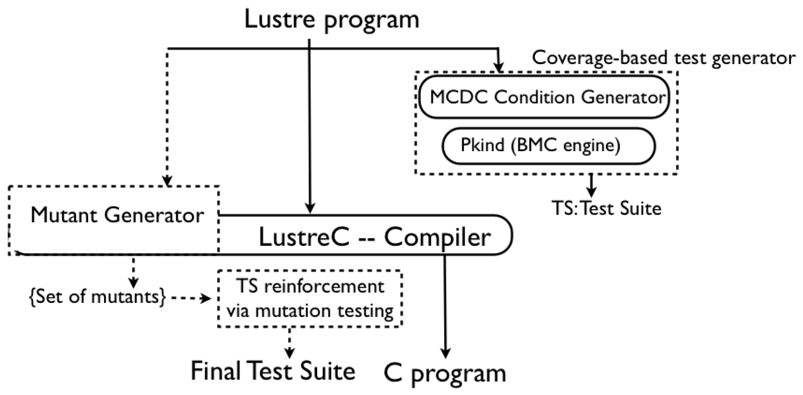


Figure 2.13: Combinaison of approaches to support test generation.

2.6.5 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 2.5.2. It can also be used to annotate Simulink models with specification written as Lustre models.

2.6.6 Compiler Validation

This back-end is interested in validating the compiler from Simulink to Lustre. See section 2.5.2.

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

2.6.7 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] Albert Benveniste et al. "The synchronous languages 12 years later". In: *IEEE* 91.1 (2003), pp. 64–83.

[2] Paul Caspi et al. "Lustre: A Declarative Language for Programming Synchronous Systems". In: *POPL'87*. 1987, pp. 178–188.

[3] 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`.

[4] 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`.

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

[6] 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`.

[7] Nicolas Halbwachs, Fabienne Lagnier, and Pascal Raymond. "Synchronous Observers and the Verification of Reactive Systems". In: *AMAST'93*. 1993, pp. 83–96.

[8] N. Halbwachs et al. "The synchronous dataflow programming language LUSTRE". In: *Proceedings of the IEEE*. 1991, pp. 1305–1320.

[9] C. A. R. Hoare. "An Axiomatic Basis for Computer Programming". In: *Commun. ACM* 12.10 (1969), pp. 576–580. DOI: `10.1145/363235.363259`. URL: `http://doi.acm.org/10.1145/363235.363259`.

[10] MathWorks. *Stateflow*. `http://www.mathworks.com/products/stateflow/`.

[11] Mathworks. *Simulink*. `http://www.mathorks.com/products/simulink/`.

[12] John Rushby. "The Versatile Synchronous Observer". In: *Specification, Algebra, and Software, A Festschrift Symposium in Honor of Kokichi Futatsugi*. LNCS. 2014.

[13] Martin Westhead, Simin Nadjm-tehrani, and Forrest Hill Edinburgh U. K. “Verification of Embedded Systems Using Synchronous Observers”. In: *In Proceedings of the 4th International Conference on Formal Techniques in Real-time and Fault-tolerant Systems, LNCS 1135*. Springer Verlag, 1996, pp. 405–419.

Appendices

Appendix A

Pre-processing libraries

A.1 Supported blocks

The blocks registered in this section are converted into a CoCoSim-friendly equivalent, but certain option may not be handled by the translation. Some of the blocks are complex and need a very customized handling, the blocks to replace them are generated by matlab scripts. Other blocks are very similar each time you find them in a model, these blocks are replaced by generic blocks that are contained in the file `gal_lib.slx` contained in the `cocosim/lib` folder.

- **Clock or Digital clock** : Replace by a CoCoSim-friendly block provided in the `gal_lib.slx` located in `src/pp/lib/common`.
  We recommend to use Digital clock as it is discrete.

  Details : The clock is basically an integrator of the constant Sample time of the Simulink model.

- **Integrator** : Replace by a CoCoSim-friendly block provided in the `gal_lib.slx` located in `src/pp/lib/common`.
  Supported options :

  - `InitialConditionSource`: internal/external
  - `External reset`: we support all reset options

  Details : We do not recommend using Continuous integrator as its semantic is different from the discrete one, so the outputs are not the same. The best way is to use directly the discrete integrator.

- **Discrete Integrator** : Replace by a CoCoSim-friendly block provided in the `gal_lib.slx` located in `src/pp/lib/common`.
  Supported options :

  - `Integration method`: Integration: Forward Euler

  - `InitialConditionSource`: internal/external
  - `External reset`: we support all reset options

  Details : The blocks provided in `gal_lib.slx` have the expected behavior if the sample time of sum blocks are respected by CoCoSim, CoCoSim try to use the Simulink model sample time while compilation.

- **Dead Zone** : Replace by a CoCoSim-friendly block provided in the `gal_lib.slx` located in `src/pp/lib/common`.
  Supported options:
  - `LowerValue`
  - `UpperValue`

- **Dynamic Dead Zone** : Replace by a CoCoSim-friendly block provided in the `gal_lib.slx` located in `src/pp/lib/common`.
  Supported options:
  - `LowerValue`
  - `UpperValue`

  Details : To modify the behavior of these blocks, you must modify their replacements in `gal_lib.slx`.

- **DiscreteStateSpace** : Replace by a CoCoSim-friendly block generated from the state space representation.
  Dynamic Saturation : Replace by a CoCoSim-friendly block provided in the `gal_lib.slx` located in `src/pp/lib/common`.
  Supported options:
  - `LowerLimit`
  - `UpperLimit`

  Details : To modify the behavior of these blocks, you must modify their replacements in `gal_lib.slx`.

- **Function** : Replaced by a subsystem containing the calculation made by blocks. In case of the pre-processing does not succeed, we have a second choice of translating directly the Function expression using regular expression. The matlab function that handles this block is in `src/middleEnd/write_function_block`.

  Details : a python parser called from Matlab gives a tree of the mathematical expression which is converted into blocks with only variables and figures stored

in constant blocks.
The the parser recognize any function by default (of the form 'func(arg)'), the handling of new functions has to be added in `expression_process` located in `src/pp/lib/math`.

- **FromWorkspace** : Replaced by an input in the first level system.
  The goal is to provide this input as a variable for PKind tests.

- **Goto/From** : Create lines between blocks that have the same "tag".
  Supported options :

  – `TagVisibility`: global/local

  Details : This function also create inputs/outputs to some subsystems in order to link together the blocks.

- **Lookup tables** : Replaced by a subsystem providing an interpolation based on table provided (from the initial block) during the process. Supported options:

  – `InputValues`

  – `Table data`

  – `Lookup method: Interpolation-Use End Values`

- **Lookup tables nD** : Replaced by a subsystem providing an interpolation based on table provided (from the initial block) during the process. Supported options:

  – `NamberOfTableDimensions` : 1

  – `BreabpointsForDimension1`

  – `Table`

  – `Lookup method: Interpolation-Use End Values`

  Details : In this block we only support **Interpolation** method it means we perform linear interpolation but we do not extrapolate outside the end points of the input vector. Instead the block uses the end values.
  One of the possible enhancement would be to handle tables with more than 1 dimension.

- **Math** : Replace Math blocks with yet unsupported operations.
  Operations handled by the script (not handled natively by CoCoSim):

  – `magnitude^2`: replaced by a `pow` using the parsing operation on 'u^2'

- **RateTransition** : Replaced by a CoCoSim-friendly block provided in the `gal_lib.slx` located in `src/pp/lib/common`.
  Supported options:

  - `OutPortSampleTime`
  - `X0`

  Details : The block allow to output the `X0` initial condition during the first step of execution of the Lustre code. The output is also updated at a `OutPortSampleTime` rate, this behavior will be effective when CoCoSim will consider sample times provided in each block. If CoCoSim don't consider different sample times, this block has no purpose other than the initial condition.

- **Saturation** : Replace by a CoCoSim-friendly block provided in the `gal_lib.slx` located in `src/pp/lib/common`.
  Supported options:

  - `LowerLimit`
  - `UpperLimit`

- **Selector** : Replace by a subsystem block containing a demux of the input, a mux of the output and link to the required inputs.
  Supported options:

  - `IndexMode`: Zero-based/One-based
  - `IndexOptionArray`: 'Index vercor (dialog)' only supported.
    new options may easily be added.

- **ToWorkspace** : Replaced by an output in the first level system.

- **TransferFunction** : Replace by a CoCoSim-friendly block generated from the state space representation provided by Matlab from the numerator and denominator of the function.

- **Zero-Pole** : Replace by a CoCoSim-friendly block generated from the state space representation provided by Matlab from the poles and zeros of the function.

Remarks

The blocks `FromWorkspace` are replaced by inports in the top level layer of the model, and blocks `ToWorkspace` are replaced by outports in the top level layer of the model. But these blocks can sometimes be used to load some precise data into the model (curves from experiments for example), and in this case, the model generated won't support this input of information. The block should be replaced by a lookup table with a clock in input for example if we want this data to go threw the compilation. Or the user can make assumptions about its input (see section 1.2) to give an abstraction on the inport and use an inport instead.