CoCoSim, a code generation framework for control/command applications

An overview of CoCoSim for multi-periodic discrete Simulink models

Hamza Bourbouh, Pierre-Loïc Garoche, Thomas Loquen
Eric Noulard, Claire Pagetti

Abstract

We present CoCoSim, a framework to support the design, code generation and analysis of discrete dataflow model expressed in Simulink. In this work, we specifically focus on the analysis and code generation of multi-periodic systems. For that CoCoSim provides two complementary approaches: the first amounts to encode the multi-periodic semantics in a pure-synchronous one – à la Lustre–, enabling the use of model checker for verifying properties. The second provides a faithful code generation into multiple communicating (mono)synchronous components – à la Prelude– that can be then simulated or embedded in the final platform with any real-time scheduler. These approaches have been experimented in various settings.

1 Introduction

1.1 Context

Safety-critical systems design requires a thorough development process including formal verification and correct by construction behaviour. In that area, Model-Based Design has been widely used for software development. Such an approach offers the refinement of a system from High Level Requirements down to the embedded code while having an executable model at different stages. Matlab/Simulink[1] from MathWorks, is a *de facto* model-based design standard in industry, offering verification and code generation means.

Nonetheless, other development frameworks are used in addition in some industries, such as aeronautic, railways or space. Indeed, control/command applications have received a particular attention over the years and several synchronous programming languages such as Esterel [2], Lustre [8] or Signal [38] have been defined to help their design. Scade [12] is an industrial and DO 178C qualified Lustre-based framework that provides strong guarantees and proofs well appreciated, in particular for certification.

Offering frameworks linking Simulink and synchronous approaches is thus appealing. CoCoSim belongs to this category as it is an *open source* tool that translates Simulink specification in Lustre while preserving semantics and providing many associated traceability or test capabilities. In this paper, we describe an extension of CoCoSim to allow the safe translation and verification from multi-periodic systems to Lustre/Prelude programs.

1.2 Contributions

The CoCoSim approach to deal with multi-periodic systems is highlighted in the Fig. 1.

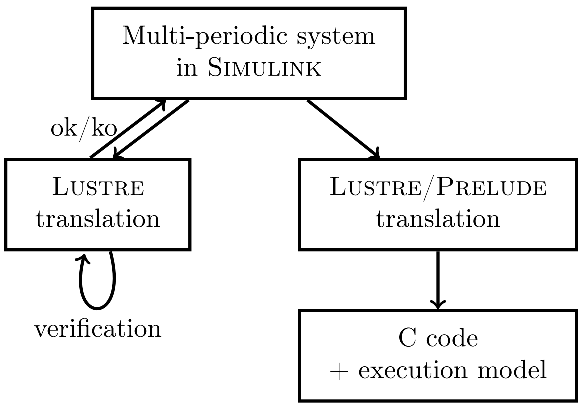


Figure 1: CoCoSim overview

First, we need to define precisely the semantics of multi-periodic systems in Simulink and connect it to the one of synchronous programming. In synchronous languages, execution time is neglected while each computation is performed repetitively, e.g., every *ts* seconds. In Simulink, most discrete subsets of blocks are fitted with a synchronous se-

[1] https://www.mathworks.com/products/simulink.html

mantics, but the case of multi-periodic systems is more complex and requires an analysis of the internal semantics.

Once the semantics has been defined, the second contribution is the extension of CoCoSim current Lustre translation in order to encode the multi-periodic communication with classical Lustre over- and sub-sample operators (left hand side of Fig. 1). This amounts to express the whole system on a base clock. This Lustre model is then used to perform formal analysis using SMT-based model-checking. While required to properly analyze the full system, this encoding is not efficient for execution.

Once the verification is valid, the last step is the efficient code generation (right hand side of Fig. 1). Each synchronous component is translated as a Lustre model which will eventually be compiled into C code, while aggregating nodes, mixing different clocks or execution rates are expressed as Prelude programs. Prelude [30] is a synchronous language that has been defined to program multi-periodic applications. From a Prelude program, the compiler generates a set of classical real-time tasks and many *predictable* implementations have been proposed for multi- and many-core architectures [32].

1.3 Outline

We start by presenting an overview of CoCoSim framework (Section 2) for mono-periodic discrete Simulink models. We then define the specification of multi-periodic applications with Simulink (Section 3). We detail the multi-periodic extension of CoCoSim (Section 4). Last, Section 5 presents some experiments settings and two detailed use cases.

2 Overview of mono-periodic CoCoSim

CoCoSim is a highly automated framework for verification and code generation of Simulink/Stateflow models. It consists of an open architecture, allowing the integration of different analyses. CoCoSim is structured as a compiler, sequencing a series 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 outlines the different steps.

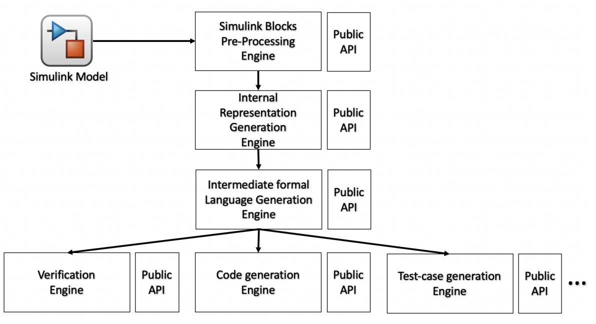

Simulink Model
Simulink Blocks Pre-Processing Engine
Public API
Internal Representation Generation Engine
Public API
Intermediate formal Language Generation Engine
Public API
Verification Engine
Public API
Code generation Engine
Public API
Test-case generation Engine
Public API
...

Figure 2: CoCoSim framework

2.1 Formal semantic

CoCoSim provides a **formal semantic** of a well defined subset of Simulink/Stateflow blocks. This formal representation will permit the use of formal verification methods and code generation.

CoCoSim starts first by simplifying some complex blocks into a set of basic blocks. Then an internal representation of the model is generated containing all information needed for code generation. Based on the work of Caspi et al. [7], Gene-Auto [33, 39] and P [4] projects, CoCoSim translates modularly the pre-processed mono-periodic Simulink model into an equivalent Lustre model. The translator is developed using a visitor pattern, each Simulink Subsystem is translated into a Lustre node, each instance of a Subsystem is translated into a Lustre node call and each Simulink atomic block is represented by a local Lustre equation defining the semantic of the block. The generated Lustre model has the same hierarchy as the original Simulink model and preserves the initial semantic.

CoCoSim is **customizable** and **configurable**. Indeed, it supports most of frequently used Simulink blocks libraries (around 100 blocks) and new blocks can be easily supported.

2.2 Supported analyses

Once a formal representation of Simulink model is generated, CoCoSim is connected to a set of external tools to provide code generation, formal verification or test case generation. The toolchain is **highly automated** as all the steps of verification or code generation are automated.

The goal of the CoCoSim framework is to ease the application of formal methods and analysis of Simulink-based systems. The external tools 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. A set of Matlab functions libraries are provided by CoCoSim to ease the integration of tools analysis results. Eg. displaying the counterexample at the Simulink level, importing test harness as a Signal builder or create a new test-harness model, generating HTML reports or other helpful support functions.

All CoCoSim analyses are performed on the compiled artifact and the results are expressed back at Simulink level thanks to **traceability** information. We sketch here the features of the connected tools. At the current moment all tools are opensource and freely available. It **scales** well with large models, therefore various verification techniques and compositional reasoning can be used.

2.3 Formal Verification: SMT-based model checking

Once requirements have been expressed using CoCoSim library and attached to the Simulink model, different tools can perform SMT-based model checking and check their validity. In case the property supplied is falsified, CoCoSim provides means to simulate the counterexample trace in the Simulink environment. Currently, CoCoSim is connected to three verifiers.

First, Kind2 [10] is a powerful tool that implement multiple algorithms including k-induction [36] and IC3/PDR [5] as well as on-the-fly invariant generation. All of these can be performed with various SMT solvers: CVC4, Z3, Yices.

Second, Zustre [21] relies on the LustreC [20] modular compiler. The input Lustre model is compiled in a Horn encoding [19, 22] describing the transition relation. The hierarchy of the input model is preserved: each Lustre is associated to a local Horn description of the computation. This transition system along with a similar expression of its requirements is analyzed with Spacer [26, 27], a PDR algorithm integrated in Z3. Thanks to the modular encoding at Horn level and to Spacer, a valid property will produce witnesses as node-local invariants. These invariants are expressed in the feedback to CoCoSim and can be re-introduced in the original Simulink model as fresh annotations.

Third, JKind [16], a similar model checker, developed at Rockwell Collins, is also integrated.

2.4 Code generation:

Some of CoCoSim backends provide code generation. Eg. LustreC [20] is an implementation of the modular compilation scheme [3] used in Scade. It preserves the hierarchy of the initial model, easing the checking of traceability between Lustre and generated C code. LustreC targets mainly C code but extends the general compilation of Lustre to CoCoSpec [9, 13]. LustreC also provides modular compilation with multiple source files, call to external C libraries or externally defined C functions.

Kind2 [10] is, first of all, a model checker but it is capable of producing Rust code from the provided models.

2.5 Test cases generation:

LustreT [17] is based on some compilation stages of LustreC [20]. It provides two different methods to perform test case generation [18].

In the first case a coverage criteria such as MC-DC is expressed as a reachability problem. For example, an atom of a boolean predicate has to be true at some point. Then we check the validity of the negation of that property. Model checker such as Kind2 or Zustre will then perform bounded-model checking or, possibly, exhibit a counter-example; that is, a test case activating that specific criteria. A MC-DC criteria will then be mapped to a large set of such predicates. The test generation process will populate a set of test cases activating each of these conditions.

The second approach relies on the notion of mutants. Usually, mutants are used to evaluate the quality of a test suite. We generate a set of mutant programs and apply different test suites. A good test suite distinguishes valid program from mutants. Here the approach is different. After generating mutants, we use the same bounded-model checking tools to build a test case that will distinguish them. This does not always succeed since some mutations may be invisible, or in dead code. But considering the large set of mutants, this approach is efficient at building test cases.

3 Multi-periodic in Simulink

3.1 Reminder on Simulink

Simulink is a graphical, dataflow programming environment for modeling and simulating dynamical systems. Roughly speaking, in a Simulink model, the user can use blocks, signals and annotations. Each Simulink block implements a given mathematical function or a stateful system specification and has a specific number of inputs and outputs connected with other blocks using Simulink signals. These blocks are either basic blocks from Simulink library (e.g., *Sum*, *UnitDelay*, *Gain*) or a grouping of several blocks into a *Subsystem* block. Simulink diagrams are hierarchical and graphically organized using Subsystems. There are two types of Subsystems, *Virtual Subsystem* that is flatten during Simulation and only used to organize the model at design time and *Atomic Subsystem* that is executed as an atomic unit in the final code.

A discrete SIMULINK model runs on a fixed time step defined with a period π and initial offset θ. A SIMULINK model can be mono-periodic where all blocks runs with the same period, or multi-periodic where blocks can run on different periods. A block is executed and its outputs are updated only when certain execution conditions are satisfied. If these conditions are not met, the block is not executed and its output signals hold their values. MATLAB/SIMULINK defines three types of execution conditions and block:

- Unconditional blocks or subsystems: The block is set with a sample time $D = (\pi_i, \theta_i)$ with period π_i and initial offset θ_i and it is updated only at times $k\pi_i + \theta_i$ for $k \in \mathbb{N}$, whereas, it remains constant during the intervals $[k\pi_i + \theta_i, (k+1)\pi_i + \theta_i)$.

- Conditionally executed subsystems: such as enabled/triggered subsystems, are subsystems that are conditionally executable when a certain guard condition over certain variables, called control-inputs, holds. Furthermore, the data of the system-block may be reset when the guard condition holds, and the outputs may be reset when the guard condition is violated.

- Logically-executed subsystems: the subsystem is executed one or more times at the current time step when it is enabled by a signal from a control block. An example of logically-executed subsystems are *if Action Subsystem*, *Switch Case Action Subsystem* and *For Each Subsystem*.

3.2 Multi-periodic blocks

From now on, we only consider unconditional blocks and systems.

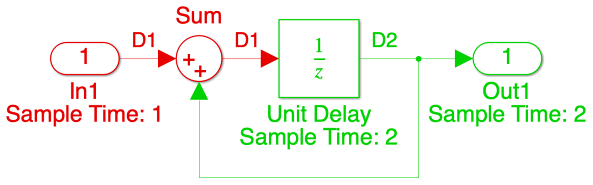


Figure 3: Example of a SIMULINK model.

Figure 3 is a simple example of SIMULINK blocks running on different sample times D1 = (1s, 0s) and D2 = (2s, 0s). The Inport *In1* is running on D1. The *Sum* block is adding two signals running on different periods D1 and D2. The *Unit Delay* is delaying its input signal by one step but updating its output every 2 seconds (sample time D2).

The output of the model is incremented by *In1* every 2 seconds because of the *UnitDelay* block updates its outputs every two seconds and because *Sum* is a stateless block that adds its inputs values at the same time step. The behaviour can be represented as:

t	0	1	2	3	4	5
In1	1	1	1	1	1	1
Out1	0	0	1	1	2	2

Since *UnitDelay* block is running on a different period than the *Sum* block, an implicit data transfer block (*Rate Transition*) is introduced by SIMULINK to ensure all input signals runs on the same period. The user can force SIMULINK to reject models with unspecified data transfers between different rates. In the case of the SIMULINK example in Figure 3, we get the following error when we set "Multi-task (or Single task) rate transition" to "error": *The sample time 1 of 'Sum' at input port 2 is different from the sample time 2 of 'Unit Delay' at output port 1.*

3.3 Explicit clock transition

The model described in Fig. 3 will be rejected by COCOSIM. Instead, the user must specify explicitly the right data transfers block type. In Figure 4 we introduced two *Rate Transition* blocks to ensure *Sum* block and *UnitDelay* block have their inputs running on the same period as the block itself. Thanks to these introduced *Rate Transition* blocks, the boundaries between blocks that run on different period become explicit and makes the reasoning about clocks to be limited to these data transfers blocks.

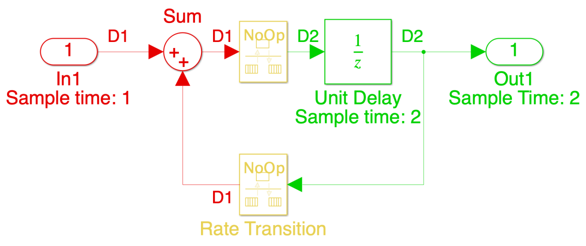


Figure 4: Introducing explicit *Rate Transition* block in the SIMULINK model of Fig. 3.

The *Rate Transition* block (RTB) has two block parameters that control its execution: **Ensure data integrity** and **Ensure deterministic data transfer**. When the first parameter is checked, it ensures data integrity when the block transfers data. A problem of data integrity exists when the input to a block changes during the execution of that block. For instance, a faster block supplies the input to a slower block. In a protected data transfer, the output of the faster block is held until the slower block finishes executing. The second parameter enforces a deterministic data transfer where the timing of the data transfer is completely predictable, as determined by the sample rates of the blocks. The timing of a nondeterministic data transfer depends on the availability of data, the sample rates of the blocks, and the time at which the receiving block begins to execute relative to the driving block.

CoCoSim only allows rate transition blocks that ensure data integrity and determinism. With this restriction, there exist two main types of data transfer:

- ZOH: the Zero-Order-Hold is a deterministic RTB that implements direct communications from fast to slow blocks at harmonic periods. $\exists n \in \mathbb{N}$, $\text{inTs} = \text{outTs}/n$ *and* $\text{inTsOffset} = \text{outTsOffset} = 0$.

- $1/Z$: Acts as a unit delay that implements delayed deterministic communications from slow to fast block at harmonic periods. $\exists n \in \mathbb{N}$, $\text{inTs} = \text{outTs} * n$ *and* $\text{inTsOffset} = \text{outTsOffset} = 0$.

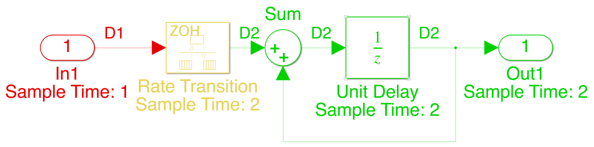


Figure 5: Second clock explicit solution of Fig. 3

The Fig. 5 is an other solution to make clock transition explicit and accepted by CoCoSim. The other three rate transition blocks (Buf, Copy or NoOp, Db-buf) are not allowed by CoCoSim. They depend on the priority of input rate and output rate. Simulink uses a proprietary algorithm using single or multiple buffers to protect data integrity during data transfer. These algorithm are not documented, therefore, CoCoSim supports only Rate Transition blocks with Data Integrity and Determinism.

3.4 Common base clock identification

In Simulink, each period π_i and offset θ_i of a block is defined by a floating point number. In the following, we will encode the semantics of the whole system over a base symbolic clock either using counters (in Lustre) or using rational numbers (in Prelude) to describe the individual local period and offset.

To characterize such common clock, we can express periods and offsets without loss of generality as rational numbers $p/q \in \mathbb{Q}$ where $p \wedge q = 1$. Let us consider n unconditional blocks, each associated to the period and offset $\pi_i, \theta_i \in \mathbb{Q}$. Let $\pi_i^n, \pi_i^d, \theta_i^n, \theta_i^d \in \mathbb{N}$ be such that $\pi_i = \pi_i^n/\pi_i^d$ and $\theta_i = \theta_i^n/\theta_i^d$. We can define the common base clock as $\pi = (\pi^n/\pi^d, 0)$ where $\pi^d = lcm\left(\{\pi_i^d, \theta_i^d\}_i\right)$ and $\pi^n = gcd\left(\{\pi_i \times \pi^d, \theta_i^n \times \pi^d\}_i\right)$. Let us remark that since π^d is defined as the least common multiplier of all denominators of periods and offsets, the terms in the gcd expression of π^n are all integers.

Example 1 *In Figure 5, In1 is defined on* $D1 = (1/1s, 0/1s)$ *and UnitDelay, Sum, Out1 on* $D2 = (2/1s, 0/1s)$. *Thus, the common base clock is* $D = (1s, 0s)$ *since* $1 = lcm(1)$ *and* $1 = gcd(1, 2)$.

We can now express each period π_i and offset θ_i with respect to the base period, as positive integers, respectively $\tilde{\pi_i} = \pi_i \times \pi$ and $\tilde{\theta_i} = \theta_i \times \pi$. This gives a symbolic and denumerable characterization of each period and offset as a multiple of the base clock.

Example 2 *In Figure 5,* $\tilde{D_1}$ *is defined as* $(1, 0)$ *and* $\tilde{D_2}$ *is defined as* $(2, 0)$ *relatively to the base clock* $(1s, 0s)$.

4 Multi-periodic CoCoSim

We present here the extension of CoCoSim to address multi-periodic Simulink models. It uses Lustre as an intermediate formal language for verification since many external model checkers take Lustre model as an input (e.g., Kind2, Zustre, Jkind). It relies on Prelude for code generation.

4.1 Lustre

Lustre code consists of a set of nodes transforming infinite streams of input flows into streams of output flows. A notion of symbolic *abstract* universal clock (also referred as the basic clock) is used to model system progress. In Lustre, a node is defined as a set, i.e. 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 functional and should be well typed: they build values instead of sequencing imperative statements. For instance, 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 corresponds to the unit delay block in

SIMULINK. The arrow operator, known as *follow-by*, allows to build a stream `x0 → e` as the expression `e` while specifying the first value `x0` at time zero. A flow defined as `x0 → pre u` will then correspond in SIMULINK to a Unit Delay over flow `u` with initial value `x0`.

Clocks. 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 (e1 when c) (e2 when not c);
```

t	0	1	2	3
c	true	true	false	true
e1	0	2	4	6
e2	1	3	5	7
e1 when c	0	2	-	6
e2 when not c	-	-	5	-
x	0	2	5	6

Expressions have to be clocked appropriately. The clocking phase of the compiler allows to check the consistency of clocks definitions and their uses.

4.2 Translation in Lustre

The first COCOSIM multi-periodic translation technique [40] produces a pure LUSTRE specification. The first question is to translate the sample time in LUSTRE as sub-sampled clock of the common base clock. The idea is to compute the common base clock as explained in section 3.4 and express each couple (period, offset) relatively to it. For the example in Fig. 5, one solution to encode D1 and D2 could be:

```
D1 = true;
D2 = true → not pre(D2);
```

D1 was exactly equal to the common base clock whereas D2 was twice slower. Then, alternating true and false in the flow D2 will indeed lead to true every two times. However, when the clocks become complex, the intrication of `pre` will become unsustainable. Instead, we will use some counters and will have a similar approach as the one in PRELUDE.

```
D1 = make_clock(1,0);
D2 = make_clock(2,0);
```

Where `make_clock` is a LUSTRE node that generates a Boolean clock that is true at the logical instants $k * \text{period} + \text{offset}$ with $k \in \mathbb{N}$ and false otherwise.

```
node make_clock(period, offset : int)
returns(clk : bool)
var count: int;
let
 count = ((period - offset) -> (pre(count) + 1))
      mod period;
 clk = (count = 0);
tel
```

The second question is how to combine different sample times. The idea is to bring back a flow on the common base clock and to sub-sample. Let us detail the translation for the two rate transition blocks, ZOH and 1/z, supported by COCOSIM. Let (inTs, inTsOffset) (resp. (outTs, outTsOffset)) be the sample time of RTB input port called RTB_U (resp. output port called RTB_Y) relatively to the common base clock. We thus have:

```
C_in = make_clock(inTs, inTsOffset);
C_out = make_clock(outTs, outTsOffset);
```

From Fast to slow: outTs $>$ inTs, ZOH block The communication is direct and the output port RTB_Y simply takes the value of the input port RTB_U on its sample time:

```
RTB_tmp =
  merge C_in  RTB_U ((dft -> pre RTB_tmp) when
        not C_in);
RTB_Y = RTB_tmp when C_out;
```

From slow to fast: outTs $<$ inTs, 1/z block The block behaves as a *Unit Delay*. We first compute the previous value of the input signal, then compute the values in the base clock by keeping the previous values when it is undefined and finally sample the signal to the output clock.

```
RTB_tmp =
 merge C_in (dft -> pre RTB_U)
        ((dft -> pre RTB_tmp) when not C_in);
RTB_Y = RTB_tmp when C_out;
```

Example 3 *The example of Fig. 5 is translated as follows:*

```
In1_on_cc =
  merge D1 In1 (In1_on_cc when not D1);
RateTransition = In1_on_cc when D2;
Sum = RateTransition + UnitDelay;
UnitDelay = 0.0 → pre Sum;
Out1 = UnitDelay;
```

Example 4 *Let us consider Fig. 6 to illustrate the translation of few examples of Rate Transition (RTB) block (ZOH and 1/z).*

In Table. 1 we give few different settings of RTB block and their translation in LUSTRE *and* PRELUDE. *We set the Counter Subsystem with a* SIMULINK *clock of* $(2s, 0s)$ *that is the counter is incremented by 1 every 2 seconds. The common base clock is* $(1s, 0s)$.

t	0	1	2	3	4	5	6	7
Counter: Sample Time [2s, 0]	0	0	1	1	2	2	3	3
RTB: Determinism:ON; Integrity:ON; Sample time [4s,0] => type ZOH								
SIMULINK: Y	0	0	0	0	2	2	2	2
LUSTRE: $C\_4\_0 = make\_clock(4, 0);$ $Counter = Counter\_SS();$ $Y = Counter\ when\ C\_4\_0;$	true 0 0	false - -	false 1 -	false - -	true 2 2	false - -	false 3 -	false - -
PRELUDE: $Y = Counter/\hat{}\,2;$	0	-	-	-	2	-	-	-
RTB: Determinism:ON; Integrity:ON; Sample time [1s,0] => type 1/z								
SIMULINK: Y	0	0	0	0	1	1	2	2
LUSTRE: $C\_2\_0 = make\_clock(2, 0);$ $Counter = Counter\_SS();$ $Y = merge\ C\_2\_0$ $(0.0 \rightarrow pre\ Counter)$ $((0.0 \rightarrow pre\ Y)\ when\ not\ C\_2\_0);$	true 0 0	false - 0	true 1 0	false - 0	true 2 1	false - 1	true 3 2	false - 2
PRELUDE: $Y = (0.0\ fby\ Counter) *\hat{}\ 2;$	0	0	0	0	1	1	2	2

Table 1: Translation to LUSTRE and PRELUDE of different settings of block RTB in Fig. 6.

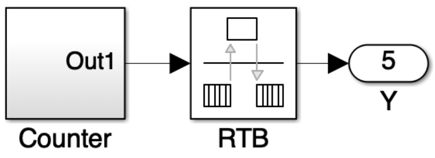


Figure 6: Sampled counter

4.3 Translation in Prelude

PRELUDE [30] is a synchronous language that has been defined to program multi-periodic applications. The language considers *imported nodes* that can be programmed in C or LUSTRE. An example is given below where a node *Sum* has two inputs, one output and a worst case execution time (WCET) of 1 logical time.

```
imported node Sum (v1, v2 :real)
returns (v :real) wcet 1;
```

In the COCOSIM framework, the code generated from SIMULINK will be composed of LUSTRE nodes for mono-periodic (sub-) systems and PRELUDE when several rates are handled. Thus in that case, LUSTRE nodes are assembled together in a PRELUDE program, which details the real-time constraints of the system and the semantics of the communications between the LUSTRE nodes. This assembly is not done directly in LUSTRE because communications relate nodes executing at different periodic rates and PRELUDE is better suited to the specification and efficient compilation.

```
node assembly(In1: real rate(10,0))
 returns (Out1_1 :real)
var
  Sum_1_1 :real;
  UnitDelay_1_1 :real ;
 let
  Sum_1_1 = Sum(0.0 fby UnitDelay_1_1, In1/^ 2);
  UnitDelay_1_1 = UnitDelay(Sum_1_1);
  Out1_1 =  UnitDelay_1_1;
tel
```

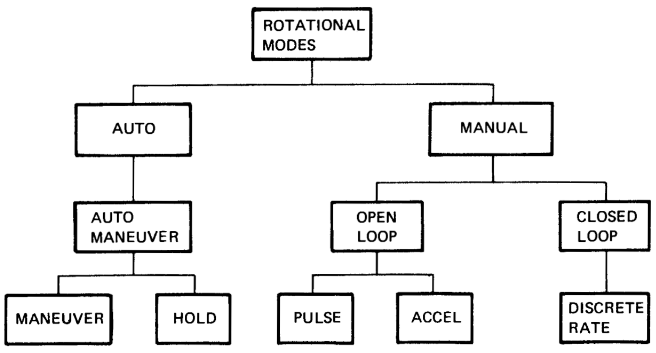
The example corresponds to the Fig. 5. The *assembly* node has one input *In1* with a clock $(10, 0)$ meaning that *In1* has a period of 10 and an offset of 0. It is up to the user to link the logical clock with the physical one. Since *In1* is running at D1=(1s,0s), the logical clock is at $100ms$.

The imported node *Sum* consumes the flows `In1 /^2` and `0.0 fby UnitDelay_1_1`. Nodes are synchronous, thus the two inputs and the output must have the same clock. As the clock of `In1` is $(10, 0)$ by definition, the flow `In1/^2` has $(20, 0)$ as clock. The operator `^2` is a deceleration by 2.

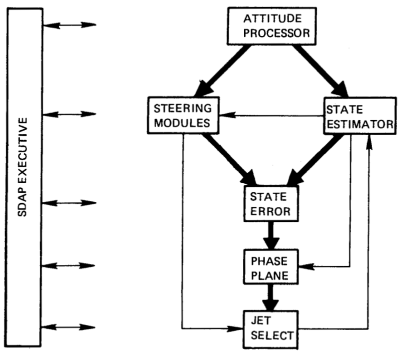
The operator `*^2` is an acceleration that will divide the period by 2. It is used for the 1/z block as shown in table 1.

5 Experiments

In this section, we will present two use cases that will be experimented with COCOSIM.



(a) Modding possibilities



(b) Auto-Maneuver Modules architecture

Figure 7: SDAP Architecture

5.1 Rosace

The first one is Rosace (Research Open-Source Avionics and Control Engineering) [31] a longitudinal flight controller. Although of modest size, this controller is representative of real avionics applications.

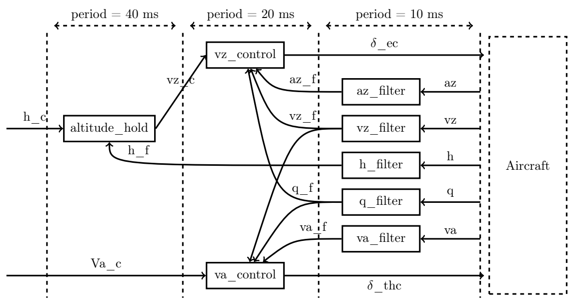


Figure 8: Rosace architecture

Figure 8 depicts the Rosace architecture. The *controller* is composed of 8 functions (depicted by boxes) that run at different periods and exchange data (depicted by arrows). A series of filters (named *X_filter*) consolidate the data measured on the aircraft and two controllers manage the airspeed (*va_ control*) and the vertical speed (*vz_ control* and *altitude_ hold*). Data-dependencies are causal, for instance, the function *vz_filter* produces the variable *vz_f*, which is consumed by functions *vz_ control* and *va_ control.* The controller receives two inputs from the cockpit, which are the orders requested by the pilot on the altitude h_c and on the vertical air speed Va_c. The *Aircraft* receives the orders δ_{ec} and δ_{thc} computed by the controller.

We have translated Rosace in Lustre/Prelude and executed it on many architectures. We have now a Lustre version on which we could make some verification.

5.2 Attitude Control of the Space Shuttle

The second use case is the Attitude and Orbital Control System (AOCS) of the famous Space Shuttle [11]. Fig. 7a presents the different modes in which the Simplified Digital Autopilot (SDAP) of the Space Shuttle can operate. Each of them is modeled as a Simulink component. The most complex one being the *AUTO MANEUVER* sub-mode within the *AUTO* mode. Its architecture is depicted in Fig. 7b: The SDAP executive will command and execute the software logic at a frequency of 12.5 Hz. The Auto Maneuver module is processed at 1.04 Hz, it is called every 12^{th} DAP pass whenever SDAP is in the auto mode.

After collecting the current state from the IMUs (current attitude), the DAP, depending on the switches values entered by the crew, will send appropriate commands to the actuators (jets) to bring about a desired state. The crew can set the behavior they desire via keyboard, and by push-buttons modding discrete, such as the choices between primary/vernier jets and automatic/manual attitude control mode.

We identified and labeled in the Space Shuttle report numerous sentences that could act as requirements. Each of these requirements is then expressed as a Lustre CoCoSpec contract and can then be expressed in the Simulink model. A list of 49 identified requirements can be found in `https://coco-team.github.io/spaceshuttle/requirements.html`. Table 2 represents some of the identified requirements and their verification results.

We have now a Lustre/Prelude version that we will port on multi- and many-core architectures.

Req. ID	Simulink Component	Text
Req_p63_1	Call Jet Select	The two types of thrusters may not be used simultaneously
Req_p19_1	Auto Manual Switch	If the hand controller is deflected in any axis, the SDAP automatically switches to manual mode
Req_p19_5	Auto Manual Maneuver	When the maneuver mode is changed from manual to auto, if the bypass flag is ON, it is set to OFF and the auto maneuver initialization flag is set to ON.
Req_p27_1	Auto Manual Maneuver	Auto Maneuver tests the rotation angle rotation_angle_delta_theta against two numerical criteria. If rotation_angle_delta_theta is larger than $y = SCALARBIAS + 2 * Deadband$, the module places itself in the maneuver mode; if rotation_angle_delta_theta is less than $x = SCALARBIAS + Deadband$, the hold mode results.

Req. ID	Simulink Component	#blocks	SLDV Result	SLDV Time	CoCoSim Result	Total Time	Lustre Gen. Time	Verification Time
Req_p63_1	Call Jet Select	34	Valid	27s	Valid	21s	19s	2s
Req_p19_1	Auto Manual Switch	8	Valid	7s	Valid	12s	10s	2s
Req_p19_5 Req_p27_1	Auto Manual Maneuver	589	Valid	23s	Valid	34s	30	4s

Note that total time reported for CoCoSim is detailed as compilation time to Lustre and actual verification. SLDV is the model-checking tool provided by MathWorks.

Table 2: Selection of requirements and verification results

6 Related Works

Model-based development has been adopted widely in the design of cyber-physical system. Simulink and Scade are the reference and the most successfully used toolkits in the development of such systems. Both Simulink and Scade have code generators and some means of applying formal verification on safety requirements to find bugs early on the development process. Simulink Design Verifier (SLDV) [28] and Scade Design Verifier [34] are both commercial tools and they have their own limitations. For instance, SLDV does not support all Simulink blocks and suffers from state space exmplosion.

The objective of the CoCoSim framework is to provide an open source tool for code generation and an integration hub for others to contribute with different techniques of formal verification and code generation capable of scaling on real-case systems.

In order to perform any type of analysis on the Simulink/Stateflow models, a formal specification needs to be defined. Since the semantic of Simulink/Stateflow [2] is informal, many efforts have been made to define a formal semantic of a subset of Simulink/Stateflow in many formal languages for code generation as well as for formal verification. In its PhD, Cédric Klikpo [24] has proposed a clear explanation of rate transition blocks.

Discrete subset of Simulink has been translated to various input languages of formal verification such as NuSMV model [29], Lustre language [40], SDF (Synchronous Data Flow) [24] or hybrid automata [37]. All of the previous works are either closed source or not maintained or does not work with new Simulink version as their transla-

[2]https://www.mathworks.com/help/pdf_doc/simulink/slref.pdf

tor parses the model in its textual format. We addressed those two challenges: it is open source and is based on SIMULINK API to get and compute all model information needed for translation, therefore any new SIMULINK release does not affect badly the translator.

Expressing multi-periodic systems with synchronous languages has been proposed in previous works. In [6], SCADE was used to describe multi-periodic systems with harmonic period. In [15], we made a state of the art of the question for the works before 2008. Some approach, similar to PRELUDE, was proposed in [1]. More recently, Didier et al. [14] have defined an extension of HEPTAGON and Hili and al. have extended ForeC [23], a C-based synchronous programming language.

Regarding code generation, there exist other works on implementation of synchronous multi-periodic systems. For code generation, [25] presents an approach to formally define the synchronous semantic of multi-periodic SIMULINK system using Synchronous Dataflow Graph. In their work, they focus only on the semantic-preserving implementation of the interactions between tasks. [35] presents another translation of a discrete-time SIMULINK into the synchronous subset of the BIP language to compare runtime performances with SIMULINK code generation tool.

7 Conclusion

We presented an overview of COCOSIM framework, a modular framework to support the analysis and the code generation of discrete Simulink models. We focused on two COCOSIM backends addressing the analysis and code generation of multi-periodic SIMULINK models. The first backend is addressing the V&V of multi-periodic systems by encoding them into a pure LUSTRE models to properly analyze the full system. The second backend is dedicated to code generation based on PRELUDE. We presented two use cases, the first is demonstrating the capabilities of PRELUDE, whereas, the second use case we demonstrated the V&V capabilities of COCOSIM.

References

[1] Mouaiad Alras, Paul Caspi, Alain Girault, and Pascal Raymond. "Model-Based Design of Embedded Control Systems by Means of a Synchronous Intermediate Model". In: *International Conference on Embedded Software and Systems (ICESS'09)*. Hangzhou, China, May 2009.

[2] Gérard Berry and Georges Gonthier. "The Esterel synchronous programming language: design, semantics, implementation". In: *Science of Computer Programming* 19.2 (1992), pp. 87–152. ISSN: 0167-6423.

[3] Dariusz Biernacki, Jean-Louis Colaço, Grégoire Hamon, and Marc Pouzet. "Clock-directed modular code generation for synchronous data-flow languages". In: *LCTES'08*. 2008.

[4] Matteo Bordin, Tonu Naks, Marc Pantel, and Andres Toom. "Compiling heterogeneous models: motivations and challenges". In: *Proceedings of the 6th International Congress Embedded Real Time Software (ERTS'12)*. 2012.

[5] Aaron R. Bradley. "IC3 and beyond: Incremental, Inductive Verification". In: *CAV'12*. 2012.

[6] Jean-Louis Camus, Olivier Graff, and Sébastien Poussard. "A verifiable architecture for multitask, multi-rate synchronous software". In: *4th Embedded Real-Time Software Congress (ERTS'08)*. 2008.

[7] Paul Caspi, Adrian Curic, Aude Maignan, Christos Sofronis, and Stavros Tripakis. "Translating Discrete-Time Simulink to Lustre". In: *Third International Conference on Embedded Software EMSOFT*. 2003, pp. 84–99.

[8] Paul Caspi, Daniel Pilaud, Nicolas Halbwachs, and John Plaice. "Lustre: A Declarative Language for Programming Synchronous Systems". In: *POPL'87*. 1987, pp. 178–188.

[9] Adrien Champion, Arie Gurfinkel, Temesghen Kahsai, and Cesare Tinelli. "CoCoSpec: A Mode-Aware Contract Language for Reactive Systems". In: *SEFM'16*. 2016, pp. 347–366. ISBN: 978-3-319-41591-8.

[10] Adrien Champion, Alain Mebsout, Christoph Sticksel, and Cesare Tinelli. "The Kind 2 Model Checker". In: *Computer Aided Verification - 28th International Conference, CAV 2016, Toronto, ON, Canada, July 17-23, 2016, Proceedings, Part II*. 2016, pp. 510–517.

[11] Raphaël Cohen, Pierre-Loïc Garoche, and Hamza Bourbouh. *Space Shuttle Attitude and Orbital Control System*. `https://github.com/coco-team/spaceshuttle`.

[12] Jean-Louis Colaço, Bruno Pagano, and Marc Pouzet. "SCADE 6: A formal language for embedded critical software development (invited paper)". In: *11th International Symposium on Theoretical Aspects of Software Engineering, TASE 2017, Sophia Antipolis, France, September 13-15, 2017*. 2017, pp. 1–11.

[13] Guillaume Davy, Christophe Garion, Pierre-Loïc Garoche, Pierre Roux, and Xavier Thirioux. "Ensuring functional correctness of cyber-physical system controllers: from model to code analyses". In: *Forum on Specification and Design Languages, Special session on Logic and Mathematics Behind Design Automation, FDL'18, TU Munich 10.9-12.9.2018*. Sept. 2018.

[14] Keryan Didier, Dumitru Potop-Butucaru, Guillaume Iooss, Albert Cohen, Jean Souyris, Philippe Baufreton, and Amaury Graillat. "Correct-by-Construction Parallelization of Hard Real-Time Avionics Applications on Off-the-Shelf Predictable Hardware". In: *TACO* 16.3 (2019), 24:1–24:27.

[15] Julien Forget, Frédéric Boniol, David Lesens, Claire Pagetti, and Marc Pouzet. "Programming Languages For Hard Real-Time Embedded Systems". In: *Proceedings of the 4th Conference on Embedded Real Time Software and Systems and Software (ERTS'08)*. 2008.

[16] Andrew Gacek, John Backes, Mike Whalen, Lucas G. Wagner, and Elaheh Ghassabani. "The JKind Model Checker". In: *CoRR* abs/1712.01222 (2017). arXiv: `1712.01222`.

[17] Pierre-Loïc Garoche, Falk Howar, Temesghen Kahsai, and Xavier Thirioux. "Testing-Based Compiler Validation for Synchronous Languages". In: *NASA Formal Methods*. Ed. by Julia M. Badger and Kristin Yvonne Rozier. Cham: Springer International Publishing, 2014, pp. 246–251. ISBN: 978-3-319-06200-6.

[18] Pierre-Loïc Garoche, Falk Howar, Temesghen Kahsai, and Xavier Thirioux. "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.

[19] Pierre-Loïc Garoche, Temesghen Kahsai, and Xavier Thirioux. "Hierarchical State Machines as Modular Horn Clauses". In: *Proceedings 3rd Workshop on Horn Clauses for Verification and Synthesis, HCVS@ETAPS 2016, Eindhoven, The Netherlands, 3rd April 2016*. 2016, pp. 15–28.

[20] Pierre-Loïc Garoche, Temesghen Kahsai, and Xavier Thirioux. *LustreC*. `https://github.com/coco-team/lustrec`.

[21] Pierre-Loïc Garoche, Temesghen Kahsai, and Xavier Thirioux. *Zustre*. `https://github.com/coco-team/zustre`.

[22] Pierre-Loıc Garoche, Arie Gurfinkel, and Temesghen Kahsai. "Synthesizing Modular Invariants for Synchronous Code". In: *HCVS'14*. 2014.

[23] Alain Girault, Nicolas Hili, Eric Jenn, and Eugene Yip. "A Multi-Rate Precision Timed Programming Language for Multi-Cores". In: *2019 Forum for Specification and Design Languages, FDL 2019, Southampton, United Kingdom, September 2-4, 2019*. 2019, pp. 1–8.

[24] Enagnon Cédric Klikpo. "Méthode de conception de systèmes temps réels embarqués multi-coeurs en milieu automobile. (Methodology of designing embedded real-time multi-core systems in automotive)". PhD thesis. Sorbonne University, Paris, France, 2018.

[25] Enagnon Cédric Klikpo, Jad Khatib, and Alix Munier Kordon. "Modeling Multi-Periodic Simulink Systems by Synchronous Dataflow Graphs". In: *2016 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS), Vienna, Austria, April 11-14, 2016*. 2016, pp. 209–218.

[26] Anvesh Komuravelli, Arie Gurfinkel, and Sagar Chaki. "SMT-based model checking for recursive programs". In: *Formal Methods in System Design* 48.3 (2016), pp. 175–205. ISSN: 1572-8102.

[27] Anvesh Komuravelli, Arie Gurfinkel, Sagar Chaki, and Edmund M. Clarke. "Automatic Abstraction in SMT-Based Unbounded Software Model Checking". In: *Computer Aided Verification - 25th International Conference, CAV 2013, Saint Petersburg, Russia, July 13-19, 2013. Proceedings*. 2013, pp. 846–862.

[28] The MathWorks. *Simulink Design Verifier (SLDV)*. `https://www.mathworks.com/products/sldesignverifier.html`.

[29] B. Meenakshi, Abhishek Bhatnagar, and Sudeepa Roy. "Tool for Translating Simulink Models into Input Language of a Model Checker". In: *Formal Methods and Software Engineering*. Ed. by Zhiming Liu and Jifeng He. Berlin, Heidelberg: Springer Berlin Heidelberg, 2006, pp. 606–620. ISBN: 978-3-540-47462-3.

[30] Claire Pagetti, Julien Forget, Frédéric Boniol, Mikel Cordovilla, and David Lesens. "Multi-task Implementation of Multi-periodic Synchronous Programs". In: *Discrete Event Dynamic Systems* 21 (Sept. 2011), pp. 307–338.

[31] Claire Pagetti, David Saussié, Romain Gratia, Eric Noulard, and Pierre Siron. "The ROSACE case study: From Simulink specification to multi/many-core execution". In: *20th IEEE Real-Time and Embedded Technology and Applications Symposium, RTAS*. 2014, pp. 309–318.

[32] Wolfgang Puffitsch, Eric Noulard, and Claire Pagetti. "Off-line Mapping of Multi-rate Dependent Task Sets to Many-core Platforms". In: *Real-Time Systems* 51.5 (Sept. 2015), pp. 526–565. ISSN: 0922-6443.

[33] Ana-Elena Rugina, David Thomas, Xavier Olive, and G. Veran. "Gene-Auto: Automatic Software Code Generation for Real-Time Embedded Systems". In: *Proceedings of DASIA 2008 Data Systems In Aerospace*. 2008.

[34] *SCADE Suite Design Verifier / Esterel Technologies*. `https://www.ansys.com/products/embedded-software`.

[35] Vassiliki Sfyrla, Georgios Tsiligiannis, Iris Safaka, Marius Bozga, and Joseph Sifakis. "Compositional Translation of Simulink Models into Synchronous BIP". In: *IEEE Fifth International Symposium on Industrial Embedded Systems, SIES 2010, University of Trento, Italy, July 7-9, 2010*. 2010, pp. 217–220.

[36] Mary Sheeran, Satnam Singh, and Gunnar Stålmarck. "Checking Safety Properties Using Induction and a SAT-Solver". In: *FMCAD'00*. 2000, pp. 127–144. ISBN: 978-3-540-40922-9.

[37] B. I. Silva and B. H. Krogh. "Formal verification of hybrid systems using CheckMate: a case study". In: *Proceedings of the 2000 American Control Conference. ACC (IEEE Cat. No.00CH36334)*. Vol. 3. June 2000, 1679–1683 vol.3.

[38] "Synchronous programming with events and relations: the SIGNAL language and its semantics". In: *Science of Computer Programming* 16.2 (1991), pp. 103–149. ISSN: 0167-6423.

[39] Andres Toom, Tonu Naks, Marc Pantel, M Gandriau, and Indrawati. "Gene-Auto: an Automatic Code Generator for a safe subset of Simulink/Stateflow and Scicos". In: *Proceedings of the 4th International Congress Embedded Real Time Software (ERTS'08)*. 2008.

[40] Stavros Tripakis, Christos Sofronis, Paul Caspi, and Adrian Curic. "Translating discrete-time simulink to lustre". In: *ACM Trans. Embedded Comput. Syst.* 4.4 (2005), pp. 779–818.