From Lustre to Simulink: reverse compilation for verifying Embedded Systems Applications

Hamza Bourbouh, Pierre-Loïc Garoche, Christophe Garion, Xavier Thirioux

From Lustre to Simulink: Reverse Compilation for Embedded Systems Applications

HAMZA BOURBOUH, KBR/NASA Ames Research Center
PIERRE-LOÏC GAROCHE, KBR/NASA Ames Research Center/ENAC, Université de Toulouse
CHRISTOPHE GARION and XAVIER THIRIOUX, ISAE-SUPAERO, Université de Toulouse

Model-based design is now unavoidable when building embedded systems and, more specifically, controllers. Among the available model languages, the synchronous dataflow paradigm, as implemented in languages such as MATLAB Simulink or ANSYS SCADE, has become predominant in critical embedded system industries. Both of these frameworks are used to design the controller itself but also provide code generation means, enabling faster deployment to target and easier V&V activities performed earlier in the design process, at the model level. Synchronous models also ease the definition of formal specification through the use of synchronous observers, attaching requirements to the model in the very same language, mastered by engineers and tooled with simulation means or code generation.

However, few works address the automatic synthesis of MATLAB Simulink annotations from lower-level models or code. This article presents a compilation process from Lustre models to genuine MATLAB Simulink, without the need to rely on external C functions or MATLAB functions. This translation is based on the modular compilation of Lustre to imperative code and preserves the hierarchy of the input Lustre model within the generated Simulink one. We implemented the approach and used it to validate a compilation toolchain, mapping Simulink to Lustre and then C, thanks to equivalence testing and checking. This backward compilation from Lustre to Simulink also provides the ability to produce automatically Simulink components modeling specification, proof arguments, or test cases coverage criteria.

CCS Concepts: • **Computer systems organization → Real-time system specification**; **Embedded software**; • **Software and its engineering → Model-driven software engineering**; **Data flow languages;**

Additional Key Words and Phrases: Model-based design, formal verification, translation validation, equivalence checking, Simulink

This work was partially supported by grant ANR JCJC FEANICSES ANR-17-CE25-0018.
Authors' addresses: H. Bourbouh, KBR/NASA Ames Research Center, Mountain View, CA; email: hamza.bourbouh@nasa.gov; P.-L. Garoche, KBR/NASA Ames Research Center/ENAC, Université de Toulouse, Toulouse, France; email: Pierre-Loic.Garoche@enac.fr; C. Garion and X. Thirioux, ISAE-SUPAERO, Université de Toulouse, Toulouse, France; emails: {christophe.garion, xavier.thirioux}@isae-supaero.fr.

https://doi.org/10.1145/3461668

1 INTRODUCTION

When developing safety-critical software and systems like aircraft controllers, system designers and engineers are now using **Model-Based System Engineering (MBSE)**. MBSE provides early stage prototyping and often tools enabling simulation or even code generation. Among the standard modeling languages used, we shall mention MATLAB Simulink and ANSYS SCADE as industry-grade model languages used in multiple contexts from aerospace systems to healthcare devices or nuclear plants.

Let us focus on safety-critical controller software and systems. In most cases, such systems are designed and implemented as the composition of several reactive components, each performing a specific and relatively simple function. In the aerospace domain, the certification regulation DO178C [1] specifies the different steps of software development and emphasizes the need to *specify requirements* and to *verify the validity* of intermediate models or code with respect to their requirements. A first question that naturally arises is the following: in a MBSE context, how can system designers specify these requirements and verify the validity of their models?

In addition to models, a leading methodology to develop component-based software is contract-based design. In this paradigm, each component is associated with a contract specifying its input-output behavior in terms of guarantees provided by the component when its environment satisfies certain given assumptions. These assume/guarantee pairs can thus be used to specify *requirements* at the component level. This approach was first proposed by Hoare [27] to specify axiomatic semantics of imperative programs; however, it was later lifted to reactive systems through the notion of *synchronous observers* [12, 16, 25, 26, 35]. When contracts are specified formally for individual components, they can facilitate several development activities, such as compositional reasoning during static analysis, step-wise refinement, systematic component reuse, and component-level and integration-level test case generation.

At the model level, writing requirements with synchronous observers can usually be performed in the same language as the model, easing its deployment and the adoption of the approach by engineers. These requirements may be verified on the model by simulation or other techniques, such as model checking. Notice that contracts or synchronous observers attached to model components can also be used to specify additional knowledge such as invariants computed by a first analysis.

At a research level, the topics of certified compilation, test-case generation, or formal analysis of dataflow languages are very active. Yet the integration of research ideas from academia into commercial frameworks such as Simulink is quite difficult. The main reason is the lack of formally publicly available semantics of the Simulink language. It is more common to develop methods and tools on well-defined and publicly available languages such as Lustre [7, 8, 36]. For instance, approaches such as FRET [23] or Dassault Systèmes STIMULUS[1] ease the formalization of requirements but can hardly be directly linked to Simulink models to provide genuine Simulink components representing the specification. However, the FRET tool can generate these requirements in the CoCoSpec specification language [12], an extension of the Lustre language to support assume-guarantees contracts. Formal analysis of Simulink models is also addressed by first providing a formal semantic of the model allowing either executable embedded code or a model for analysis by formal tools. CoCoSim [6] supports the analysis of a discrete subset of Simulink/Stateflow models by generating an equivalent Lustre model for which contracts are expressed using the CoCoSpec specification language and verified using model-checking techniques with tools such as Kind2 [12, 28].

Although translating Simulink models into formal languages such as Lustre allows to formally analyze such models, bringing back analysis artifacts generated from Lustre in the Simulink model

[1] https://www.3ds.com/products-services/catia/products/stimulus/.

is currently not done and needs a translation of Lustre code to Simulink. These artifacts can include invariants generated from SMT solvers that can be used to infer contracts for subcomponents, Lustre annotations such as test-case coverage conditions, and auto-generated CoCoSpec contracts from formalized requirements using FRET for instance. Connecting Lustre artifacts back to the Simulink model makes it simpler for engineers using Simulink to adopt the use of formal tools, and no knowledge of formal languages such as Lustre is required.

Contributions. The main contribution of this work is a compilation process from Lustre models to Simulink models allowing closer connection to formal analysis tools of Simulink models (e.g., CoCoSim). More specifically:

- We based this new compilation on the existing modular compilation of synchronous dataflow languages [5, 22], providing a sound compilation scheme, preserving the semantics.
- Connected to the FRET [23] tool, we automatically produce Simulink contracts from English language specifications, generating an intermediate Lustre encoding.
- The compilation proposed is also used to provide runnable evidence at the model level, soundly translating analysis results from a model checker.
- The Simulink model can be annotated with test-case coverage conditions.
- The approach has been applied to large benchmarks, showing the applicability on a large set of components and requirements.
- Last, all of the presented approaches are implemented and available in the open source CoCoSim toolbox [6].

Related works. Programming languages could be fitted with specification languages, such as ACSL [3] for C or SPARK [2]. In the case of synchronous languages and models, multiple works [14, 31, 39] advocate for the use of component-attached requirements. Notice that the actual definition of such contracts or reasoning on them is still a challenge. Formalized contracts can be used for a large set of applications: test oracles, test synthesis, reactive synthesis [29, 30], compositional reasoning, or validation of individual contracts. CoCoSpec [12, 13] is a specification language for Lustre [24] and has been extended to Simulink models.

Regarding the compilation of Lustre models as Simulink components, an interesting approach is the **Assume Guarantee Reasoning Environment (AGREE)** framework [31]. In AGREE, AADL components are associated to Lustre contracts. Validation of the composition of components is performed thanks to the combination of these contracts by Lustre-based tools such as JKind [19]. In the case of AADL components defined by Simulink models, the Lustre contract is also checked against the Simulink model using Simulink Design Verifier. The method proposed by Liu et al. [31] provides an encoding of the Lustre contract as a piece of MATLAB code—that is, an imperative program embedded within a MATLAB S-function Simulink component. This approach enables the analysis of the model with the Simulink Design Verifier but is restricted to MATLAB-based tools since few external approaches are capable of analyzing or compiling MATLAB code.

Outline. The article is structured as follows. Section 2 presents the CoCoSim framework, as well as the Lustre language and the associated CocoSpec contracts used within the framework as intermediate models. Section 3 develops the compilation process from Lustre nodes to Simulink subsystems. Section 4 presents experimental results, and Section 5 presents various uses such as synthesis of contracts as runnable evidence or test oracles.

2 BACKGROUND

In this section, we present CoCoSim, a Simulink analysis framework, as well as the syntax and semantics of Lustre [24] and associated contracts.

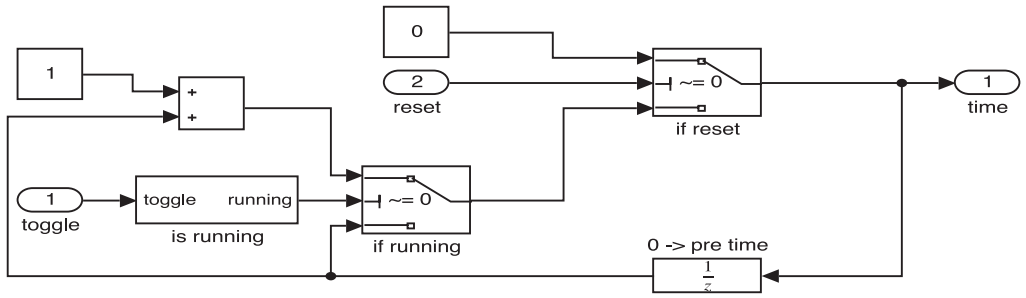


Fig. 1. Stopwatch example in Simulink.

2.1 Simulink

Simulink [32], developed by MathWorks, is a graphical programming language for modeling dynamical systems, including discrete time ones (i.e., synchronous dataflow systems). Simulink has gained popularity in critical embedded systems development. It supports the design and simulation of complex systems before automatically generating embedded C code. A Simulink model consists of a set of blocks connected by signals that can be organized as hierarchical models. Figure 1 illustrates a stopwatch example that measures the amount of time elapsed between its activation and deactivation. The stopwatch is controlled by two external signals: a `toggle` signal to toggle the activation of the stopwatch and a `reset` signal to reset the counter.

Simulink has a rich library of blocks and also supports both continuous and discrete solvers in its simulation engine. Blocks can run on different sample times (multi-periodic) or on one global sample time (mono-periodic). However, Simulink is lacking a formally published reference semantics for its library of blocks, which makes formal analysis of such models difficult.

2.2 CoCoSim

CoCoSim [6] is an open source toolbox for verifying Simulink/Stateflow models. It is integrated within MATLAB as a toolbox and provides easy access to a set of tools. Specification-wise, CoCoSim allows attaching contracts, such as synchronous observers, to Simulink subsystems. Contracts are dedicated subsystems relying on Boolean dataflows to denote elements of the contract (e.g., assumptions and guarantees). CoCoSim is structured as a compiler and follows a simple schema initially developed for the discrete subset of Simulink [38]. Using the MATLAB API, it iterates over blocks and produces their equivalent version as Lustre nodes.

Figure 2 presents the CoCoSim architecture. In practice, a first preprocessing phase performs model-to-model transformation and replaces some blocks by equivalent but simpler versions. The second phase consists of compiling this simpler version of the Simulink model to Lustre (cf. Section 2.3). This compilation is modular and produces a Lustre node for each Simulink subsystem. Once the Lustre model is obtained, it can be either compiled to C code with the LustreC compiler [16] or submitted to Lustre model checkers such as Kind2 [12, 28] or Zustre [21].

CoCoSim carefully addresses traceability issues by manipulating a model along its processing chain. This enables the expression of feedback from model checkers to Simulink models. For example, a counterexample can be replayed at the Simulink level using its simulation engine.

2.3 Lustre

Lustre [11] is a synchronous language for modeling systems of synchronous reactive components.

A Lustre program L is a collection of nodes $N_0, N_1, \ldots, N_m$. The nodes satisfy the grammar described in Figure 3 in which td denotes type constructors, including enumerated types, and

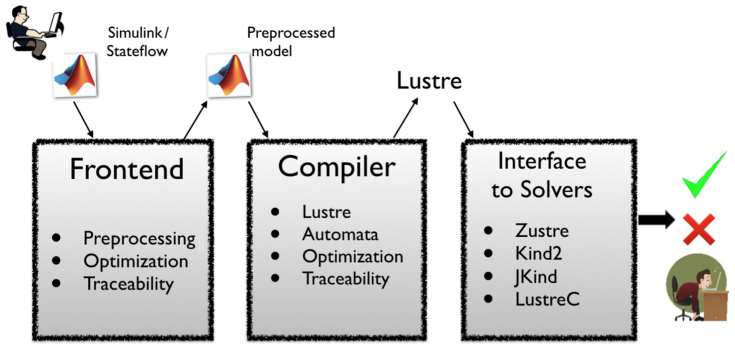


Fig. 2. The overall architecture of CoCoSim.

value v denotes either constants of enumerated types C or primitive constants such as integers i or reals r. Each node is declared by the grammar construct d of Figure 3 and is represented by the following tuple:

$$N_i = (\mathcal{I}_i^N, O_i^N, \mathcal{L}_i, Eqs_i),$$

where $\mathcal{I}_i^N, O_i^N$, and $\mathcal{L}_i$ are sets of typed input, output, and local variables. Eqs_i represents the set of stream definitions defined as

$$Eqs_i = \left\{(v_i^j)_{1\leq j\leq nb_i} = expr_i\right\}_{i\in\{0,\ldots,|Eqs_i|-1\}},$$

where $nb_i \in \mathbb{N}^*$ denotes the number of output variables defined by the expression $expr_i$, $v_i^j \in O_i^N \cup \mathcal{L}_i$ and $expr_i$ is an expression where $Vars(expr_i) \subseteq O_i^N \cup \mathcal{I}_i^N \cup \mathcal{L}_i$. $Vars(expr_i)$ is the set of variables in $expr_i$, and expressions $expr_i$ are arbitrary Lustre expression, as presented in Figure 3 by constructor e, including node calls $N_j(u_1, \ldots, u_n)$.

Lustre code consists of a set of nodes transforming streams of input values into streams of output values. Lustre models are synchronous in the sense that the processing time of each component is neglected and communication is assumed to be instantaneous [4]. A notion of symbolic "abstract" universal clock is used to model system progress.

Let us illustrate Lustre syntax on a possible model for the stopwatch example. The code is presented in Listing 1.

Listing 1. The stopwatch example with clocks.

```
1  node count (tick : bool) returns (time : int);
2  let
3    time = 0 -> pre time + 1;
4  tel
5
6  node stopwatch (tick, toggle, reset : bool) returns (time : int);
7  var running : bool clock;
8  let
9    running = ((false -> pre running) <> toggle) or reset;
10   time = merge running (true  -> count(tick when running) every reset)
11                        (false -> (0 -> pre time) when not running);
12 tel
```

$$
\begin{array}{rcl}
td & ::= & \textbf{type}\ enum\_ident\ = enum\ \{\ C_1, \ldots, C_n\} \\
bt & ::= & real \mid bool \mid int \mid enum\_ident \\
d & ::= & \textbf{node}\ f\ (p)\ \textbf{returns}\ (p);\ \textbf{vars}\ p\ \textbf{let}\ D\ \textbf{tel} \\
p & ::= & x : bt; ...; x : bt \\
D & ::= & pat = e; D \mid pat = e; \\
pat & ::= & x \mid (pat, ..., pat) \\
e & ::= & v \mid x \mid (e, ..., e) \mid e \rightarrow e \mid op(e, ..., e) \mid \textbf{pre}\ e \\
 & & \mid f(e, ..., e) \mid f(e, ..., e)\ \textbf{every}\ e \\
 & & \mid e\ \textbf{when}\ C(x) \mid \textbf{merge}\ x\ (C \rightarrow e)...(C \rightarrow e) \\
 & & \mid \textbf{if}\ e\ \textbf{then}\ e\ \textbf{else}\ e \\
v & ::= & C :: enum\_ident \mid i :: int \mid r :: real \mid true :: bool \mid false :: bool
\end{array}
$$

Fig. 3. A subset of Lustre syntax.

Lines 1 through 4 of Listing 1 define a `count` node returning an integer stream representing the sequence of natural numbers. Primitive types like **`bool`**, **`int`**, or **`real`** are available. Note that, in general, a node may declare several output streams.

Line 6 declares a node named `stopwatch` that takes three Boolean streams as parameters, namely `tick`, `toggle`, and `reset`, and declares a single integer stream as output, namely `time`.

In Lustre, a node is defined by a set of *stream equations* with possible local variables denoting internal flows. Stream equations are defined between the **`let`** and **`tel`** keywords. For instance, line 7 declares `running` as a local Boolean flow.

When defining equations, regular arithmetic and comparison operators are lifted to streams and are evaluated at each timestep. For instance, line 9 of Listing 1 defines stream `running` as a disjunction of the `reset` input stream and the result of comparing two Boolean streams: `false -> pre running`, a Lustre expression, and `toggle`, one of the input streams of the node. The temporal operator **`pre`**, for *previous*, enables a limited form of memory, allowing to read the value of a stream at the previous instant. The arrow operator allows to build a stream `c -> e` as the expression `e` while specifying the first value `c`. Therefore, the expression `false -> pre running` denotes a Boolean stream whose first value is `false` and whose next values are the previous values of the `running` stream.

A node that relies on these constructs is considered as *stateful*; its internal state is defined by the values of the memories. Without these temporal operators, nodes act as mathematical functions.

Another specific construct of Lustre 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 for such clock values. For instance, let us consider the expression `e when c` where `c` is a Boolean clock, then the expression `e when c` is not defined when variable `c` is false.

Let us now explain the expressions in lines 10 and 11:

- `count(tick when running)` is a call to node `count` with argument `tick` clocked on the clock `running`. Therefore, there is no value for this expression when `running` is false. Notice that the `tick` parameter of the node `count` is unused in the definition: it is only used for clocking.
- `count(tick when running) every reset` is the previous call to node `count` completed with a reset expression `every reset`. This specifies that if Boolean `reset` is true, then the call to `count` reinitializes the node to its initial state and therefore the `time` stream to `0`. The local stream `running` is true whenever `reset` is true, and therefore the node `count` is always executed when `reset` is true and the arrow operator will reset to its initial value 0.

Table 1. Evolution of Expressions and Variables Using a Clock in the Stopwatch Example

`toggle`	False	True	False	False	False	False	True	False	False
`running`	False	True	True	True	True	True	False	False	False
`count(tick when running)`		1	2	3	4	5			
`(0 -> pre(time)) when not running`	0						5	5	5
`time`	0	1	2	3	4	5	5	5	5

- `(0 -> pre time) when not running` is an integer stream. It starts with value `0` and then uses the previous value of `time`. This expression is clocked on the negation of the `running` clock. Notice that it is defined if and only if the previous expression is not defined.
- Lines 10 and 11 define a **`merge`** expression. It is used to create a flow clocked on a particular clock using expressions clocked on subclocks of this particular clock. Here, this means the following:
  - `time` will be clocked on the base clock.
  - When `running` is true, the expression `count(tick when running) every reset` is used to define `time`. This expression must be clocked on `running`, which is trivially the case here, but we may have used an external expression clocked on `running` for instance.
  - When `running` is false, the expression `(0 -> pre time) when not running` is used to define `time` and is also trivially clocked on the negation of `running`.

Table 1 presents an example of an evaluation of several streams from the `stopwatch` node: first `running` is `false`, then becomes `true` when the `toggle` is `true` and becomes `false` when the `toggle` is `true` again (simulating the operator pressing the toggle button of the stopwatch). The `reset` parameter is considered always false in this example.

Finally, expressions associated with each clock case have to be clocked appropriately, and the clocking phase of the compiler allows checking the consistency of clock definitions and uses, as would do a typing compilation phase.

Nodes and calls form a hierarchy of nodes comparable to the notion of *subsystems* in Simulink. Type and clock inferences 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, consider the following equations `x = f(y); y = g(x);`. These two equations create a causality problem and produce an *algebraic loop error*. However, notice that the same definitions with either `f(pre y)` or `g(pre x)` would be typable and accepted by the compiler. A more complete definition including additional constructs such as automata or clocks based on enumerated types [15], as used in our framework, can be found in the work of Garoche et al. [21]. Our approach handles those constructs, but their definition is not required to present the contribution.

2.4 Synchronous Observers

The synchronous observer acts as a description of an axiomatic semantics for a synchronous model. The observer is defined in the same language as the model itself and corresponds to a set of Boolean streams. If the property is valid, the output flow encoding the property should remain `true` during the execution of the program.

In CoCoSim, this is performed with a specific specification subsystem and implemented at Lustre level by expressing these Boolean flows as comments in the code. Different syntaxes enable such

annotations, and CoCoSpec contracts [12] are one of them in which annotations specify assume-guarantee contracts, a kind of Hoare triples extended to the synchronous setting [16].

For instance, a synchronous observer on the previous stopwatch example would be the assertion that the stream `time` always has a non-negative value. Listing 2 illustrates a simple CoCoSpec contract using the stopwatch node introduced earlier and specifying that the stream `time` always has a non-negative value given the assumption that `toggle` and `reset` cannot be pressed at the same time.

Listing 2. The stopwatch CoCoSpec contract example.

```
1 contract stopwatchSpec (toggle, reset : bool) returns (time : int);
2 let
3   -- we can assume that the two buttons are never pressed together
4   assume not (toggle and reset);
5   -- the elapsed time is always non-negative
6   guarantee time >= 0;
7 tel
```

Our objective is then to be able, starting from a Lustre implementation and specification of a system, to generate a Simulink design with the same behavior as the Lustre one with the Simulink synchronous observers corresponding to the specification of the Lustre nodes.

3 COMPILING LUSTRE NODES AS SIMULINK BLOCKS

The compilation of Lustre nodes into Simulink subsystems is performed in two steps:

- The first step is to produce a simplified version of the input Lustre model, preserving the hierarchical structure of nodes. This is done using a dedicated backend we implemented in LustreC [16], an open source Lustre compiler.
- The second step is to submit the previously produced description to a dedicated backend of CoCoSim that creates Simulink objects of the associated hierarchy of components and connects the corresponding ports in the Simulink model.

Translating discrete Simulink to Lustre is algorithmically simple. Each connection between blocks is associated with a fresh variable, and each block is mapped to a function call, a basic operator, or another node in the case of Simulink subsystems. The challenge is more on defining the semantic of each Simulink block. For instance, Simulink unit delays modeling memories are mapped to expressions `i -> pre e` where `i` is the initial value specified in the unit delay block and `e` is the input signal of the unit delay.

Going backward (i.e., translating Lustre to Simulink) is more challenging when dealing with a complex Lustre AST and requires a simplified version of Lustre equations. For instance, although a basic expression `pre e` could be associated with unit delay, its presence as an argument in a complex expression or a node call is more difficult to tackle. Our idea is to use a Lustre compiler to simplify expressions and produce appropriate constructs. We use LustreC, a Lustre compiler implementing the synchronous dataflow languages hierarchical compilation scheme [5, 9]. The LustreC compiler is implemented as a sequence of transformations and could eventually produce an imperative version of the Lustre input model.

3.1 From Lustre to Normalized Lustre

LustreC compilation is essentially structured in three main phases. LustreC takes as input Lustre models composed of "classic" dataflow nodes, mixed with hierarchical state machines [7, 15, 21]. The first phase of the compiler, therefore, amounts to producing pure dataflow Lustre by

$$
\begin{array}{rcl}
td & ::= & \textbf{type}\ enum\_ident\ = enum\ \{\ C_1, \ldots, C_n\} \\
bt & ::= & real \mid bool \mid int \mid enum\_ident \\
d & ::= & \textbf{node}\ f\ (p)\ \textbf{returns}\ (p);\ \textbf{vars}\ p\ \textbf{let}\ D\ \textbf{tel} \\
p & ::= & x : bt; ...; x : bt \\
\tilde{D} & ::= & pat = \tilde{e}; D \mid pat = \tilde{e}; \\
pat & ::= & x \mid (pat, ..., pat) \\
l & ::= & v \mid x \\
\tilde{e} & ::= & l \\
 & & \mid true \rightarrow false \\
 & & \mid op(l, ..., l) \mid \textbf{pre}\ l \\
 & & \mid f(l, ..., l) \ \mid \ f(l, ..., l)\ \textbf{every}\ l \\
 & & \mid f(l\ \textbf{when}\ C(x), ..., l\ \textbf{when}\ C(x)) \mid f(l\ \textbf{when}\ C(x), ..., l\ \textbf{when}\ C(x))\ \textbf{every}\ l \\
 & & \mid \textbf{if}\ l\ \textbf{then}\ l\ \textbf{else}\ l \\
 & & \mid \textbf{merge}\ x\ (C \rightarrow l)...(C \rightarrow l) \\
 & & \mid l\ \textbf{when}\ C(x) \\
v & ::= & C :: enum\_ident \mid i :: int \mid r :: real \mid true :: bool \mid false :: bool
\end{array}
$$

Fig. 4. The normalized Lustre syntax.

introducing fresh variables for each automaton to represent its states and encoding transitions in automata as clocked expressions and merges of them. The second phase performs normalization, of which the updated version will be detailed in the following. The main idea of this normalization phase in LustreC is to introduce fresh Lustre variables to encode intermediate values as in classic three-address code. Finally, the last phase translates each normalized node into imperative machine code.

Since the previously presented compilation scheme used by the compiler LustreC is reliable and used to produce trustable C code [5, 22], we adapted it to perform our required simplifications on the Lustre code by modifying the existing normalization stage to produce for each Lustre node a normalized node that can be easily compiled into a Simulink construct, preserving the hierarchy of the initial Lustre nodes. Whereas the original normalization of the LustreC tool was the direct implementation of Biernacki et al. [5], the updated normalization introduces extra variables and associated definition for all operators or function calls, including primitive operators such as arithmetic or logical operators.

The normalization process transforms a Lustre model defined into the grammar of Figure 3 into one of Figure 4. It introduces an additional grammar element l denoting a leaf value (i.e., a variable or a constant). Normalization of an expression returns a fresh typed and clocked variable along with a set of newly bound stateful normalized equations and associated fresh variables. These normalized equations, except for node calls, do not involve nested constructs and correspond to three-address code for binary operators. The arguments of node calls are constants or variables, except for a special case where they are all sampled on the same clock that optimized in Simulink block generation as explained in the following section.

Let us illustrate the normalization of the stopwatch example presented in Listing 1. After normalization, the Lustre code presented in Listing 3 is generated. The original Lustre expressions are given in the comments.

Listing 3 follows the grammar described in Figure 4. The nodes `count` and `stopwatch` are normalized in a classic three-address code so each complex expression is decomposed into simple expressions involving new fresh variables or constants. Each stateful node has a Boolean variable `is_init` denoting its first timestep defined by `true -> false`. The arrow expression `l1→l2` is

replaced by a conditional statement **if** `is_init` **then** `l1` **else** `l2`; see variable `time` in node `count` and streams `__stopwatch_7` and `__stopwatch_2` in node `stopwatch`. Node `stopwatch` contains clocked expressions using **when** and **merge** operators, and their semantics is explained in Section 2.3. Stream `__stopwatch_4` (respectively, `__stopwatch_3`) is clocked on `running` (respectively, `not running`). The expression `(count(tick` **when** `running)`**every** `reset)` is not further normalized since it respects the grammar rule $f(l$ **when** $C(x), \ldots, l$ **when** $C(x))$ **every** l described in Figure 4. The advantage of keeping it unnormalized is explained in the next section.

Listing 3. Normalized Lustre code of the stopwatch example.

```
1  node count (tick: bool) returns (time: int)
2  var is_init: bool; __count_2, __count_3: int;
3  let
4         -- Norm. of: time = 0 -> pre time + 1;
5         is_init = true -> false;
6         __count_2 = pre time;
7         __count_3 = __count_2 + 1;
8         time = if is_init then 0 else __count_3;
9  tel
10
11 node stopwatch (tick: bool; toggle: bool; reset: bool) returns (time: int
       )
12 var running: bool clock; is_init, __stopwatch_5, __stopwatch_6,
       __stopwatch_7: bool;
13 __stopwatch_1, __stopwatch_2 : int;
14 __stopwatch_3: int when not running;
15 __stopwatch_4: int when running;
16 let
17        is_init = true -> false;
18        -- Normalization of: running = ((false -> pre running) <> toggle)
              or reset;
19        __stopwatch_6 = pre running;
20        __stopwatch_7 = if is_init then false else __stopwatch_6;
21        __stopwatch_5 = __stopwatch_7 <> toggle
22        running = __stopwatch_5 or reset;
23        -- Norm. of: (0 -> pre time) when not running
24        __stopwatch_1 = pre time;
25        __stopwatch_2 = if is_init then 0 else __stopwatch_1;
26        __stopwatch_3 = __stopwatch_2 when not running;
27        -- Norm. of: count(tick when running) every reset
28        __stopwatch_4 = count(tick when running) every reset;
29        -- Norm. of: time = merge running (true  -> count(tick when
              running) every reset) (false -> (0 -> pre time) when not
              running);
30        time = merge running
31        (true -> __stopwatch_4) (false -> __stopwatch_3);
32 tel
```

3.2 From Normalized Lustre to Simulink

A normalized node is translated to a Simulink subsystem. We first start translating leaf nodes and then finish with the top nodes. All nodes are translated as subsystems and used as a library of

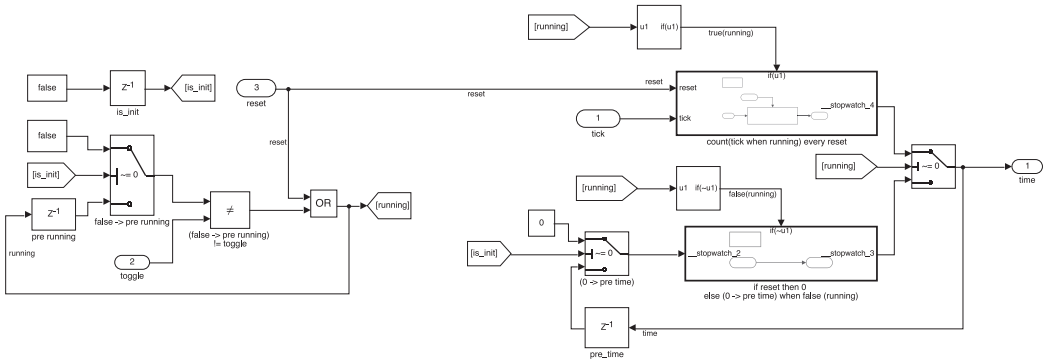


Fig. 5. The Simulink model generated from the Lustre example of Listing 3.

nodes; if a node g calls a node f, then the subsystem that corresponds to node f is instantiated and used inside the subsystem that corresponds to node g.

Each equation definition is mapped to Simulink components. Since both Simulink and Lustre are synchronous dataflow languages, the order of equations is not important. When a Lustre variable is defined, a *Goto* Simulink block is used with the same name as the variable. When the Lustre variable is used in an equation, a *From* Simulink block is used to read from the signal associated with the same variable. For each Lustre variable, there is one *Goto* Simulink block that stores the value of the variable and many *From* Simulink blocks that read from the *Goto* Simulink block.

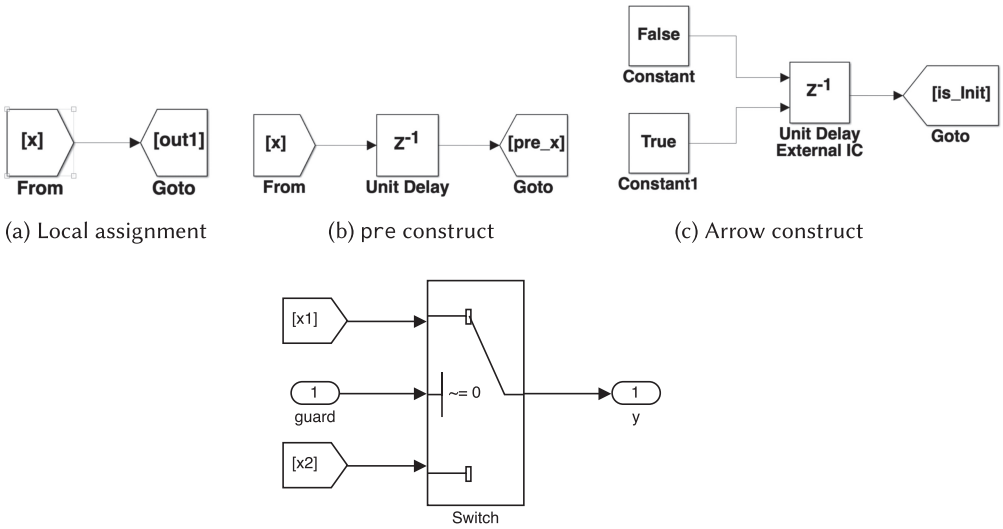
The generated Simulink model of the Lustre example of Listing 3 is illustrated in Figure 5. The user has the choice to keep the *Goto* and *From* blocks or remove them and link signals with the same tag to each other. For readability, we kept only the `is_init` and `running` *Gotos*. We note that the generated Simulink models are sometimes difficult to read when the Lustre source is large. We recommend that engineers always update the Lustre files and regenerate the Simulink blocks rather than try to edit the generated model directly.

To explain the generated Simulink model of Figure 5, we will go over the grammar in Figure 4 of the normalized Lustre code and define the equivalent Simulink components. Each case is illustrated in Figures 6 through 8. We present the following:

(a) Rule $x = l$, local assignment of a variable or constant to another variable (e.g., `out1 = x`). The equation is a simple alias between variables and is modeled in Figure 6(a). A *From* Simulink block is used to read from the signal associated with the variable `x` specified in the tag. A *Goto* Simulink block is used to write on the signal associated with the variable `out1` specified in the tag. In the case of an assignment of a constant $x = C$, a *Constant* Simulink block is used in place of the *From* Simulink block.

(b) Rule $x = pre\ l$, state assignment—that is, a **`pre`** construct over a variable (e.g., `pre_x = pre x`). Thanks to our modified normalization phase, each **`pre`** operator argument is aliased to a fresh variable, here `pre_x`. In Figure 6(b), *Unit Delay* acts as a memory, but its initial value, usually specified in Simulink, is left unused since, in a valid Lustre model, any **`pre`** is guarded by an arrow construct, preventing its use at the first timestep or after a reset.

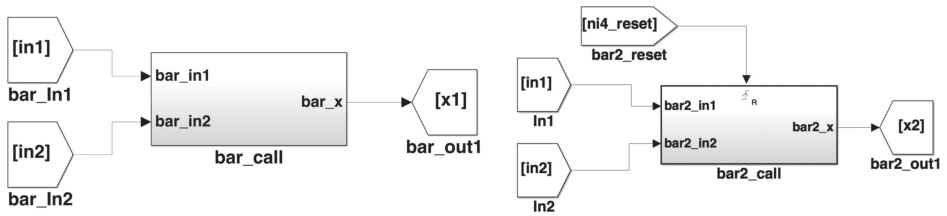
(c) Rule $x = true \rightarrow false$, this is the only arrow construct in our normalized Lustre (e.g., `is_init = true→false`) and is modeled in Figure 6(c). The *Unit Delay* block uses its initial condition `true` at the first step and then the previous value of its input for the following steps. Since the input of the *Unit Delay* in Figure 6(c) is the constant `false`, the *Unit Delay* will produce `false` at all times except the initial step defined by `true`. The *Unit Delay* block could be reset to its initial value by an external signal (see Reset block in Figure 9).



(a) Local assignment (b) pre construct (c) Arrow construct

(d) Branching construct

Fig. 6. Primitive and branching constructs.



(a) Call to a function or node (b) Call to a stateful node with resettable port

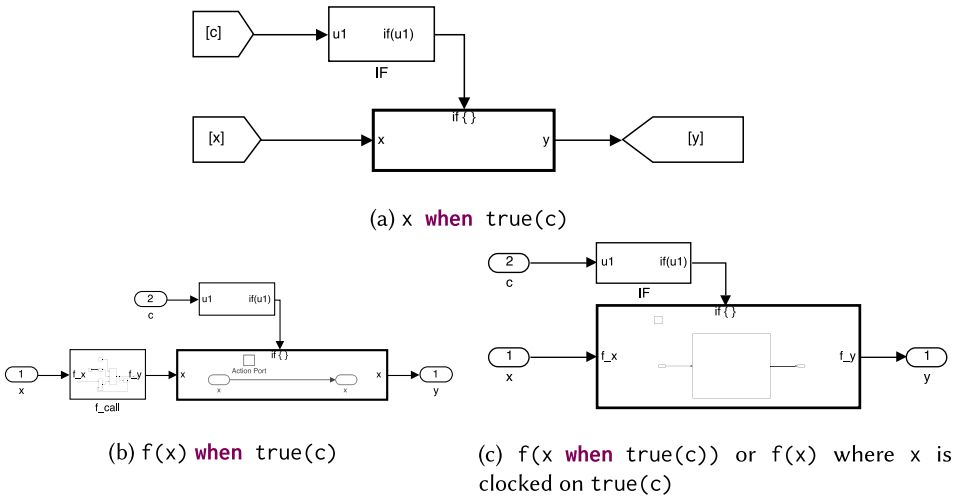
Fig. 7. Node calls.

(d) Rule $x =$ **if** l **then** l **else** l, conditional statements (e.g., `y = if guard then x1 else x2`). Both `x1` and `x2` run on the same clock, and the equation is therefore mapped to a switch as depicted in Figure 6(d). The *Switch* Simulink block uses its second input as a condition, and in Figure 6(d), the condition should be different from zero. The output of the *Switch* Simulink block is its first input if the condition is true; otherwise, it is the third input.

(e) Rule $x =$ **merge** l $(C \rightarrow l)$...$(C \rightarrow l)$, merging construct (e.g., `time = merge running (true→__stopwatch_5)(false→__stopwatch_4)`). Since the Lustre equations are properly clocked, we can soundly represent the merge as a similar branching construct like in Figure 6(d) and assume that input expressions `__stopwatch_5` and `__stopwatch_4` are properly clocked (see Listing 3).

(f) Rules $x = op(l, \ldots, l)$ or $x = f(l, \ldots, l)$ where op is a Lustre binary or unary operator and f is a Lustre node running on the same base clock as its parent node (e.g., `x = bar(in1, in2)`). This is modeled in Figure 7(a) where `bar_call` is an instantiation of subsystem `bar` associated to some Lustre node called `bar`. In the case of an operator of the standard library, such as `x = in1 + in2`, the subsystem `bar_call` would be a basic Simulink block, such as `Add` block, or `Gain` with scalar -1 for unary minus.

Let us finish with two complex constructs: clocked expressions and resetting nodes.



(a) `x when true(c)`

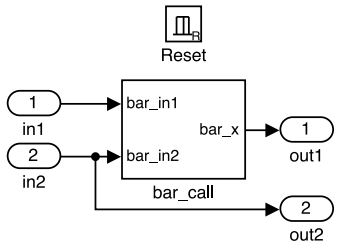
(b) `f(x) when true(c)`

(c) `f(x when true(c))` or `f(x)` where `x` is clocked on `true(c)`

Fig. 8. Branching construct for clocked expressions.

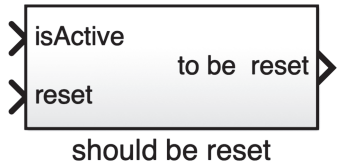
Clocked expressions. Following Biernacki et al. [5], Lustre nodes are considered to be homogeneous in terms of clocks—that is, all input and output flows have to be clocked with the same base clock. However, within the node content, internal flows may be clocked on other signal values, either local signals or inputs. Let *ck* be the base clock of the node. Then, any local clock is defined as a subclock of this base clock *ck*. Thus, any expression and equation, including those listed previously, are clocked, perhaps implicitly through the application of the clock calculus phase, and subject to the transformations presented here. Clocked expressions are any right-hand side of an equation where the defined variable is assigned a specific clock different from the base clock *ck*. In the grammar of the normalized Lustre defined in Figure 4, the right-hand side expressions that can be clocked are $f(l,\ldots,l)$ and $f(l,\ldots,l)$ **every** l if the arguments of node f are clocked on a different clock than the base clock *ck*, expressions $f(l$ **when** $C(x),\ldots,l$ **when** $C(x))$ and $f(l$ **when** $C(x),\ldots,l$ **when** $C(x))$ **every** l, and the expression l **when** $C(x)$. Clocked expressions can be modeled in Simulink with an *If Action subsystem*; it is a subsystem whose execution is enabled by an If block. The If block evaluates a logical expression and then, depending on the result of the evaluation, outputs an action signal that enables the execution of the *If Action subsystem* linked to it. When the latter is not activated, it is configured to either reset its memories (e.g., *Unit Delays* use the initial condition) or hold the previous value. We choose the `Held` feature to ensure that the `If Action subsystem` maintains its state when not active. For instance, in the expression `count(tick when running)`, the node `count` holds the value of the counter when it is `not running`.

The advantage of not further normalizing the equation `y = f(x when true(c))` is illustrated in Figure 8(c); the argument `x` does not need to be clocked twice, since node `f` is only executed when condition `c` is positive. The equivalent normalized Lustre code is `x2 = x when true(c); y = f(x2);`, where both expressions `x when true(c)` and `f(x2)` will be embedded inside two different *If Action Subsystems* with the same condition. The first *If Action Subsystem* is shown in Figure 8(a), and the second is shown in Figure 8(c). This special construction is also handled since first it occurs frequently in our LustreC compilation chain and second it exempts us from implementing a *If Action subsystem* factorization algorithm.

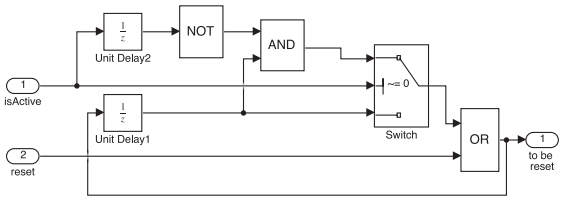



```
node foo (in1, in2: int) returns (out1,
    out2: int);
var x : int;
let
  x = bar(in1, in2); -- a stateful node
  out1 = x;
  out2 = in2;
tel
```

Fig. 9. Stateful Lustre node call is translated as a Simulink Resettable subsystem when the node `foo` is called with **`every`** operator.



(a) The subsystem keeping track if an `Action Subsystem` should be reset.



(b) The content of the "should be reset" subsystem

Fig. 10. `Action Subsystem` with `Held` feature might need to be reset when it is re-activated.

Note that the expression `f(x) when true(c)` is different since the call to `f(x)` is running on the base clock and then its output is sampled on the clock `c`. In fact, the equation `y = f(x)when true(c)` is normalized into `x2 = f(x); y = x2 when true(c);`, and the equivalent Simulink blocks are shown in Figure 8(b).

Stateful nodes and reset. In Figure 9, the called node `bar` is stateful: either it contains an arrow function, and typically **`pre`** expressions, or it calls other stateful nodes.

Since `foo` contains a stateful node, it is itself stateful. The definition of the node as a resettable subsystem as suggested in Figure 7(b) will recursively reset each memory in the node and its children, performing the expected behavior. This produces the Simulink diagram presented in Figure 9.

There is, however, a case where this encoding is erroneous. Indeed, in Simulink, if a resettable subsystem contains `Action Subsystems` with the `Held` feature, then the reset action is not propagated within these `Action Subsystems`. The only place where we use these `Action Subsystems` in our translation is in the treatment of clocked expressions explained earlier. Therefore, in the case of a node conditionally reset with an **`every`** cond and containing subexpressions defined over subclocks, then one needs to explicitly propagate the reset signal to these `Action Subsystems`.

The generation of subsystems associated with clocked node calls should then be explicitly extended with reset inputs, adding memory to record the reset status of the node for clocked substreams. The subsystem (that keeps track of the reset status is presented in Figure 10. Its output is added as an input of the `Action Subsystem` and the `isActive` input is the condition associated with the `If Action Subsystem`, whereas the `reset` input is the reset condition of the parent subsystem. The output "`to be reset`" is positive when the reset input is positive or the `Action Subsystem` is re-activated and "`to be reset`" was previously active. If the `Action Subsystem` is not active, the previously "`to be reset`" value is kept. For instance, in Table 2,

Table 2. A Simulation of the "Should Be Reset" Subsystem of Figure 10

`isActive`	False	False	False	True	True
`reset`	False	True	False	False	False
`to be reset`	False	True	True	True	False

we give an example of an execution of the subsystem presented in Figure 10. The `If Action Subsystem` was inactive for the first three steps, and the `reset` signal was active for the second timestep. The `If Action Subsystem` should be reset when it was re-activated at the fourth step, and the "`to be reset`" signal is going to negative at the fifth step since the `reset` input was not triggered. Thanks to the extensive validation we performed, we are confident that this encoding faithfully addresses this specific case. The validation process is detailed in the following section.

4 EVALUATION

Our approach was evaluated on a set of case studies, from small benchmarks taken from our regression test suite to industrial ones, using equivalence checking, which will be defined in the following.

Let us denote by $\mathcal{L}$ and $\mathcal{S}$ the sets of Lustre and Simulink models. Model semantics is denoted by $[\![\cdot]\!]_L$ for $L \in \mathcal{L}$ and $[\![\cdot]\!]_S$ for $S \in \mathcal{S}$. Each function is a (possibly stateful) map from a set of n typed input flows to m typed output flows (e.g., $[\![L]\!]_L : \mathcal{T}^n \to \mathcal{T}^m$).

The objective of equivalence checking is to verify that two models, in the same language, are equivalent. For Lustre models $L_1, L_2 \in \mathcal{L}$, we say that L_1 is equivalent to L_2, denoted by $L_1 \equiv_L L_2$ if and only if for any input flow $i \in \mathcal{T}^n$, $[\![L_1]\!]_L(i) = [\![L_2]\!]_L(i)$. The property $\equiv_S$ is defined similarly on Simulink models. Behavioral equivalence can be evaluated either at the Simulink level with Simulink Design Verifier or at the Lustre level with the Kind2 model checker [12, 28]. It can also be evaluated through tests when formal tools cannot conclude.

One can consider the compilation process of the CoCoSim toolbox from Simulink to Lustre as a map $S2L : \mathcal{S} \to \mathcal{L}$. Similarly, the algorithm proposed in Section 3 characterizes a map $L2S : \mathcal{L} \to \mathcal{S}$. We assume that $S2L$ is sound, and we target specifically the validation of $L2S$. Therefore, we consider the following verification challenges:

$$\text{For all model } S \in \mathcal{S}, \quad S \equiv_S L2S \circ S2L(S), \tag{1}$$

$$\text{For all model } L \in \mathcal{L}, \quad L \equiv_L S2L \circ L2S(L). \tag{2}$$

Let us add that in both cases, there is a single call to $L2S$ and that the function $S2L$, implemented in the CoCoSim toolbox, shares no code or algorithm with the LustreC tool used to implement the function $L2S$.

The results are presented in Table 3. The first three columns give metrics about the size of the models. Simulink Design Verifier was applied to the top level of the system. Lustre equivalence checking (cf. Equation (2)) using Kind2 has been used both on the main node and modularly, considering each subnode as a verification target.

Concerning the 89 Lustre benchmarks from our regression suite (see the first line of Table 3), 85 benchmarks contain only one large top node with 1,600 code lines and 465 variables on average. Since there is a single top node in these benchmarks, applying compositional verification does not improve the results. Table 3 shows that model checking using a global (top-level) encoding both in Simulink Design Verifier and Lustre was able to prove 87 benchmarks but unable to prove the validity of two remaining benchmarks. We applied compositional verification on the two remaining benchmarks that were hard to prove due to their complexity. For the first benchmark, 9 out of 16

Table 3. Experiments

	#loc	#nodes	#vars	Simulink Design Verifier	Lustre Model Checking	
					Monolithic	Compositional
89 Benchmarks [L]	289,181	160	43,001	87 S + 2 U	87 S + 2 U	152 S + 8U
TCM [S]	2,040	91	1,239	U	U	79 S + 12 U
ROSACE [S]	926	94	520	U	U	87 S + 7 U
FADEC [S]	68	1	46	U	S	S
AOCS [S]	3,649	93	3,390	U	S	79 S + 14 U

S, safe (proven valid); U, unknown (i.e., unable to conclude with model checkers; to be evaluated through tests). [L] use cases were initially defined in Lustre and [S] in Simulink. Note the larger set of cases in the last column since it considers all subnodes as intermediate challenges.

nodes were proved safe, and one node was proved safe by k-induction [28, 37]. Similar results were obtained on the second benchmark. Nodes that were hard to validate were hierarchical state machines expressed in Lustre. Lustre automata are compiled into pure dataflow equations, encoding transitions as clocked expressions, which explains that the final Lustre code is more complicated than the original model. All unproved nodes were validated by equivalence testing.

The approach was also applied to four industrial Simulink benchmarks: the NASA **Transport Class Model (TCM)** [10], the ROSACE use case [34], a **Full Authority Digital Engine Control (FADEC)**, and a CNES Attitude and **Orbital Control System (AOCS)**. All of these benchmarks were analyzed using Equation (1).

Table 3 shows the effectiveness of compositional verification compared to monolithic verification. Simulink Design Verifier was unable to globally prove any model. This can be explained by both the use of nonlinear arithmetic operators, which are hardly analyzed by solvers, and the size of the model. Using model checking on the Lustre global encoding, we were able to prove two of the four models. Compositional verification in Lustre shows better performance: more than 85% of nodes are proved safe. Equivalence testing was applied to the unproved nodes. For the AOCS case study, the fact that we were able to globally prove the system with Lustre but unable to prove all of the corresponding nodes can be explained by the elimination of some behaviors difficult to prove for particular nodes when considering the global system.

5 APPLICATIONS

Producing Simulink subsystems from Lustre models has several advantages. We mention a few of them next.

5.1 Easing the Formalization of Requirements at the Model Level

An essential step when it comes to supporting the formalization of requirements is the capability to add the specification to a model. Most of the tools handling formalized requirements use some formal annotation and formal languages to express these requirements. For instance, the AGREE framework [31] and FRET [23] formalization tool use Lustre and CoCoSpec, respectively, to express requirements. We integrated our work in CoCoSim and connected it with FRET output, automatically translating CoCoSpec contracts generated by FRET to contracts expressed in Simulink and supported by CoCoSim. This work was applied to publicly available industry-provided examples[2] from Lockheed Martin Cyber-Physical Systems challenges [17, 18], which is a set of aerospace-inspired examples provided as text documents specifying the requirements along with

[2] https://github.com/hbourbouh/lm_challenges.

associated Simulink models. Examples range from a basic integrator to complex autopilots. The complete case study and formalized requirements are presented in a detailed technical report [33].

5.2 Generation of Runnable Evidence at the Simulink Level

The initial motivation for this work came from the use of the property-directed reachability-based tool Zustre [20] to analyze synchronous observers associated with Simulink models by translating them to Lustre before analyzing them. The Zustre model checker can provide a counterexample in case of failure and also returns a set of invariants in case of success. However, although traceability information was sufficient to execute the counterexample on the initial Simulink model, the expression of the produced invariant as runnable evidence was not possible. More specifically, the hierarchy-preserving encoding of the Lustre model into the model checker provides, in case of success, a set of local invariants that could be attached to Simulink subsystems. As an example, Listing 4 presents such a generated local invariant (can be read as (**pre**(time)>= 0 => time >= 0)). The Lustre to Simulink translation allows attaching this property to a subsystem as a synchronous observer.

Listing 4. Example of a generated Lustre invariant for Stopwatch example.

```
node inv(toggle, reset : bool; time:int;) returns (inv:bool);
let
 inv = true ->  (time >= 0 or  not (pre(time) >= 0));
tel
```

5.3 Generation of Lustre Annotations at the Simulink Level

The LustreC compiler can generate the MC-DC coverage criterion as Lustre annotations in Lustre code. In addition, these annotations are included in the compilation process described in Section 3. These annotations could be attached to Simulink subsystems.

Listing 5 provides an example of generated local MC-DC coverage conditions. All MC-DC coverage conditions can be added to a subsystem as a synchronous observer (cf. Figure 11(a)). We use it to calculate the coverage of a given test suite by simulation. Figure 11 illustrates all 10 MC-DC conditions of the expression *out* in Listing 5.

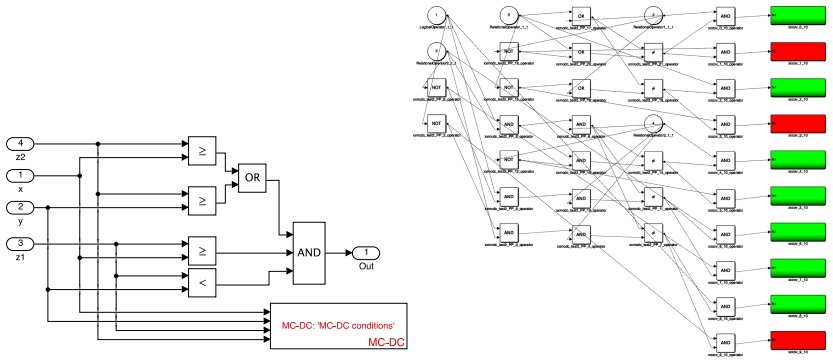
Listing 5. Example of a Lustre annotation.

```
node top (x, y, z1, z2: int) returns (out: bool)
var __cov_1: bool;
let
  out = ((((z2 >= x) or (z2 >= y)) and (z1 >= x)) and (z1 < y));
  --The following is a special annotation:
  (*! /coverage/mcdc/:  __cov_1; *)
  __cov_1 = ((z2 >= y) and (((z2 >= x) or (z2 >= y)) != ((z2 >= x) or (
      not ((z2 >= y))))));
tel
```

5.4 Transforming Simulink Models into Equivalent Simpler Models

As presented in Section 3, the set of generated constructs in Simulink is limited: basic operators, logically executed subsystems with action blocks, and unit delays. However, the input language accepted by the CoCoSim toolchain is much larger.

The combination of the CoCoSim compilation of a Simulink+Stateflow model to Lustre and its translation back to simple Simulink provides an interesting feature. Addressing the analysis of the



(a) (b)

Fig. 11. (a) MC-DC observer attached to its subsystem. (b) MC-DC conditions subsystem from inside. Each condition is colored green if it was covered during simulation and red otherwise.

Table 4. Number of Unique Block Types in the Original TCM Model vs. the Simplified Version

Model	Original Model		Simplified Model	
	#blocks	#unique block types	#blocks	#unique block types
TCM	570	27	7587	14

large set of constructs considered, such as Stateflow blocks or conditionally activated subsystems, is then reduced to the minimal subset of basic constructs.

For instance, it could support the definition of new analysis tools that could concentrate the effort on the handling of this restricted subset of Simulink constructs while addressing a much larger set of input models. In Table 4, we give the example of the TCM model from Table 3. The original model contains around 27 unique blocks. After simplification of the model, we get 14 unique blocks (*Constant*, *Delay*, *From*, *Gain*, *Goto*, *Inport*, *Logic*, *Outport*, *Product*, *RelationalOperator*, *Subsystem*, *Sum*, *Switch*, *Trigonometry*) consisting of almost all of the basic blocks we use.

6 CONCLUSION

The presented approach enables the translation of Lustre nodes to Simulink subsystems. The proposed algorithm can be used to produce regular subsystems or to support the definition of contracts at the Simulink level, using Boolean flows.

The added value of our approach to alternative approaches such as those of Liu et al. [31] is the production of basic Simulink subsystems relying only on primitive blocks such as unit delays, merge, and relational and arithmetic operators. It is also capable of addressing the complete input language of the compiler we used. Particularly, we can handle clocks, hierarchical definition through multiple Lustre nodes and Lustre automata. The implementation is however limited since it does not yet handle machine-level types nor external C functions, although this could technically be implemented since Simulink supports both constructs.

The approach has been validated on large use cases, demonstrating the behavioral equivalence between some compiled models.

The applications are numerous, from validation of the framework to support of formal specification or production of runnable proof evidence as synchronous observers. It is now integrated

into the CoCoSim toolbox and is mature enough to be used automatically to provide feedback at the model level.

Future work includes the extension of the input language to enable the use of externally defined functions, such as C code, and the handling of machine data types (i.e., int8, uint8, int16, uint16 ...).

REFERENCES

[1] RTCA. 2011. *DO-178C. Software Considerations in Airborne Systems and Equipment Certification.* RTCA.

[2] AdaCore and Altran UK Ltd. 2020. SPARK 2014 Reference Manual. Retrieved June 3, 2021 from http://docs.adacore.com/spark2014-docs/html/lrm/.

[3] Patrick Baudin, Pascal Cuoq, Jean-Christophe Filliâtre, Claude Marché, Benjamin Monate, Yannick Moy, and Virgile Prevosto. 2020. ACSL: ANSI/ISO C Specification Language, Version 1.16. Retrieved June 3, 2021 from https://www.frama-c.com/download/acsl.pdf.

[4] Albert Benveniste and Gérard Berry. 1991. *The Synchronous Approach to Reactive and Real-Time Systems.* Research Report RR-1445. INRIA. https://hal.inria.fr/inria-00075115.

[5] Dariusz Biernacki, Jean-Louis Colaço, Grégoire Hamon, and Marc Pouzet. 2008. Clock-directed modular code generation for synchronous data-flow languages. In *Proceedings of the 2008 ACM SIGPLAN-SIGBED Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES'08).* ACM, New York, NY, 121–130. https://doi.org/10.1145/1375657.1375674

[6] Hamza Bourbouh. n.d. CoCoSim—Automated Analysis Framework for Simulink. Retrieved June 3, 2021 from https://github.com/NASA-SW-VnV/CoCoSim.

[7] Hamza Bourbouh, Pierre-Loïc Garoche, Christophe Garion, Arie Gurfinkel, Temesghen Kahsai, and Xavier Thirioux. 2017. Automated analysis of Stateflow models. In *LPAR-21, 21st International Conference on Logic for Programming, Artificial Intelligence and Reasoning, Maun, Botswana, May 7-12, 2017.* EPiC Series in Computing, Thomas Eiter and David Sands (Eds.), Vol. 46. EasyChair, 144–161. http://www.easychair.org/publications/paper/340361.

[8] Hamza Bourbouh, Pierre-Loïc Garoche, Thomas Loquen, Éric Noulard, and Claire Pagetti. 2020. CoCoSim, a code generation framework for control/command applications: An overview of CoCoSim for multi-periodic discrete Simulink models. In *Proceedings of the 10th European Congress on Embedded Real Time Software and Systems (ERTS'20).*

[9] Timothy Bourke, Lélio Brun, Pierre-Evariste Dagand, Xavier Leroy, Marc Pouzet, and Lionel Rieg. 2017. A formally verified compiler for Lustre. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'17).* ACM, New York, NY. https://hal.inria.fr/hal-01512286.

[10] Guillaume Brat, David H. Bushnell, Misty Davies, Dimitra Giannakopoulou, Falk Howar, and Temesghen Kahsai. 2015. Verifying the safety of a flight-critical system. In *FM 2015: Formal Methods.* Lecture Notes in Computer Science, Vol. 9109. Springer, 308–324. https://doi.org/10.1007/978-3-319-19249-9_20

[11] Paul Caspi, Daniel Pilaud, Nicolas Halbwachs, and John Plaice. 1987. Lustre: A declarative language for programming synchronous systems. In *Proceedings of the 14th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages (POPL'87).* 178–188.

[12] Adrien Champion, Arie Gurfinkel, Temesghen Kahsai, and Cesare Tinelli. 2016. CoCoSpec: A mode-aware contract language for reactive systems. In *Software Engineering and Formal Methods.* Lecture Notes in Computer Science, Vol. 9763. Springer, 347–366. https://doi.org/10.1007/978-3-319-41591-8_24

[13] Adrien Champion, Alain Mebsout, Christoph Sticksel, and Cesare Tinelli. 2016. The Kind 2 model checker. In *Computer Aided Verification.* Lecture Notes in Computer Science, Vol. 9780. Springer, 510–517. https://doi.org/10.1007/978-3-319-41540-6_29

[14] Alessandro Cimatti and Stefano Tonetta. 2012. A property-based proof system for contract-based design. In *Proceedings of the 2012 38th Euromicro Conference on Software Engineering and Advanced Applications (SEAA'12).* 21–28. https://doi.org/10.1109/SEAA.2012.68

[15] Jean-Louis Colaço, Bruno Pagano, and Marc Pouzet. 2005. A conservative extension of synchronous data-flow with state machines. In *Proceedings of the 5th ACM International Conference on Embedded Software (EMSOFT'05).* 173–182.

[16] Arnaud Dieumegard, Pierre-Loïc Garoche, Temesghen Kahsai, Alice Taillar, and Xavier Thirioux. 2015. Compilation of synchronous observers as code contracts. In *Proceedings of the 30th Annual ACM Symposium on Applied Computing (SAC'15).* 1933–1939.

[17] Chris Elliott. 2016. An example set of cyber-physical V&V challenges for S5, Lockheed Martin Skunk works. In *Proceedings of the 2016 Safe and Secure Systems and Software Symposium (S5'16).* http://mys5.org/Proceedings/2016/Day_2/2016-S5-Day2_0945_Elliott.pdf.

[18] Chris Elliott. 2015. On example models and challenges ahead for the evaluation of complex cyber-physical systems with state of the art formal methods V&V, Lockheed Martin Skunk works. In *Proceedings of the 2015 Safe and Secure Systems and Software Symposium (S5'15).* http://mys5.org/Proceedings/2015/Day_1/2015-S5-Day1_1405_Elliott.pdf.

[19] Andrew Gacek, John Backes, Mike Whalen, Lucas G. Wagner, and Elaheh Ghassabani. 2018. The JKind model checker. In *Computer Aided Verification*. Lecture Notes in Computer Science, Vol. 10982. Springer, 20–27. https://doi.org/10.1007/978-3-319-96142-2_3

[20] Pierre-Loïc Garoche, Arie Gurfinkel, and Temesghen Kahsai. 2014. Synthesizing modular invariants for synchronous code. In *Proceedings of the 2014 Workshop on Horn Clauses for Verification and Synthesis (HCVS'14)*. https://doi.org/10.4204/EPTCS.169.4

[21] Pierre-Loïc Garoche, Temesghen Kahsai, and Xavier Thirioux. 2016. Hierarchical state machines as modular horn clauses. In *Proceedings of the 2016 Workshop on Horn Clauses for Verification and Synthesis (HCVS'16)*. 15–28. https://doi.org/10.4204/EPTCS.219.2

[22] Pierre-Loïc Garoche, Falk Howar, Temesghen Kahsai, and Xavier Thirioux. 2014. Testing-based compiler validation for synchronous languages. In *Proceedings of the NASA Formal Methods Symposium*. 246–251.

[23] Dimitra Giannakopoulou, Thomas Pressburger, Anastasia Mavridou, and Johann Schumann. 2020. Generation of formal requirements from structured natural language. In *Requirements Engineering: Foundation for Software Quality*. Lecture Notes in Computer Science, Vol. 12045. Springer, 19–35.

[24] N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud. 1991. The synchronous dataflow programming language LUSTRE. *Proceedings of the IEEE* 79, 9 (1991), 1305–1320.

[25] Nicolas Halbwachs, Fabienne Lagnier, and Pascal Raymond. 1993. Synchronous observers and the verification of reactive systems. In *Algebraic Methodology and Software Technology (AMAST'93)*. Workshops in Computing. Springer, 83–96.

[26] Nicolas Halbwachs and Pascal Raymond. 1999. Validation of synchronous reactive systems: From formal verification to automatic testing. In *Advances in Computing Science—ASIAN'99*. Lecture Notes in Computer Science, Vol. 1742. Springer, 1–12. https://doi.org/10.1007/3-540-46674-6_1

[27] C. A. R. Hoare. 1969. An axiomatic basis for computer programming. *Communications of the ACM* 12, 10 (1969), 576–580. https://doi.org/10.1145/363235.363259

[28] Temesghen Kahsai and Cesare Tinelli. 2011. PKind: A parallel k-induction based model checker. In *Proceedings of the 10th International Workshop on Parallel and Distributed Methods in Verification (PDMC'11)*. 55–62. https://doi.org/10.4204/EPTCS.72.6

[29] Andreas Katis, Grigory Fedyukovich, Andrew Gacek, John D. Backes, Arie Gurfinkel, and Michael W. Whalen. 2016. Synthesis from assume-guarantee contracts using skolemized proofs of realizability. arXiv:1610.05867.

[30] Uri Klein and Amir Pnueli. 2010. Revisiting synthesis of GR(1) specifications. In *Hardware and Software: Verification and Testing*. Lecture Notes in Computer Science, Vol. 6504. Springer, 161–181. https://doi.org/10.1007/978-3-642-19583-9_16

[31] Jing Liu, John D. Backes, Darren Cofer, and Andrew Gacek. 2016. From design contracts to component requirements verification. In *NASA Formal Methods*. Lecture Notes in Computer Science, Vol. 9690. Springer, 373–387. https://doi.org/10.1007/978-3-319-40648-0_28

[32] Mathworks Inc. 2018. *MATLAB Simulink (R2018b)*. Mathworks Inc., Natick, MA.

[33] Anastasia Mavridou, Hamza Bourbouh, Pierre-Loïc Garoche, and Mohammad Hejase. 2019. *Evaluation of the FRET and CoCoSim Tools on the Ten Lockheed Martin Cyber-Physical Challenge Problems*. Technical Report NASA/TM-2019-220374. NASA. http://www.garoche.net/publication/nasatm-2019-220374/nasatm-2019-220374.pdf.

[34] Claire Pagetti, David Saussié, Romain Gratia, Eric Noulard, and Pierre Siron. 2014. The ROSACE case study: From Simulink specification to multi/many-core execution. In *Proceedings of the 2014 IEEE 19th Real-Time and Embedded Technology and Applications Symposium (RTAS'14)*. https://doi.org/10.1109/RTAS.2014.6926012

[35] John Rushby. 2014. The versatile synchronous observer. In *Specification, Algebra, and Software*. Lecture Notes in Computer Science, Vol. 8373. Springer, 110–128.

[36] Norman Scaife, Christos Sofronis, Paul Caspi, Stavros Tripakis, and Florence Maraninchi. 2004. Defining and translating a "safe" subset of Simulink/Stateflow into Lustre. In *Proceedings of the 4th ACM International Conference on Embeeded Software (EMSOFT'04)*. 259–268.

[37] Mary Sheeran, Satnam Singh, and Gunnar Stålmarck. 2000. Checking safety properties using induction and a SAT-solver. In *Proceedings of the International Conference on Formal Methods in Computer-Aided Design (FMCAD'00)*. 127–144. https://doi.org/10.1007/3-540-40922-X_8

[38] Stavros Tripakis, Christos Sofronis, Paul Caspi, and Adrian Curic. 2005. Translating discrete-time Simulink to Lustre. *ACM Transactions on Embedded Computing Systems* 4, 4 (2005), 779–818. https://doi.org/10.1145/1113830.1113834

[39] M. W. Whalen, A. Gacek, D. Cofer, A. Murugesan, M. P. E. Heimdahl, and S. Rayadurgam. 2013. Your what is my how: Iteration and hierarchy in system design. *IEEE Software* 30, 2 (March 2013), 54–60. https://doi.org/10.1109/MS.2012.173