Determining Robustness of Synchronous Programs under Stuttering

Ingram Bondin and Gordon J. Pace
University of Malta
Malta
ingrambondin@gmail.com, gordon.pace@um.edu.mt

***Abstract*—Robustness of embedded systems under potential changes in their environment is crucial for reliable behaviour. One typical environmental impact is that of the inputs being slowed down — due to which, the system may no longer satisfy its specification. In this paper, we present a framework for analysing the behaviour of synchronous programs written in Lustre under such environmental interference. Representing slow input by stuttering, we introduce both strong and weak slowdown robustness constraints with respect to this phenomenon. Furthermore, static and dynamic algorithmic techniques are used to deduce whether such constraints are satisfied, and the relationship between stateful programs and the slowdown model considered is explored.**

Keywords–Synchronous Languages; Lustre; Slowdown; Stutter

I. INTRODUCTION

Software is increasingly becoming more prominent as a controller for a variety of devices and processes. Embedded systems operate within an environment, by which they are affected and with which they interact — this tight interaction usually means that changes to the environment directly change the behaviour of the embedded system. One such situation can occur when the environment slows down its provision of input to the system, possibly resulting from a variety of reasons. For example, the system producing the inputs or the communications channel on which these inputs pass to the program might be under heavy load, delaying the inputs; or the program is deployed on a faster platform, therefore making the input relatively slower.

One question that arises immediately in such scenarios is how the system behaves when its input slows down. Does it act in an expected manner, or does the slow input cause it to produce unwanted output? In this paper (which is an extended version of [1]), we develop an approach to study whether a system continues to behave correctly when the inputs are slowed down. This leads us to different notions of robustness to slow input since, for instance, in some cases we may desire the output to be delayed by the same amount as the inputs, whereas in others, the values but not the actual delays on the outputs are important.

The theory we develop is applied to the synchronous language Lustre [2], which enables the static deduction of a program's resource requirements, making it ideal for the design of embedded systems. Although retiming analysis techniques for continuous time can be found in the literature [3], our approach adapts them for discrete time, the timing model used by Lustre and other synchronous languages.

Such a theory requires addressing a number of considerations. In Section II we define streams [4], which are infinite sequences of values, as well as the Lustre programs that manipulate them. In the model we adopt, streams can be slowed down through the repetition of values, which is also called stuttering. Stuttering can be a valid model for slow input under several scenarios:

- If a memory's clock signal becomes slower, the memory will take more time to read new input, and thus will maintain its present output for a longer time. A program that samples the values of this memory at the same rate will then experience repetition in its input.
- The system providing the input might not be ready to provide its output, or it might experience a fault from which it needs time to recover. In these situations, some systems might keep their present output constant until they are ready once again. In this case, the receiving program will also experience repetition in its input.
- A physical process that is being sampled in order to provide input to a program might slow down. Under certain sampling conditions, the resulting input received by the program corresponds to experiencing stutter in its inputs.

The effect of slow input on program behaviour when modelled through stutter, provides a more complex scenario than that considered in the literature [3], [4], [5], [6], [7]. In particular, additional input symbols from a slow stream can change a program's internal state, requiring a more complex analysis to determine whether it behaves correctly. Our use of the Lustre language (with its simple semantics) will provide a useful setting for studying how program state and this type of slow input interact.

In Section III we identify a number of robustness properties that characterise acceptable program behaviour to slow input in a number of different scenarios. We consider both properties similar to those found in simpler slowdown models as well as weaker ones which are useful in our kind of model.

Given a robustness property, one desires an algorithmic way of checking whether or not it holds for a given program. Sections IV and V address this issue. Section IV considers a method based on the static analysis of the program's text, and yields compositionality results for the properties being considered. On the other hand, Section V focuses on a method based on the dynamic analysis of a program's state space.

The analysis techniques we present focus on Boolean (or finite type) Lustre programs. Although a number of results can be lifted to programs over arbitrary types, this does not apply in general and will be regarded as outside the scope of this work.

To demonstrate the application of the analysis techniques, we apply the two aforementioned approaches to various Boolean Lustre programs in Section VI. Finally, in Section VII we compare our work to existing results, while in Section VIII we give some concluding remarks. It should be noted that this paper is an extended version of [1]. The main extensions are the following.

- In Section III, we provide a proof that all robustness properties considered are relaxations of a strong property known as stretch robustness.
- In Section IV, we provide detailed proofs for all the static analysis results stated.
- In Section V, we give a new strong condition for showing that a program is not stretch robust, and a new condition for showing that a program is stretch robust without relying on the program being effectively stateless. This resolves the question of whether there are non-trivial stateful programs that satisfy stretch robustness when using a stutter based slowdown model.
- In Section VI, we apply the new conditions discussed in Section V to the programs considered in [1].

II. Streams, Slowdown and Lustre Programs

We adopt the standard view of a stream s as an infinite sequence of values over a particular type, representing the value of the stream over a discrete time domain. We shall write $s(t)$ to denote the value taken by stream s at time t.

Given a number of streams $s_1, \ldots, s_n$ we will find it convenient to collect these into a vector of streams $v = \langle s_1, \ldots, s_n \rangle$. In this case, we shall use the notation $v(s_i)$ to denote the stream s_i in the vector. Two vectors v_1 and v_2, can be combined, modulo renaming of streams, into one vector $v_1 \cup v_2$, containing all the streams from these two vectors.

Assuming a vector of streams $v = \langle s_1, \ldots, s_n \rangle$, we denote the behaviour of all streams at a particular time t by $v(t)$, which will yield the tuple of values $(s_1(t), \ldots, s_n(t))$.

By slowing down a stream, one obtains the same sequence of values, but possibly with some of the values repeated a number of times, representing stutter. A slowdown can be characterised using a latency function — a total function that returns the number of times each value in the stream will stutter for. Given a stream s that is slowed down according to a latency function λ, one obtains the slowed down stream s_λ:

$$s_\lambda = \underbrace{s(0), \ldots s(0)}_{\lambda(0)+1}, \underbrace{s(1), \ldots s(1)}_{\lambda(1)+1}, \ldots \underbrace{s(n), \ldots s(n)}_{\lambda(n)+1} \ldots$$

Note that s_λ is obtained from s by replacing the value of s at time t by a block of of $\lambda(t) + 1$ copies of this value. We will write Start_t^λ to denote the time instant at which the t^{th} such block begins: $\sum_{i=0}^{t-1} \lambda(i)$. Similarly, End_t^λ denotes the time instant at which the block ends and is analogously defined.

Note that the constant zero latency function leaves the stream untouched. If a latency function is a constant function, we shall refer to it as *uniform*.

As before, we will extend this notation for vectors of streams, with $\langle s_1, \ldots s_n \rangle_\lambda$ being equivalent to $\langle s_{1\lambda}, \ldots, s_{n\lambda} \rangle$. From this it is easy to derive the useful fact that latency functions distribute over vector union, giving us the identity $(v_1 \cup v_2)_\lambda = (v_{1\lambda} \cup v_{2\lambda})$.

Lustre [2] provides a way of symbolically specifying systems that process streams in a declarative manner. A Lustre program $P = \langle V, I, O, E \rangle$ is defined over a set of stream variables V, with two disjoint subsets I and O consisting of the input and output stream variables of the program, respectively, and a set of equations E that explains how to compute the value of each output variable at every instant of time in terms of other program variables. Equations can take one of the following forms:

$$\begin{array}{lcl} y & = & \otimes(x_1, \ldots x_n) \\ y & = & \mathit{pre}\ x_1 \\ y & = & x_1\ \texttt{->}\ x_2 \\ y & = & x_1\ \mathit{fby}\ x_2 \end{array}$$

Instantaneous operators $\otimes$ are used to represent computation performed at each time instant. For instance, the equation $y = \wedge(x_1, x_2)$ would update the value of stream variable y with the value of the conjunction of the stream variables x_1 and x_2 at each time instant: $y(t) = x_1(t) \wedge x_2(t)$. The *delay* operator *pre* allows access to the previous value of a given stream variable: $(\mathit{pre}\ x)(t+1) = x(t)$ with the resulting stream being undefined for the initial time point, at which it is said to take the value *Nil*. In fact, *pre* behaves like an uninitialised memory. The *initialisation* operator x_1 -> x_2 yields a stream behaving like x_1 at the first time instant, and like x_2 elsewhere: $(x_1$ -> $x_2)(0) = x_1(0)$ and $(x_1$ -> $x_2)(t+1) = x_2(t+1)$. These last two operators are frequently combined to produce an initialised memory using the *followed-by* operator, with x_1 *fby* x_2 being equivalent to x_1 -> *pre* x_2.

Below we illustrate two sample programs. The program Toggle represents a toggle switch that starts in the Boolean state *true*, and which outputs its present state if its toggle input is *false* and inverts and outputs its present state if the toggle input is *true*. On the other hand, the program Siso is a 4-bit serial in serial out register, which starts with all its memories set to true.

```
node Toggle(toggle : bool)
returns(out : bool);
var X, Y : bool;
let
   out = if toggle then x else y;
   x = not y;
   y = true fby out
tel;
```

```
node Siso(i1 : bool)
returns(i5 : bool);
var i2, i3, i4 : bool;
let
   i2 = true fby i1;
   i3 = true fby i2;
   i4 = true fby i3;
   i5 = true fby i4;
tel;
```

We will use the notation P_{inst}, P_{delay}, P_{init}, and P_{fby} for the primitive programs with just one equation consisting of a single application of an instantaneous, delay, initialisation or

followed-by operator, respectively. For each primitive program, the variable occurring on the left hand side of its equation is an output variable, those appearing on the right are inputs.

For a Lustre program P, $\text{dep}_0(P) \subseteq V \times V$ relates a stream variable y to a stream variable x if y is defined in P by an equation with x appearing on the right hand side. The irreflexive transitive closure of this relation denotes the dependencies between the stream variables and is written as $\text{dep}(P)$. Another important concept is that of an instantaneous dependency relation. This relation can be obtained by starting from the relation $\text{inst}_0(P) \subseteq V \times V$, which relates a stream variable y to a stream variable x only if y's defining equation involves x, and x does not appear in a pre equation or on the right hand side of an fby equation. The irreflexive transitive closure of this relation, $\text{inst}(P)$ denotes the instantaneous dependencies between stream variables. A Lustre program P is said to be well-formed if none of its variables instantaneously depend on themselves: $\forall s \cdot (s, s) \notin \text{inst}(P)$.

Given two Lustre programs P_1 and P_2 (with inputs I_1, I_2 and outputs O_1, O_2, respectively) their composition, written $P_1 \mid P_2$, is the Lustre program whose equation set is the union of the equation sets of the respective programs. Its inputs are the inputs of either program not appearing as outputs of the other ($I = (I_1 \cup I_2) \setminus (O_1 \cup O_2)$), and vice versa for its outputs ($O = (O_1 \cup O_2) \setminus (I_1 \cup I_2)$). In particular, certain specific types of composition shall be referred to as follows:

- *Disjoint composition*, if $O_2 \cap I_1 = O_1 \cap I_2 = \emptyset$.
- *Composition without feedback*, if $O_2 \cap I_1 = \emptyset$ or $O_1 \cap I_2 = \emptyset$.
- *Fully connected composition*, if $O_2 \cap I_1 = \emptyset$ and $O_1 = I_2$, or conversely $O_1 \cap I_2 = \emptyset$ and $O_2 = I_1$.

Another important operation is that of *adding a feedback loop* to a program P by connecting an output y to an input x written $P[y \to x]$, provided that y does not depend in any way on x, that is, $(y, x) \notin dep(P)$. Adding a feedback loop can also be defined in terms of composition of the original program with the Lustre program $P' = \langle \{x, y\}, \{y\}, \{x\}, \{x = y\} \rangle$, as follows:

$$P[y \to x] \stackrel{\text{df}}{=} P \mid P'$$

Assuming the existence of an ordering on the program's variables, given a Lustre program P, and a vector i that assigns a stream to each of the program's input variables, $P(i)$ denotes the vector o of output streams corresponding to the output variables of P as computed by the semantics of Lustre [2].

Our goal is therefore that of identifying Lustre programs P such that upon slowing down their inputs i according to a latency function λ, will result in P still being well behaved. In the next section we will identify different forms of robustness of $P(i_\lambda)$ with respect to the unslowed behaviour $P(i)$.

Boolean Lustre programs can also be compiled into automata spanning over the state space they cover [8]. This can be defined for Lustre programs using fby (instead of delays) as follows:

Definition 1: (Lustre Automaton). Let P be a Boolean Lustre program with n input variables, m output variables, and k fby equations of the form $y = x_1$ *fby* x_2. Then, this program can be compiled into an automaton $A = \langle S, s_{init}, \tau, \delta \rangle$, where S is its set of states, s_{init} is its initial state, $\tau : \mathbb{B}^n \times S \to S$ is its transition function and $\delta : \mathbb{B}^n \times S \to \mathbb{B}^m$ is its output function. The automaton, processes the input vector provided to the program one tuple at a time. During each instant, it uses its current input tuple and its present state to (i) move to a new state under the guidance of its transition function τ and (ii) output an output tuple as defined by its output function δ, which represents the values of the program's output variables at that particular time instant. The program P can be converted into automaton A using the following procedure.

External Initialisation: A program is said to be initialised externally if in at least one of its fby statements x_1 *fby* x_2, the initial variable x_1 depends on one of the program's input variables.

States: Each fby statement x_1 *fby* x_2 corresponds to a memory element in the program, whose value is determined by the variable x_1 at the first instant and by the variable x_2 at all further instants. Since each such memory can either be true or false, we create 2^k states, with each state representing one possible configuration of the program's memories. If the program is initialised externally, we also add a special initial state $init$ to the set of states.

Initial State: If the program is initialised externally, the initial state is $init$. Otherwise, the initial state is the state corresponding to the configuration obtained by evaluating the variables of the form x_1 within the program's fby statements.

Transition Function: With n input variables, there are 2^n possible input tuples. Each state therefore has 2^n transitions, with each transition labelled with the associated input tuple. Given a state $s \neq init$ and input tuple a, the next state $\tau(a, s)$ is computed as follows (i) assign the configuration represented by present state s to the respective variables of the form x_2 occurring on the right hand side of *fby* statements, (ii) assign the input values represented by tuple a to the respective input variables and (iii) simulate the Lustre program, using the defining equations of the variables of the form x_2 to determine the configuration of the memories at the next time instant, allowing the selection of the appropriate next state. The initial state $init$, if present, also has 2^n transitions. The next states are determined as follows (i) assign the input values represented by tuple a to the respective input variables, (ii) use the defining equations of variables of the form x_1 to compute the value of the initialisation variables and (iii) simulate the Lustre program using the defining equations of the variables of the form x_2, which determine the next state. Again, these values determine the configuration of the memories at the next time instant and allow the selection of the appropriate next state.

Output Function: Each transition is associated with an m-tuple, which represents the values of the output variables when the automaton finds itself in a certain state and processes a certain input tuple. The procedure for obtaining the output tuple is similar to that for obtaining the next state, except that the output tuple is constructed by simulating the program and considering the values of the output variables.

Figure 1 shows the automaton that would be obtained by applying the above procedure to the toggle switch program TOGGLE. The two states represent the two possible configurations that the memory corresponding to the program's only *fby* equation can be in. Meanwhile, for each transition, the value on the left shows the value of the toggle input variable that causes the transition, and the value on the right shows the output value computed by the program. We shall return to this representation of the TOGGLE program at a later stage.

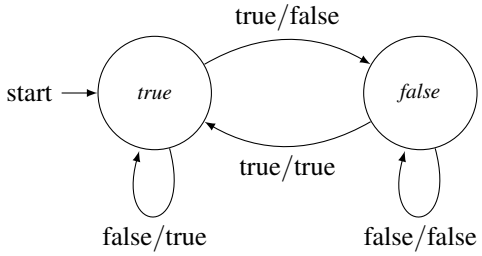

Figure 1. Automaton obtained from toggle switch program

We now consider a number of different forms of program robustness to slow input.

III. SLOWDOWN ROBUSTNESS

Whether a program behaves in an acceptable way depends on the scenario it is operating in. In this section, the four robustness properties of *stretch robustness*, *stutter robustness*, *fast-enough robustness* and *immediate-at-first robustness* are introduced, characterising desirable behaviour under different circumstances.

A. Stretch Robustness

Stretch robustness (STR) specifies the fact that if the input of a program slows down by some amount, then the output of a program should slow down by the same amount. This property can be formalised by requiring that whenever a latency function λ is applied to a program's input, the program will respond by applying the same latency to its output.

Definition 2: (Stretch Robustness). A program P is said to be *stretch robust with respect to a latency function* λ, if for any input vector i: $P(i_\lambda) = P(i)_\lambda$. P is simply said to be *stretch robust* if it is stretch robust with respect to all latency functions.

The figure below shows the relationship between a slow input vector i_λ and the required program output $P(i_\lambda)$:

i_λ	$\underbrace{i(0),\dots i(0)}_{\lambda(0)+1}$,	$\underbrace{i(1),\dots i(1)}_{\lambda(1)+1}$,	$\dots$	$\underbrace{i(n),\dots i(n)}_{\lambda(n)+1}$	$\dots$
$P(i_\lambda)$	$\underbrace{o(0),\dots o(0)}_{\lambda(0)+1}$,	$\underbrace{o(1),\dots o(1)}_{\lambda(1)+1}$,	$\dots$	$\underbrace{o(n),\dots o(n)}_{\lambda(n)+1}$	$\dots$

One immediate consequence of this property is that additional repetition of the program's input does not cause the program to change its output. Stretch robustness is thus useful in situations where one requires the program not to change its output when faced with additional latency. Stretch robustness is a very strong property, which can be relaxed in a number of ways to obtain weaker criteria that may be sufficient in certain circumstances. We shall now consider these criteria, leaving the proof that these are indeed relaxations of stretch robustness to Theorem 6.

B. Stutter Robustness

Stutter robustness (STU) requires that if the input of a program slows down by some amount, the output of the program should also slow down, but possibly at a different rate. This will be modelled by requiring that whenever a latency function λ is applied to a program's input, the program will respond by applying *some* latency function λ' to its output. Unlike stretch robustness, λ and λ' need not be equal:

Definition 3: (Stutter Robustness). A program P is *stutter robust with respect to a latency function* λ if there exists a latency function λ' such that for every input vector i: $P(i_\lambda) = P(i)_{\lambda'}$. P is said to be *stutter robust* if it is stutter robust with respect to any latency function.

The relationship between a slow vector of inputs i_λ and the required program output $P(i_\lambda)$ is shown below:

i_λ	$\underbrace{i(0),\dots i(0)}_{\lambda(0)+1}$,	$\underbrace{i(1),\dots i(1)}_{\lambda(1)+1}$,	$\dots$	$\underbrace{i(n),\dots i(n)}_{\lambda(n)+1}$	$\dots$
$P(i_\lambda)$	$\underbrace{o(0),\dots o(0)}_{\lambda'(0)+1}$,	$\underbrace{o(1),\dots o(1)}_{\lambda'(1)+1}$,	$\dots$	$\underbrace{o(n),\dots o(n)}_{\lambda'(n)+1}$	$\dots$

Thus for a stutter robust program, the output under slow input can be obtained from the original output by adding *any* number of repetitions to the values appearing in the original output, without adding any other artifacts nor removing any values. This means that stutter robustness is useful in situations where one needs to ensure that the output under slow input has the same structure as the original output, but one is able to tolerate additional repetition in the slow output.

C. Fast-Enough and Immediate-at-First Robustness

In stretch robustness, the value of the outputs remains equal to the original value in the unslowed system. In fast-enough robustness (FE) this constraint is relaxed by requiring only that the program converge to the original output before the slowed down input ends. Formally, we shall say that a program is fast-enough robust if, when we apply a latency function λ to the program's input, the slow output has the property that its value at the end of each block of repetitions (at points of the form End_t^λ) is equal to the value taken by the original output at the points t (i.e., those points that were expanded into the blocks of repetitions).

Definition 4: (Fast-Enough Robustness). A program P is *fast-enough robust with respect to a latency function* λ if for any input vector i:

$$\forall t : \mathbb{T} \cdot P(i_\lambda)(End_t^\lambda) = P(i)(t)$$

P is said to be *fast-enough robust* if it is fast-enough robust with respect to any latency function.

Fast-enough robustness will be primarily of interest for particular latency functions. As a property, it can be visualised as follows (using ? to indicate don't-care values):

i_λ	$\underbrace{i(0),\ldots i(0)}_{\lambda(0)+1}$,	$\underbrace{i(1),\ldots i(1)}_{\lambda(1)+1}$,	$\ldots$	$\underbrace{i(n),\ldots i(n)}_{\lambda(n)+1}$	$\ldots$
$P(i_\lambda)$	$\underbrace{?,\ldots ?,o(0)}_{\lambda(0)+1}$,	$\underbrace{?,\ldots ?,o(1)}_{\lambda(1)+1}$,	$\ldots$	$\underbrace{?,\ldots ?,o(n)}_{\lambda(n)+1}$	$\ldots$

This robustness property is useful in scenarios in which one can tolerate the fact that additional latency on the input might produce undesirable intermediate results as long as the original value is produced by the end of the latency period.

The dual of fast-enough robustness is *immediate-at-first robustness* (IAF) — instead of constraining the slow input to converge to the original value before a block of repetitions ends, it requires it to produce the original value as soon as a block of repetitions starts, leaving it free to assume any value until that block of repetition ends.

Definition 5: (Immediate-At-First Robustness). A program P is said to be *immediate-at-first robust with respect to latency function* λ if for any input vector i:

$$\forall t : \mathbb{T} \cdot P(i_\lambda)(Start_t^\lambda) = P(i)(t)$$

P is said to be *immediate-at-first robust* if it satisfies the above constraint with respect to any latency function.

Immediate-at-first robustness can be visualised as follows:

i_λ	$\underbrace{i(0),\ldots i(0)}_{\lambda(0)+1}$,	$\underbrace{i(1),\ldots i(1)}_{\lambda(1)+1}$,	$\ldots$	$\underbrace{i(n),\ldots i(n)}_{\lambda(n)+1}$	$\ldots$
$P(i_\lambda)$	$\underbrace{o(0),?,\ldots ?}_{\lambda(0)+1}$,	$\underbrace{o(1),?,\ldots ?}_{\lambda(1)+1}$,	$\ldots$	$\underbrace{o(n),?,\ldots ?}_{\lambda(n)+1}$	$\ldots$

This robustness property is useful in scenarios in which one requires the program to react immediately as soon as the latency on a previous input value wears off, but in which further repetition of the input can be safely ignored by outputting any result.

We now prove that the stutter robustness, immediate-at-first robustness and fast-enough robustness properties are relaxations of the stretch robustness property. To achieve this, it will be convenient to treat each property as the set of all Lustre programs that satisfy it.

Theorem 6: (Stutter robustness, immediate-at-first robustness and fast-enough robustness are relaxations of stretch robustness) $STR \subseteq STU, IAF, FE$.

Proof: We first show that $STR \subseteq STU$. Let P be a stretch robust program. We shall prove that it is also stutter robust. For program to be stutter robust, it must respond to a latency function λ on its input by applying some latency function λ' to its output. Since P is stretch robust, when a latency function λ is applied to its input, it will respond by applying an identical λ to its output. Hence P also satisfies stutter robustness.

To prove the other two inclusions, we first note that when P is stretch robust, its output under slow input, $P(i_\lambda)$ is organised into successive blocks, with the t^{th} block lying between $Start_t^\lambda$ and End_t^λ. In addition, the symbols within the t^{th} block are all of the form $P(i)(t)$, that is, equivalent to the t^{th} output of P under the unslowed input i. From this, it follows directly that $P(i_\lambda)$ at the points of the form $Start_t^\lambda$ is equal to $P(i)(t)$ as required by immediate-at-first robustness. It also follows that $P(i_\lambda)$ at the points of the form End_t^λ is equal to $P(i)(t)$, thus satisfying fast-enough robustness. Hence the directions $STR \subseteq IAF$ and $STR \subseteq FE$ are also proven. □

We now proceed to consider algorithmic means for detecting whether a Lustre program satisfies a robustness property.

IV. Detecting Robustness: Static Analysis

The first approach to checking whether a Boolean Lustre program satisfies a robustness property is based on a static analysis of the structure of the Lustre program. The analysis is based on two kinds of result: (i) Theorem 7, which identifies those primitive programs that satisfy certain robustness properties and; (ii) Theorems 8, 9, 10 and 11,which identify those robustness properties that are preserved upon composition of two robust programs. We now proceed to state these theorems.

Theorem 7: Primitive Lustre programs all come with a level of guaranteed robustness: (i) instantaneous programs are robust under all four forms; (ii) delay and followed-by programs are robust under stutter and immediate-at-first robustness; and (iii) primitive initialisation programs are immediate-at-first robust.

Proof: (i) Instantaneous programs apply a pointwise operator to their input streams to obtain their output streams. Thus, the same input tuple always causes the same output tuple. Repetition of inputs through latency will therefore cause repetition of outputs, which makes the program stretch robust. (ii) The output of delay and fby programs has an additional initial value with respect to the input stream. Slowing the input stream down by a latency function, causes the program to attach this value to the slow stream. The output under slow input can therefore be obtained from the original input through a latency function, which does not repeat the attached element, and which repeats all subsequent elements accordingly. These programs are therefore stutter robust. The programs are also immediate-at-first robust as can be inferred from the figure below, which shows how the values of the original output (first row) are associated to the corresponding blocks of the output under latency (second row). It is clear that the value at the beginning of each block is equal to the corresponding value in the original output.

$P(i)$	Nil	$x_1(0)$	$x_1(1)$
$P(i_\lambda)$	$Nil, \underbrace{x_1(0)\ldots x_1(0)}_{\lambda(0)}$	$x_1(0), \underbrace{x_1(1)\ldots x_1(1)}_{\lambda(1)}$	$x_1(1), \underbrace{x_1(2)\ldots x_1(2)}_{\lambda(2)}$

(iii) Initialisation programs take the first value of stream x_1, and attach to it the stream x_2 from its second value onwards. The figure below shows how the blocks of output under slow input, relate to the original output, illustrating the fact that the value at the beginning of each block is equal to the corresponding value in the original output.

$P(i)$	$x_1(0)$	$x_2(1)$	$x_2(2)$
$P(i_\lambda)$	$x_1(0), \underbrace{x_2(0)...x_2(0)}_{\lambda(0)}$	$\underbrace{x_2(1)...x_2(1)}_{\lambda(1)+1}$	$\underbrace{x_2(2)...x_2(2)}_{\lambda(2)+1}$

□

We shall now consider the effect of composing robust programs.

Theorem 8: The composition without feedback of stretch robust programs is stretch robust.

Proof: We consider two stretch robust programs P_1 and P_2 that are composed without feedback, as shown in Figure 2. As a remark we note that P_2 obtains some input from P_1 (the vector j) and some input from an external source (the vector i_2), which can be collectively represented by the vector $j \cup i_2$. On the other hand, the output of the complete system, consists of some output from P_1 (the vector o_1) and some output from P_2 (the vector o_2), which is collectively represented by the vector $o_1 \cup o_2$.

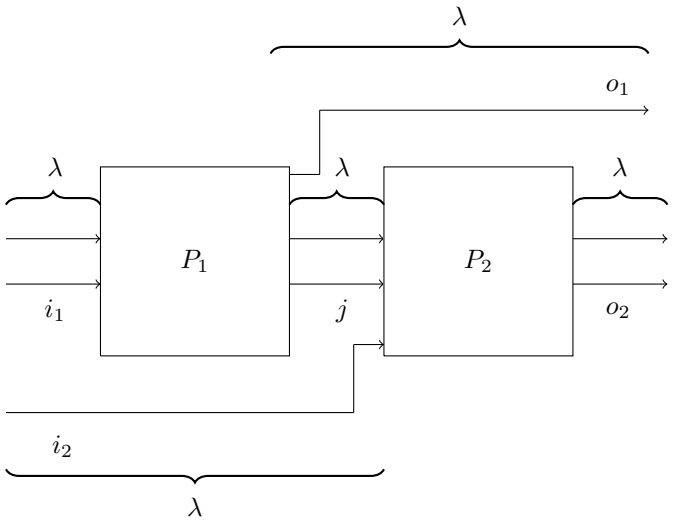

Figure 2. Composition without feedback preserves STR

Since the composed program reacts to input $i_1 \cup i_2$ with output $o_1 \cup o_2$, we aim to show that given slow input $(i_1 \cup i_2)_\lambda$, the program will react with the output $(o_1 \cup o_2)_\lambda$.

To see why this is true, we observe that P_1 is stretch robust, and thus will react to the input $i_{1\lambda}$ with the output vector $(o_1 \cup j)_\lambda$. Since latency functions distribute over the union of vectors, we can conclude that P_1 outputs two sets of streams, corresponding to the two output vectors $o_{1\lambda}$ and j_λ. P_2 will thus receive as input the vectors j_λ and $i_{2\lambda}$, which we can represent by a single input vector $(j_\lambda \cup i_{2\lambda})$ or equivalently, $(j \cup i_2)_\lambda$. Being stretch robust itself, it must then output the slow output vector $o_{2\lambda}$. Combining the output vectors of the system $o_{1\lambda}$ and $o_{2\lambda}$ we obtain the vector $(o_{1_\lambda} \cup o_{2_\lambda})$, which is equivalent to $(o_1 \cup o_2)_\lambda$ as expected. □

Theorem 9: Adding a feedback loop to a stretch robust program gives a stretch robust program.

Proof: Let P be a stretch robust program, and consider adding a feedback loop between one of its outputs y and one of its inputs x. Our definition of adding a feedback loop also requires y not to depend in any way on x, that is, $(y, x) \notin dep(P)$. This situation is shown in Figure 3, where the triangle shows that y can only be a function of a set of inputs $I^* \subseteq I$ that excludes x.

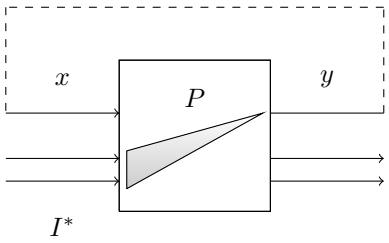

Figure 3. Adding a feedback loop to a program

To prove that this program remains stretch robust, we proceed by constructing an equivalent program that 'unwinds' the feedback loop, and which we can prove to be stretch robust by the application of Theorem 8. The new program will contain two copies of P, which we call P_1 and P_2. These are connected as shown in Figure 4.

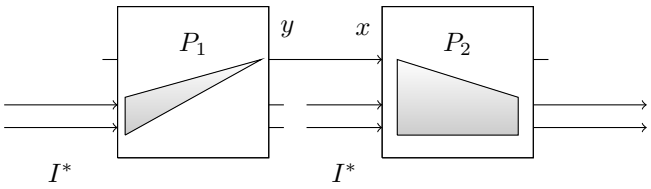

Figure 4. Unwinding the feedback loop

We shall use P_1 to compute the value of y from I^*, discarding all outputs but y. Following this, P_2 can be used to recompute all of the outputs, by using the inputs I^*, and by feeding the output y from P_1 to the input x in P_2 in order to simulate the feedback equation $x = y$. Since the feedback loop plays no further role in computing output of the original program, we simply discard y from the output of P_2 and keep the rest of the outputs. Note that in Figure 4, the parallelogram inside P_2 is meant to indicate that the outputs of P_2 that are relevant to the final computation can now depend on all of its inputs).

We now notice that i) P_1 and P_2 are copies of P, ii) P is stretch robust and iii) the construction reduces to a composition without feedback of stretch robust programs. By Theorem 8, the composition without feedback of stretch robust programs is stretch robust. Hence, the equivalent program with the feedback loop is also stretch robust, as required. □

Theorem 10: The fully connected composition of two stutter robust programs is stutter robust.

Proof: We consider two arbitrary stutter robust programs P_1 and P_2, and show that their fully connected composition is

also stutter robust. Since P_1 and P_2 are being composed in a fully connected way, every output of P_1 is connected to an input of P_2, and there are no feedback connections. This is shown in Figure 5.

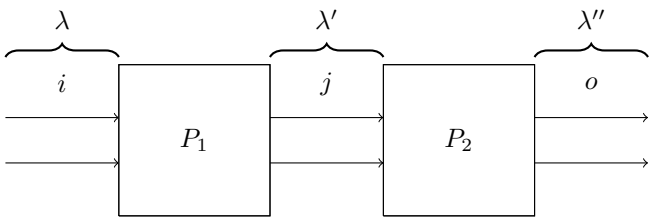

Figure 5. Fully connected composition preserves STU

Now suppose that if we pass a vector i to P_1, the program responds by outputting vector j. Also suppose that when P_2 receives vector j it outputs vector o in response. We need to show that if a latency function λ is applied to the composite program's input vector i, the composite program applies some latency function to its output vector o. Since P_1 is stutter robust, applying λ to the input vector i will make P_1 apply some latency function λ' to its output j. Hence, P_2 receives the vector j slowed down by λ' as input. Since P_2 is also stutter robust it will apply some other latency function λ'' to its output o. Thus, applying a latency function to the input of the composed program, causes the composed program to slow its output by some latency function, proving that stutter robustness is preserved by fully connected composition. □

Theorem 11: The disjoint composition of two immediate-at-first robust programs is immediate-at-first-robust.

Proof: We consider two immediate-at-first robust programs P_1 and P_2 in disjoint composition, as shown in Figure 6. We note that P_1 computes output vector $P_1(i_1)$ given input vector i_1, while P_2 computes output vector $P_2(i_2)$ when given input vector i_2. Thus, if the composed program is given the input vector $(i_1 \cup i_2)$, it will react with an output vector $(P_1(i_1) \cup P_2(i_2))$.

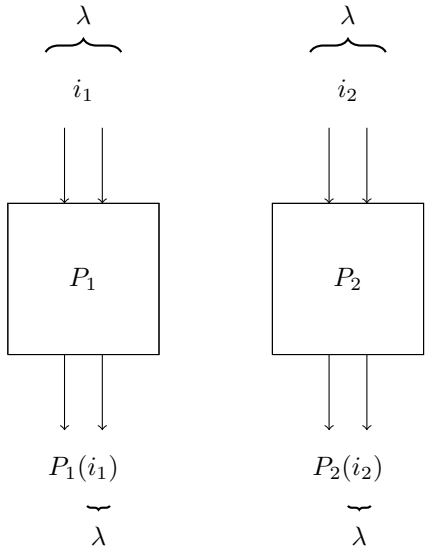

Figure 6. Disjoint composition preserves IAF

We now note that both P_1 and P_2 are immediate-at-first robust, and that under slow input, will generate two disjoint output vectors with the following properties. The output vector $P_1(i_{1\lambda})$ of P_1 will have the property that for every stream s:

$$s(Start_t^{\lambda}) = P_1(i_1)(s)(t) \quad (1)$$

while the output vector $P_2(i_{2\lambda})$ of P_2 will have the following property for every one of its streams s.

$$s(Start_t^{\lambda}) = P_2(i_2)(s)(t) \quad (2)$$

On the other hand, to prove that the composite program is immediate-at-first-robust, we would need to show that each of the streams s in its output vector $(P_1(i_{1\lambda}) \cup P_2(i_{2\lambda}))$ obeys the following property:

$$s(Start_t^{\lambda}) = (P_1(i_1) \cup P_2(i_2))(s)(t)$$

However, we note that we are always able to determine whether a stream s in $(P_1(i_{1\lambda}) \cup P_2(i_{2\lambda}))$ originates from P_1 or P_2. Hence, it is sufficient to ensure that such a stream s always satisfies the property

$$s(Start_t^{\lambda}) = P_n(i_n)(s)(t) \quad (3)$$

where n is the index of the program generating s. By equation (1), we know that property (3) holds when s is generated by P_1 (when n=1), while equation (2) ensures that property (3) holds when s is generated by P_2 (when n=2). The theorem is thus proved. □

We now consider a method that analyses the behaviour of the particular program under examination, rather than its structure.

V. Detecting Robustness: Dynamic Analysis

The theorems in Section IV allow us to conclude robustness of composed programs in a syntactically compositional manner. In this section, we give richer, although more expensive, semantic analysis techniques for Boolean Lustre programs allowing for dynamic robustness analysis of their behaviour. Through the use of symbolic methods, such as with Binary Decision Diagrams (BDDs), the analysis can be applied either on whole programs or to subprograms. In the latter case, the static analysis theorems can then be used to obtain results about the composition of the subprograms.

The techniques we shall discuss rely on identifying conditions on the Lustre automaton that are sufficient to guarantee that certain robustness properties are satisfied by that program. Two types of conditions are defined: (i) latency independent conditions, which check whether a robustness property holds in general, and (ii) latency dependent conditions, which check whether a property holds when some particular latency function is applied to the program's input.

The conditions identified can be checked using either an exhaustive analysis of the automaton's state space, or preferably using a symbolic representation of the automaton such as BDDs to ensure that the approach scales up to larger systems.

A. Latency Independent Conditions

We start by identifying properties that guarantee slowdown robustness for any latency function. The first condition we shall consider checks whether being in different states can cause the program's output to change. If this is not the case, the program is stretch robust.

Theorem 12: (Condition 1 — Stretch Robustness). Programs satisfying the following condition are stretch robust:

$$\forall a, s, s' \ \cdot \ \delta(a, s) = \delta(a, s')$$

Proof: Under such a condition, a particular input tuple always generates the same output tuple, independently of the state the automaton find itself in. Thus, any repetition of an input tuple caused by a latency function causes a repetition of the corresponding output tuple. This is sufficient to ensure stretch robustness. □

We now consider a condition that stops a program from being stretch robust. Condition 2 states that if we can find a state s, which under some tuple a moves to s', such that s and s' yield different outputs under a, then the program must fail to be stretch robust.

Theorem 13: (Condition 2 — Failure of Stretch Robustness) Programs satisfying the following condition are not stretch robust:

$$\begin{aligned} \exists\ a, b, b', s, s' \ \cdot \\ \quad \tau(a, s) = s' \wedge \delta(a, s) = b \\ \quad \delta(a, s') = b' \wedge b \neq b' \end{aligned}$$

Proof: We show that the existence of reachable states s and s' described by the theorem is sufficient to find an input vector and a latency function λ such that the program fails to be stretch robust.

We first build the input vector as follows. Starting from the initial state of the automaton, we follow any path of some length n leading to state s. This path gives us the first n tuples of the input vector i_1. At state s, we then follow the transition for tuple a. The rest of the input vector can then be selected arbitrarily, yielding $i_1\ a\ i_2$.

Given the input vector $i_1\ a\ i_2$, the automaton will first process the tuples in i_1 and output the first n tuples of the output vector o_1. Since the automaton is now in state s, it will output tuple b, followed by some sequence of tuples o_2.

To prove the theorem, we select a latency function that adds one repetition at time instant $n + 1$, and 0 repetitions elsewhere. Applying this latency function to the input vector thus gives $i_1\ a\ a\ i_2$.

On this slow vector the automaton will first output $o_1\ b$, as before, but being now in state s', will output some tuple $b' \neq b$, followed by the rest of the output vector o_3. It is clear that $o_1\ b\ b'\ o_3$ is different from $(o_1\ b\ o_2)_\lambda = (o_1\ b\ b\ o_2)$. Thus the program is not stretch robust, as required. □

Condition 1 indicates that programs whose states do not affect program output are stretch robust. The states in such programs are effectively redundant, in the sense that automata corresponding to these programs can be minimised (say, using the *partitioning minimisation procedure* [9]) into equivalent automata having just one state. Through condition 2, we also know that programs having two successive states yielding different output tuples under the same input tuple cannot be stretch robust. This raises the question of whether there are any non-trivial stateful programs that satisfy stretch robustness.

This question can in fact be given a positive answer by identifying a condition that lies between these two extremes. The intuition behind condition 3 is as follows. Given any state s, which under input a yields output b, we will allow it to transition to state s' if i) s' under a also yields b, thus escaping condition 2, and ii) s' self-loops under a. The effect of the second requirement is that s' becomes a 'sink' for repetitions of a, and preserves the output b experienced under s, guaranteeing stretch robustness. Note that as long as the requirements for transitions between states are satisfied, there can very well be states with different output behaviours. This is something that is not covered by condition 1.

Theorem 14: (Condition 3 — Stretch Robustness) Programs satisfying the following condition are stretch robust:

$$\begin{aligned} \forall a, b, s, s' \ \cdot \\ \quad (\tau(a, s) = s' \wedge \delta(a, s) = b) \\ \implies (\tau(a, s') = s' \wedge \delta(a, s') = b) \end{aligned}$$

Proof: When processing its input vector i, the automaton will use the current input $i(t)$ and state $s(t)$, to compute the output $o(t)$ and next state $s(t + 1)$. When input i has latency λ, the program receives consecutive blocks of constant inputs, with the n^{th} block consisting of tuples of the form $i(n)$. To satisfy stretch robustness, the program must apply λ to its output, thus yielding blocks of constant outputs such that the n^{th} block consisting of tuples of the form $o(n)$.

We observe that if the automaton finds itself in state $s(n)$ at the beginning of input block n, the condition guarantees that (i) at the first time instant in the block the automaton outputs $o(n)$ and moves to state $s(n + 1)$; (ii) during the rest of the time instants in the block it will again output $o(n)$ and self loop to state $s(n+1)$ and (iii) it will find itself in state $s(n+1)$ at the start of the $(n + 1)^{th}$ input block.

Noting that in the case of block 0, the automaton starts in the initial state $s(0)$, provides the base case for an inductive argument which guarantees that the automaton finds itself in state $s(n)$ at the start of the n^{th} block, causing the output to be $o(n)$ throughout that block as required. □

Although condition 3 shows that there are non-trivial stateful stretch robust programs, it also sets a strong requirement for this to be the case. We now investigate conditions for weaker robustness properties, starting from stutter robustness.

Under stutter robustness, slowing a program's input by a latency function λ, causes the program to slow its output by a latency function λ'. In practice, this means that the output under slow input can be obtained through the repetition of the original output tuples only. We show that if the automaton has a certain feature, then this property cannot hold.

Theorem 15: (Condition 4 — Failure Of Stutter Robustness). Programs satisfying the following condition are not stutter robust:

$$\begin{aligned}\exists a, b, s, s', j, k, l \cdot \\ \delta(a, s) = j \wedge \tau(a, s) = s' \wedge \\ \delta(a, s') = k \wedge k \neq j \wedge \\ \delta(b, s') = l \wedge b \neq a \wedge l \neq k\end{aligned}$$

Proof: Condition 4 looks for the presence of reachable states s and s' having the following properties: (i) under input tuple a, state s outputs tuple j and passes to state s'; (ii) under input tuple a, state s' outputs $k \neq j$ and (iii) under input tuple $b \neq a$, state s' outputs tuple $l \neq k$.

We now show that if this structure is present in the automaton, there will always be some input vector and some latency function that breaks the stutter robustness property. We first construct the input vector as follows. Choose a path from the start state s_{init} to the state s. By following this path of n transitions, we obtain the first n tuples of the input vector, which we denote by i_1. We also obtain the first n tuples of the output vector, denoted by o_1. To this initial segment of the input vector, one appends the input tuples $a\ b$, which causes the resulting output vector to be augmented by the output tuples $j\ l$. The rest of the input vector i_2 can be chosen arbitrarily.

We now choose a latency function, which when applied to the input vector $i_1\ a\ b\ i_2$ above, breaks the property. The chosen latency function will insert 1 repetition for the input tuple at time instant $n+1$, and 0 repetitions elsewhere. Applying this latency function to the input vector chosen earlier will yield $i_1\ a\ a\ b\ i_2$. Through the presence of the regularity identified in the theorem, the resulting output will be the initial segment of the output vector o_1 followed by the output tuples $j\ k$, which means that with respect to the original output an l tuple has been deleted. This fact alone makes it impossible to derive the output under slow input from the original output through the addition of repetitions only. □

Finally, we can also identify a sufficient condition for immediate-at-first robustness, which we can obtain by relaxing condition 3. We shall still insist that if a state s transitions to s' under some input, then it must loop in s' under that input. However, we will not require input in s and s' to yield the same output. As the proof shows this is enough to guarantee that the program is immediate-at-first robust.

Theorem 16: (Condition 5 — Immediate-At-First Robustness). Programs satisfying the following condition are immediate-at-first-robust:

$$\forall a, s, s' \cdot (\tau(a, s) = s') \implies (\tau(a, s') = s')$$

Proof: When processing an input vector i, the automaton uses the current input $i(t)$ and state $s(t)$, to compute the output $o(t)$ and next state $s(t+1)$. When input i has latency λ, the program receives consecutive blocks of constant inputs, with the n^{th} block consisting of tuples of the form $i(n)$. For the program to be immediate-at-first robust, the output at the beginning of the n^{th} block must have the form $o(n)$.

We observe that if the automaton finds itself in state $s(n)$ at the beginning of block n, the condition guarantees that (i) at the first time instant in the block the automaton moves to state $s(n+1)$; (ii) it stays in state $s(n+1)$ for the remainder of the block and (iii) the $(n+1)^{th}$ block starts in state $s(n+1)$. Noting that in block 0, the automaton starts in the initial state $s(0)$, provides the base case for an inductive argument which guarantees that the automaton finds itself in state $s(n)$ at the beginning of the n^{th} block, causing the output to be $o(n)$ as required. □

B. Latency Dependent Conditions

So far, we tried to identify programs that are robust under an input slowed down by an unknown latency. If one knows that the inputs of a program are going to slow down by some uniform latency function $\lambda(t) = c$, where c is a constant, it is possible to check whether the program is robust for that particular scenario using the following weakened conditions.

Condition 6 requires that for any state s, the state reached by the automaton after the occurrence of a specific input tuple, $\tau(a, s)$, is the same state reached after the occurrence of $c+1$ such input tuples, which we denote by $\tau^{c+1}(a, s)$.

Theorem 17: (Condition 6 — Immediate-At-First-Robustness). Programs satisfying the following condition for some positive natural number $c \geq 2$, are immediate-at-first robust for latency functions of the form $\lambda(t) = c$:

$$\forall a, s \cdot \tau(a, s) = \tau^{c+1}(a, s)$$

Proof: When processing an input vector i, the automaton uses the current input $i(t)$ and state $s(t)$, to compute the output $o(t)$ and next state $s(t+1)$. When input i has latency λ, the program receives consecutive blocks of constant inputs of size $c+1$, with the n^{th} block consisting of tuples of the form $i(n)$. For the program to be immediate-at-first robust, the output at the beginning of the n^{th} block must have the form $o(n)$.

Suppose that at the beginning of the n^{th} block the automaton finds itself in state $s(n)$. Then at the beginning of the $(n+1)^{th}$ block it is in state $s(n+1)$ on account of (i) at the first time instant in the n^{th} block the automaton moves to $s(n+1)$ and (ii) the condition guarantees that after $c+1$ steps of the same input the automaton will return to $s(n+1)$. Noting that in block 0, the automaton starts in the initial state $s(0)$, provides the base case for an inductive argument which guarantees that the automaton finds itself in state $s(n)$ at the beginning of the n^{th} block, causing the output to be $o(n)$ as required. □

The final condition that will be considered requires that if an automaton is in state s, it will return to the same state s after c repetitions of the input.

Theorem 18: (Condition 7 — Immediate-At-First and Fast-Enough Robustness). Programs satisfying the following condition for some positive natural number $c \geq 2$, is immediate-at-first robust and fast-enough robust for latency functions of the form $\lambda(t) = c$:

$$\forall a, s \cdot \tau^{c}(a, s) = s$$

Proof: When processing an input vector i, the automaton uses the current input $i(t)$ and state $s(t)$, to compute the output $o(t)$ and next state $s(t+1)$. When input i has latency λ, the program receives consecutive blocks of constant inputs of size $c+1$,

with the n^{th} block consisting of tuples of the form $i(n)$. For the program to be immediate-at-first robust, the output at the beginning of the n^{th} block must have the form $o(n)$. Similarly, for a program to be fast-enough robust, the output at the end of the n^{th} block must have the form $o(n)$.

Suppose that at the beginning of the n^{th} block the automaton finds itself in state $s(n)$. Then at the end of the n^{th} block it is in state $s(n)$ on account of the fact that the automaton returns to its original state after c transitions of the same input. This state also combines with input $i(n)$ to ensure passage to state $s(n+1)$ at beginning of the $(n+1)^{th}$ block. Noting that in block 0, the automaton starts in the initial state $s(0)$, provides the base case for an inductive argument which guarantees that the automaton always finds itself in state $s(n)$ at the end of the n^{th} block, causing the output to be $o(n)$ as required for fast-enough robustness, and in state $s(n+1)$ at the beginning of the $(n+1)^{th}$ block guaranteeing that the output is $o(n+1)$ as required by immediate-at-first robustness. □

We now apply the static and dynamic analysis techniques discussed earlier via a case study.

VI. Case Study

The static and dynamic analysis theorems were applied to six Boolean Lustre programs to examine whether these are strong enough to deduce slowdown robustness. For comparison purposes, a manual analysis of these programs was also performed in order to discover which robustness properties each program satisfies or fails to satisfy. The programs under consideration, with the actual properties satisfied by each are listed below:

- RCA, a (stateless) ripple carry adder that satisfies stretch robustness.
- RISE, a program that receives a Boolean stream and detects the presence of rising edges, and which satisfies immediate-at-first robustness.
- SWSR, a switch with a set and reset input, which satisfies stretch robustness.
- TOGGLE, a switch with a toggle input, which does not satisfy any property for every latency function.
- SISO, a serial in serial out register, which satisfies stutter robustness.
- PIPO, a parallel in parallel out register, which satisfies stutter robustness and immediate-at-first robustness.

We shall now discuss the application of the static and dynamic analysis theorems to the programs in question. To illustrate how the static analysis theorems can be employed to reason about a program, we will consider their use to prove that the SISO register program is stutter robust.

Example 1: Since the SISO program has 4 equations, we first break it down into four separate primitive programs SISO_1, SISO_2, SISO_3 and SISO_4 as shown in Figure 7, where $\text{SISO}_j = \langle\{i_j, i_{j+1}\}, \{i_j\}, \{i_{j+1}\}, \{i_{j+1} = true\ fby\ i_j\}\rangle$

It is clear that each such program is an fby primitive program, and that these primitive programs can be composed through fully connected composition to obtain the program

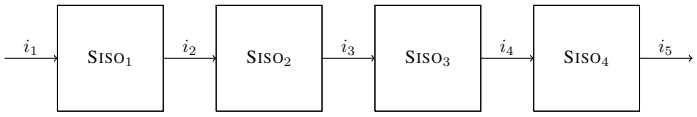

Figure 7. SISO program broken into primitive programs

SISO. This can be done by starting from SISO_1 and sequentially composing the programs SISO_2, SISO_3 and SISO_4. Since SISO can be built from stutter robust primitives and through stutter robustness preserving compositions, we can conclude that it is stutter robust.

Table I illustrates the results that can be obtained in a similar manner through the static analysis of the programs in question. An entry in the table indicates whether the corresponding program can be shown to satisfy a particular robustness property or not through this technique. Within an entry, a ✓ symbol indicates that the program was found to satisfy the property. In addition, a ? symbol indicates that the static analysis yielded an inconclusive result, while a − symbol indicates that a test was unnecessary since the program was found to satisfy the stronger property of stretch robustness.

TABLE I. RESULTS OBTAINED THROUGH STATIC ANALYSIS

Property/Program	RCA	RISE	SWSR	TOGGLE	SISO	PIPO
STR	✓	?	?	?	?	?
STU	-	?	?	?	✓	?
IAF	-	?	?	?	?	✓

As one can see, the static analysis reveals that the ripple carry adder is stretch robust, that the SISO register is stutter robust and that the PIPO register is immediate-at-first robust. Static analysis thus yields results when the programs have a simple structure in terms of the interconnections between the component primitive programs.

We now illustrate how dynamic analysis can be applied by means of another example. We shall show that the Toggle Switch program TOGGLE is both immediate-at-first robust as well as fast-enough robust for the latency function $\lambda(t) = 2$.

Example 2: Starting from the TOGGLE program, we first obtain the automaton representation of the program by using the construction outlined in Definition 1. This yields the automaton depicted earlier in Figure 1. By observing the structure of the automaton, we note that from any state, taking 2 transitions with the same input tuple returns the automaton to the same state. The program thus satisfies the properties in question through the use of Theorem 18.

Table II summarises the results obtained through the dynamic analysis of the programs under consideration. In addition to the earlier conventions, an × symbol indicates that the program was found not to satisfy the property in question, while a ✓ symbol with subscript $c = 2$, indicates that the program has been proven to satisfy the property for the latency function $\lambda(t) = 2$ through the use of a latency dependent condition. In practice, BDD techniques were used to evaluate the conditions, and the evaluation was instantaneous for the programs in question.

TABLE II. RESULTS OBTAINED THROUGH DYNAMIC ANALYSIS

Property/Program	RCA	RISE	SWSR	TOGGLE	SISO	PIPO
STR	✓	×	✓	×	×	×
STU	-	×	-	×	?	?
IAF	-	✓	-	$\checkmark_{c=2}$	?	✓
FE	-	?	-	$\checkmark_{c=2}$	?	?

Dynamic analysis enlarges the scope of automatically derived robustness results to programs that have more complex structures. In particular, this approach gives the following results.

- *Stretch Robustness:* The ripple carry adder is shown to be stretch robust, a fact also discovered through the static analysis. On the other hand, the switch with set and reset is shown to be stretch robust through condition 3, while all other programs fail to be stretch robust by condition 2. The latter two results are an improvement over the analysis conducted in [1].
- *Stutter Robustness:* The rising edge and toggle switch programs are shown not to be stutter robust, something which is not possible to discover through the static analysis, since no conditions for the failure of a property are considered.
- *Immediate-At-First Robustness:* The PIPO register and the rising edge program are shown to be immediate-at-first robust. While the former fact was discovered by the static analysis, the latter one was not.
- *Latency Dependent Conditions:* The toggle switch program is shown to satisfy immediate-at-first and fast-enough robustness for the specific latency function $\lambda(t) = 2$.

While not all of the properties satisfied by the programs have been discovered through the automated analysis, the combination of static and dynamic analysis has revealed many details about the robustness of the programs in question. The conditions for stretch robustness have in particular been effective at classifying the programs considered. The number of programs proven to be immediate-at-first robust also indicates that the condition which detects it might be applicable for some interesting set of programs. On the other hand, the results obtained using the latency dependent conditions are encouraging as they indicate the possibility of satisfying a property under a particular slowdown scenario even though the program might not satisfy it in general. One can also note that the two approaches can be complementary; in particular while dynamic analysis allows discovery of programs that are not stutter robust, static analysis allows reasoning about those that satisfy this property.

VII. RELATED WORK

The discrete theory of slowdown considers the effect of slowing down all the input streams of a stream processing program by the same amount through the addition of stutter. There are various other models of slowdown that can be found in the literature. The theory of latency insensitive design [6] allows streams to slow down through the addition of explicit stall moves into those streams. In reaction to performing a stall move on an input stream, a program reacts by performing a procrastination effect, that is by inserting additional stall moves in its other streams to ensure that causality between the events of a program is preserved. A program is said to be patient if it knows how to perform a procrastination effect in response to any possible stall move. In other words, the program is always able to delay its operation in response to slow input without breaking. Patience is thus a form of robustness to delays in the process' streams, but which, unlike our properties, does not dictate the exact form that this robustness should take.

In the theory of polychronous processes [5], used to give a semantics to the synchronous language Signal [4], streams do not have to take a value at every time instant. Given a particular program behaviour, consisting of the input and output streams of a program, the operations of stretching and relaxation can be used to obtain a slower program behaviour. Stretching remaps the time instants at which the values occur on each stream, preserving the order of values in each stream, and the simultaneity of values between different streams. The stretching operation stretches all the streams by exactly the same amount and is similar to how a stretch robust program would behave when its inputs are slowed down. On the other hand, when relaxation slows a behaviour, it only guarantees that the order of values within each stream is preserved. The notion of relaxation that arises when all input streams are slowed down by one amount, and all output streams by another amount, is similar to how to a stutter robust program would behave under input slowdown. Signal guarantees that all its programs are stretch closed (a property analogous to stretch robustness), but this is only possible because no additional values are ever inserted as a result of slowing down a stream.

Reasoning about slowdown and speedup for continuous time behaviour has been investigated in [3]. The behaviour of a program can be slowed down by stretching these real-time signals through time by using the concept of time transforms. The concept of a latency function can be seen as a discrete time version of a time transform. When one slows a behaviour through a time transform, all streams are slowed down exactly by the same amount. This manner of slowing down a behaviour corresponds to how one would expect a stretch robust system to react in our discrete theory.

Stutter invariance for Linear Temporal Logic (LTL) properties has been investigated in [7], in which a stuttered path slows down all inputs and outputs of a program by the same amount. Stutter invariant properties are ones which, if they are satisfied by a program, then they are also satisfied by all stutterings of its behaviour. If a Lustre program is stretch robust, then its inputs can be safely slowed down without the risk of breaking the constraint imposed by a stutter invariant property on the program.

The theory of stability [10] considers programs whose outputs fluctuate when their inputs are kept constant. Programs that do not exhibit such a phenomenon are said to be *stable*; when the inputs of these programs are unchanged, the outputs will converge to stable values after a finite period of time. The concept of stability relates to the concept of fast-enough robustness. An input that has stopped changing is similar to an input which is stuttering when considered over some finite horizon of time. While the theory of stability requires the output of a system to eventually converge to some particular

value, fast-enough robustness requires an output to converge to an expected value before the sequence of repetitions ends.

Instead of checking whether a system exhibits certain classes of behaviour when an environment changes, it is possible to check whether a system degrades gracefully when the environment misbehaves. In [11], the authors consider a robustness approach in response to environments that fail to obey the assumptions made during system design. A system is said to be robust, if a small number of violations of the environment assumptions causes only a small number of violations of the system specification.

It is also possible to use a probabilistic approach to understand how changes in the environment are propagated through the system's components, and how the behaviour of these components under changed or missing input contributes to cause unacceptable system wide behaviour [12]. From our perspective, the general approach is interesting because it can help to isolate those components that misbehave under slow input, causing a complex system to fail.

VIII. Conclusions

Since input stutter can arise in various situations, especially in systems that finely sample input, it is crucial that such systems do not change their behaviour as such transformations on their input occur. In this paper, we have identified a number of different levels of robustness with respect to slowdown which one may require, and presented sound checks using static analysis of the code or using symbolic verification techniques over the system's behaviour.

One important observation regards the strong (and highly compositional) property of stretch robustness. This property is found in various guises in slowdown models that are simpler than our own. These models are often found to either assume that programs will be stateless, or else will define slow input in a way that does not bear on program state. In our model, stutter is able to interact with the internal state of the programs under consideration resulting in outputs with more complex relationships to those generated under the unslowed input. Nonetheless it was found that even under these conditions, there are non-trivial stateful programs that remain stretch robust to input slowdown.

Nonetheless, the restrictions that are imposed by stretch robustness in the complex slowdown scenario considered are quite strong, and consequently it is useful to possess weaker properties for those situations in which stretch robustness does not apply. Weaker robustness properties are however not as compositional as stretch robustness, making a static analysis approach less powerful than it would be with the stronger property. On the other hand, we have seen that dynamic analysis allows for the analysis of programs on a global level, allowing greater effectiveness in checking all kinds of robustness property - albeit at an increased computational cost. The two approaches can, however, be combined, allowing for the analysis of more complex programs.

One major restriction of our results is that we assume that all the inputs of the system are slowed down by the same amount. In practice, this may be too strong a restriction, for instance with some nodes using a combination of external inputs and streams coming from other nodes and which may have been slowed down further.

Another restriction is that we limit our analysis techniques to Boolean Lustre programs or circuits. The static analysis results, depending only on abstract properties of primitive programs and the way they are interconnected, can be applied to programs over non-finite types. On the other hand, all dynamic analysis results can only be applied on Boolean Lustre programs. In the future, we plan to relax this constraint by using control graph analysis techniques to programs with numeric values, using approaches similar to [13].

References

[1] I. Bondin and G. Pace, "Static and dynamic analysis for robustness under slowdown," Proc. 4th International Conference on Computational Logics, Algebras, Programming, Tools, and Benchmarking (COMPUTATION TOOLS '13), IARIA, May 2013, pp. 14–22.

[2] P. Caspi, D. Pilaud, N. Halbwachs, and J. Plaice, "Lustre: a declarative language for programming synchronous systems," Proc. 14th ACM SIGACT-SIGPLAN symposium on Principles of programming languages (POPL '87), ACM, Jan. 1987, pp. 178–188.

[3] C. Colombo, G. Pace, and G. Schneider, "Safe runtime verification of real-time properties," Proc. 7th International Conference on Formal Modelling and Analysis of Timed Systems (FORMATS '09), Springer Verlag, Sep. 2009, pp. 103–111.

[4] A. Gamatié, Designing embedded systems with the SIGNAL programming language - synchronous reactive specification. Springer, 2010.

[5] P. Le Guernic, J.-P. Talpin, and J.-C. Le Lann, "Polychrony for system design," Journal of Circuits, Systems and Computers, vol. 12(3), Jun. 2003, pp. 261–304.

[6] L. Carloni, K. Mcmillan, and A. Sangiovanni-Vincentelli, "Theory of latency insensitive design," IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, vol. 20(9), Sep. 2001, pp. 1059–1076.

[7] D. Peled and T. Wilke, "Stutter-invariant temporal properties are expressible without the next-time operator," Information Processing Letters, vol. 63(5), Sep. 1997 pp. 243–246.

[8] N. Halbwachs, P. Raymond, and C. Ratel, "Generating efficient code from data-flow programs," Proc. 3rd International Symposium on Programming Language Implementation and Logic Programming (PLILP '91), LNCS 528, Springer Verlag, Aug. 1991, pp. 207–218.

[9] S. Brown and Z. Vranesic, Fundamentals of Digital Logic with VHDL Design, second edition. McGraw-Hill Higher Education, 2005, pp. 522–531.

[10] N. Halbwachs, J.-F. Héry, J.-C. Laleuf, and X. Nicollin, "Stability of discrete sampled systems," Proc. 6th International Symposium on Formal Techniques in Real-Time and Fault-Tollerant Systems (FTRTFT '00), Springer Verlag, London, Sep. 2000, pp. 1–11.

[11] R. Bloem, K. Greimel, T. Henzinger, and B. Jobstmann, "Synthesizing robust systems," Proc. Formal Methods in Computer-Aided Design (FMCAD '09), IEEE Computer Society, Nov. 2009, pp. 85–92.

[12] X. Ge, R. Paige, and J. McDermid, "Probabilistic failure propagation and transformation analysis," Proc. 28th International Conference on Computer Safety, Reliability and Security (SAFECOMP '09), LNCS 5775, Springer Verlag, Sep. 2009, pp. 215–228.

[13] B. Jeannet, N. Halbwachs, and P. Raymond, "Dynamic partitioning in analyses of numerical properties," Proc. Static Analysis Symposium (SAS'99), LNCS 1694, Springer Verlag, Sep. 1999, pp. 39–50.