Synthesizing Modular Invariants for Synchronous Code*

Pierre-Loïc Garoche
Onera, The French Aerospace Lab

Arie Gurfinkel
SEI / CMU

Temesghen Kahsai
NASA Ames / CMU

In this paper, we explore different techniques to synthesize modular invariants for synchronous code encoded as Horn clauses. Modular invariants are a set of formulas that characterizes the validity of predicates. They are very useful for different aspects of analysis, synthesis, testing and program transformation. We describe two techniques to generate modular invariants for code written in the synchronous dataflow language Lustre. The first technique directly encodes the synchronous code in a modular fashion. While in the second technique, we synthesize modular invariants starting from a monolithic invariant. Both techniques, take advantage of analysis techniques based on property-directed reachability. We also describe a technique to minimize the synthesized invariants.

1 Introduction

In this paper, we present an algorithm for synthesizing modular invariants for synchronous programs. Modular invariants are useful for different aspects of analysis, synthesis, testing and program transformation. For instance, embedded systems often contains complex modal behavior that describe how the system interacts with its environment. Such modal behaviors are usually described via hierarchical state machines (HSM). The latter are used in model-based development notations such as Simulink and SCADE — the de-facto standard for software development in avionics and many other industries. For the purpose of safety analysis, Simulink/SCADE models are compiled to a lower level modeling language, usually a synchronous dataflow language such as Lustre [4]. Preserving the original (hierarchical and modular) structure of the model is paramount to the success of the analysis process. In this paper, we illustrate a technique to preserve such structure via a modular compilation process. Specifically, our techniques consists of compiling in a modular fashion Lustre programs into Horn clauses.

The use of *Horn clauses* as intermediate representation for verification was proposed in [8], with the verification of concurrent programs as the main application. The underlying procedure for solving sets of recursion-free Horn clauses, over the combined theory of Linear Rational Arithmetic (LRA) and Uninterpreted Functions (UF), was presented in [7]. A range of further applications of Horn clauses includes inter-procedural/exchange format for verification problems that is supported by the SMT solver Z3 [9]. In this paper, we show how to use such techniques to generate modular invariants for Lustre programs.

While on one hand we generate modular invariants by encoding the synchronous code in a modular fashion, on the other hand, we are interested to synthesize modular invariants from a monolithic invariant. That is, given an invariant for a program that is obtained by flattening the hierarchical structure, we want to reconstruct the modular invariant. We describe a technique that generates modular invariants from a monolithic one.

*This work was partially supported by the ANR-INSE-2012 CAFEIN project, and also by NASA Contract No. NNX14AI09G.

Nikolaj Bjørner, Fabio Fioravanti, Andrey Rybalchenko,
Valerio Senni (Eds.): First Workshop on
Horn Clauses for Verification and Synthesis (HCVS 2014)
EPTCS 169, 2014, pp. 19–30, doi:10.4204/EPTCS.169.4

Finally, once we obtain modular invariants, we are interested in minimizing such invariants. In particular, in our setting the invariants correspond to a contract that a component must satisfy. In general, the contract is state-full, i.e., it is an automaton (or a protocol). In practice, it is important to generate contracts with minimal state (i.e., minimal automata). We sketch a direction for minimization based on a CEGAR-like [5] technique.

In summary, this paper makes the following contributions:

- Two techniques to generate modular invariants for synchronous code. One based on a modular compilation of Lustre code into Horn clauses, and a second one based on extracting modular invariants from monolithic invariants.
- An implementation of these techniques that targets programs written in the synchronous dataflow language Lustre.
- A sketch of a technique to minimize invariants using CEGAR-type approach.

The rest of the paper is organized as follows. In the next section, we introduce synchronous languages in general and the synchronous dataflow language Lustre. In Section 3, we describe the first technique a procedure to compile in a modular fashion Lustre code into Horn clauses. In Section 4, we illustrate the second technique to derive modular invariants from a monolithic one. We conclude the paper in Section 5 with a discussion on the minimization of invariants.

2 Preliminaries

Synchronous languages are a class of languages proposed for the design of so called "reactive systems" – systems that maintain a permanent interaction with physical environment. Such languages are based on the theory of synchronous time, in which the system and its environment are considered to both view time with some "abstract" universal clock. In order to simplify reasoning about such systems, outputs are usually considered to be calculated instantly [2]. Examples of such languages include Esterel [3], Signal [1] and Lustre [4]. In this paper, we will concentrate on the latter. Lustre combines each data stream with an associated clock as a mean to discretize time. The overall system is considered to have a universal clock that represents the smallest time span the system is able to distinguish, with additional, coarser-grained, user-defined clocks. Therefore the overall system may have different subsections that react to inputs at different frequencies. At each clock tick, the system is considered to evaluate all streams, so all values are considered stable for any actual time spent in the instant between ticks. A stream position can be used to indicate a specific value of a stream in a given instant, indexed by its clock tick. A stream at position 0 is in its initial configuration. Positions prior to this have no defined stream value. A Lustre program defines a set of equations of the form:

$$y_1, \ldots, y_n = f(x_1, \ldots, x_m, u_1, \ldots, u_o)$$

where y_i are output or local variables and u_i are input variables. Variables in Lustre are used to represent individual streams and they are typed, with basic types including streams of *Real* numbers, *Integers*, and *Booleans*. Lustre programs and subprograms are expressed in terms of *Nodes*. Nodes directly model subsystems in a modular fashion, with an externally visible set of inputs and outputs. A *node* can be seen as a mapping of a finite set of input streams (in the form of a tuple) to a finite set of output streams (also expressed as a tuple). The *top node* is the main node of the program, the one that interface with the environment of the program and never be called by another node.

At each instant t, the node takes in the values of its input streams and returns the values of its output streams. Operationally, a node has a cyclic behavior: at each cycle t, it takes as input the value of each input stream at position or instant t, and returns the value of each output stream at instant t. This computation is assumed to be immediate in the computation model. Lustre nodes have a limited form of memory in that, when computing the output values they can also look at input and output values from previous instants, up to a finite limit statically determined by the program itself. Figure 1 describes a simple Lustre program: a node that every four computation steps activates its output signal, starting at the third step. The `reset` input reinitializes this counter.

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

Figure 1: A simple Lustre example.

Typically, the body of a Lustre node consists in a set of definitions, stream equations of the form $x = t$ (as seen in Figure 1) where x is a variable denoting an output or a locally defined stream and t is an expression, in a certain stream algebra, whose variables name input, output, or local streams. More generally, x can be a tuple of stream variables and t an expression evaluating to a tuple of the same type. Most of Lustre's operators are point-wise lifting to streams of the usual operators over stream values. For example, let $x = [x_0, x_1, \dots]$ and $y = [y_0, y_1, \dots]$ be two integer streams. Then, $x + y$ denotes the stream $[x_0 + y_0;\ x_1 + y_1, \dots]$; an integer constant c, denotes the constant integer stream $[c, c, \dots]$. Two important additional operators are a unary shift-right operator *pre* ("previous"), and a binary initialization operator $\rightarrow$ ("followed by"). The first is defined as $pre(x) = [u, x_0, x_1, \dots]$ with the value u left unspecified. The second is defined as $x \rightarrow y = [x_0, y_1, y_2, \dots]$. Syntactical restrictions on the equations in a Lustre program guarantee that all its streams are well defined: e.g. forbidding recursive definitions hence avoiding algebraic loops.

3 Modular synthesis

In the last section we gave an informal overview of the synchrounous dataflow language Lustre. A formal semantics of Lustre is described in [4]. In this section, we describe our technique to generate modular Horn clauses starting from Lustre code.

A Lustre program L is a collection of nodes $[N_0, N_1, \dots, N_m]$ where N_0 is the top node, i.e., the main function. Each node is represented by the following tuple:

$$N_i = (\mathscr{I}_i, \mathscr{O}_i, \mathscr{L}_i, Init_i, Trans_i)$$

where $\mathscr{I}_i, \mathscr{O}_i$ and $\mathscr{L}_i$ are set of input, output and local variables. $Init_i$ and $Trans_i$ represents the set of formulas for the initial states and the transition relation respectively, and they are defined as follows:

$$\bigwedge_{i \in \mathbb{N}} v_i = \rho(s_i)$$

where

- $v_i \in \mathscr{O}_i \cup \mathscr{L}_i$ and s_i is the expression such that $Vars(s_i) \subseteq \mathscr{O}_i \cup \mathscr{I}_i \cup \mathscr{L}_i$. $Vars(s_i)$ is the set of variables in s_i;

- expressions s_i are arbitrary Lustre expression including node calls $N_j(u_1,\ldots,u_n)$;

- ρ function maps expression to expression and projects the binary initialization operators $\rightarrow$:

$$a \rightarrow b \text{ is projected as } \begin{cases} a \text{ in } Init_i \\ b \text{ in } Trans_i \end{cases}$$

Given a Lustre program $L = [N_0, N_1, \ldots, N_m]$, a safety property P is any expression over the signature over the main node N_0. A common way to express safety property in synchronous languages is the use of synchronous observers [11]. The latter is a wrapper used to test observable properties of a node N with minimal modification the node itself; it returns an error signal if the property does not hold, reducing the more complicated property to a single Boolean stream where we need to check if the stream is constantly true.

We now describe the compiler `lus2horn` : $L \rightarrow H$, which given a Lustre program $L = [N_0, N_1, \ldots, N_m]$ generates a set of Horn clauses H that are semantically equivalent to L. The current compiler only handles a simplified version of the Lustre v4 language without the constructs to manipulate clocks, or complex data structure. The following steps describe the various stages of `lus2horn`:

Normalization: In the first phase the compiler *lus2horn* transforms the equations of the Lustre node to extract the stateful computations that appear inside expressions. Stateful computation can either be the explicit use of a *pre* construct or the call to another node which may be stateful. The extraction is made through a linear traversal of the node's equations, introducing new equations for stateful computation[1]. When possible, tuple definition are split as simpler definitions. To ease later computation, each node call is labeled by a unique identifier. The following set of expressions give an example of such normalization:

$$a \;=\; false \rightarrow (\textbf{not } reset \textbf{ and not } (\textbf{pre } b)); \quad \longrightarrow \quad \begin{cases} pb \;=\; \textbf{pre } b; \\ a \;=\; false \rightarrow (\textbf{not } reset \textbf{ and not } pb); \end{cases}$$

$$y \;=\; 3 + node(x,2); \quad \longrightarrow \quad \begin{cases} res\_node1 = node^{uid_1}(x,2); \\ y \;=\; 3 + res\_node1; \end{cases}$$

The following definitions are the normalization functions $Norm_N$, $Norm_{Eq}$ and $Norm_{Expr}$, a single node N, an equations and an expression respectively.

- The normalization of an expression returns a modified expression along with a set of newly bound

[1] As opposed to 3 addresses code, only the stateful part of expression is extracted.

stateful expressions and associated new variables:

$$
\begin{array}{lcl}
Norm_{Expr}(e, Eqs, Vars) \triangleq & & \\
v & \rightarrow & v, Eqs, Vars \\
cst & \rightarrow & cst, Eqs, Vars \\
\mathbf{op}(e_1, \ldots, e_n) & \rightarrow & \left\{ \begin{array}{l} \textbf{let } e_1, Eqs, Vars = Norm_{Expr}(e_1, Eqs, Vars) \textbf{ in} \\ \vdots \\ \textbf{let } e_n, Eqs, Vars = Norm_{Expr}(e_n, Eqs, Vars) \textbf{ in} \\ \quad op(e'_1, \ldots, e'_n), Eqs, Vars \end{array} \right. \\
\textbf{pre } e & \rightarrow & \left\{ \begin{array}{l} \textbf{let } e', Eqs, Vars = Norm_{Expr}(e, Eqs, Vars) \textbf{ in} \\ \textbf{let } x \notin Vars \textbf{ in} \\ \quad x, \{x = pre e';\ \} \cup Eqs, \{x\} \cup Vars \end{array} \right. \\
N_i(e_1, \ldots, e_n) & \rightarrow & \left\{ \begin{array}{l} \textbf{let } e_1, Eqs, Vars = Norm_{Expr}(e_1, Eqs, Vars) \textbf{ in} \\ \vdots \\ \textbf{let } e_n, Eqs, Vars = Norm_{Expr}(e_n, Eqs, Vars) \textbf{ in} \\ \textbf{let } x \notin Vars \textbf{ in} \\ \quad x, \{x = N_i^{uid}(e'_1, \ldots, e'_n);\ \} \cup Eqs, \{x\} \cup Vars \end{array} \right.
\end{array}
$$

In $Norm_{expr}$, op is a Lustre operator, N_i is a node in a Lustre program L and uid is a unique identifier associated to the call of N_i with arguments $(e'_1, \ldots, e'_n)$.

- Normalization of a node equation simplifies tuple definitions and normalizes each expression. It returns a set of equations with normalized expressions:

$$
\begin{array}{lcl}
Norm_{Eq}(\{v_1, \ldots, v_n = s\}, Eqs, Vars) \triangleq & & \\
v_1, \ldots, v_n = s_1, \ldots, s_n & \rightarrow & \left\{ \begin{array}{l} \textbf{let } Eqs, Vars = Norm_{Eq}(\{v_1 = s_1\}, Eqs, Vars) \textbf{ in} \\ \vdots \\ \quad Norm_{Eq}(\{v_n = s_n\}, Eqs, Vars) \end{array} \right. \\
v = s & \rightarrow & \left\{ \begin{array}{l} \textbf{let } s', Eqs, Vars = Norm_{Expr}(e, Eqs, Vars) \textbf{ in} \\ \quad \{v = s'\} \cup Eqs, Vars \end{array} \right.
\end{array}
$$

- Last the normalization of a node amounts to normalize each expression in each definition; the newly bound variables are added to the set of local variables.

$$
\begin{array}{l}
Norm_N(N) = (\mathscr{I}_i,\ \mathscr{O}_i,\ \mathscr{L}_i \cup NewVars,\ Init_i,\ Trans_i) \\
\text{where } \left\{ \begin{array}{l} \textbf{let } InitVars = \mathscr{I}_i \cup \mathscr{O}_i \cup \mathscr{L}_i \textbf{ in} \\ \textbf{let } Init_{Ni}, Vars = Norm_{Eq}(Init_i, InitVars) \textbf{ in} \\ \textbf{let } Trans_{Ni}, Vars = Norm_{Eq}(Trans_i, Vars) \textbf{ in} \\ \quad NewVars = Vars \setminus InitVars \end{array} \right.
\end{array}
$$

State computation: The state of a node is characterized by its memories: variables defined by *pre* constructs, as well as the memories associated to each of its calling node instances.

We first define the set of local memories for a node:

$$xMem(N_i) = \{v \in \mathscr{L}_i \mid \{v = pre\ e;\ \} \in Trans_i\}$$

Then we characterize the set of callee instances, using their unique identifies *uid*:

$$Inst(N_i) = \{(N_j, uid) \mid \{v = N_j^{uid}(e_1, \ldots, e_n);\ \} \in Trans_{Ni}\}$$

We denote by $State_i$ the set of memories fully characterizing the state of a N_i node instance[2].

$$State(N_i) = Mem(N_i) \cup \{uid_{N_j}\_v \mid (N_j, uid) \in Inst(N_i) \wedge v \in State(N_j)\}$$

Generating Horn predicates: Once the memories of a node have been identified, a predicate describing the transition relation can be expressed as a Horn clause. The latter is defined over inputs, outputs, previous value of the node's state $State_i$ and updated state $State'_i$. We then produce the following Horn rule encoding the transition relation predicate:

$$(i) \qquad (\texttt{rule}\ (\Rightarrow\ \phi(Trans_i)\ (T_{N_i}\ \mathscr{I}_i\ \mathscr{O}_i\ State_i\ State'_i)))$$

Here, we use the Horn clause format introduced in Z3 [9], where $(\texttt{rule}\ expr)$ universally quantify the free variables of the SMT-LIB expression $expr$.The function $\phi(expr)$ is recursively defined as

$$\begin{array}{lcl}
\phi(v = pre\ e;\ ) & \rightarrow & (= v'\ \ e) \\
\phi(v_1, \ldots, v_n = N_j^{uid}(e_1, \ldots, e_m);\ ) & \rightarrow & (T_{N_j}\ e_1\ \ldots\ e_m\ v_1\ \ldots\ v_n\ uid_{N_j}_v_1\ \ldots\ uid_{N_j}_v_k\ uid_{N_j}_v'_1\ \ldots\ uid_{N_j}_v'_k) \\
\phi(v = e;\ ) & \rightarrow & (= v\ \ e) \\
\phi(eq;\ eqs) & \rightarrow & (\textbf{and}\ (\phi(eq))\ (\phi(eqs))
\end{array}$$

where

- v in $v = pre\ e$ is by construction in $Mem(N_i) \subseteq State_i$ and $v' \in State'_i$;
- $(uid_{N_j}\_v_l)_{1 \leq l \leq k} \in State_i$ denotes the state representation of the instance uid of node N_j and $(uid_{N_j}\_v'_l)_{1 \leq l \leq k} \in State'_i$.

Similarly the Horn rule encoding the initial state is defined as follows:

$$(ii) \qquad (\texttt{rule}\ (\Rightarrow\ \phi(Init_i)\ (I_{N_i}\ \mathscr{I}_i\ \mathscr{O}_i\ State'_i)))$$

Given a Horn clause H of the form $(\texttt{rule}\ \Rightarrow\ Body\ B)$, where $Body$ is a conjunction of expression and B is a predicate, a model $\pi : B \mapsto \mathscr{F}$ is a mapping from B to a set of formulas $\mathscr{F}$ such that it makes every rule H valid. In other words, π represents the set of invariants for the Horn clause H.

Let $(Main\ \mathscr{I}_{N_0}\ \mathscr{O}_{N_0}\ State_{N_0})$ be the predicate encoding the collecting semantics of the main node N_0 of the Lustre program $L = [N_0, N_1, \ldots, N_m]$. Each node N_i being defined by the two Horn clauses I_{N_i} and T_{N_i}. The semantics of the whole Lustre program is inductively encoded as follows:

$$\begin{array}{l}
(iii)\ (\texttt{rule}\ (\Rightarrow\ (I_{N_0}\ \mathscr{I}_{N_0}\ \mathscr{O}_{N_0}\ State_{N_0})\ (Main\ \mathscr{I}_{N_0}\ \mathscr{O}_{N_0}\ State_{N_0})) \\
(iv)\ (\texttt{rule}\ (\Rightarrow\ (\textbf{and}\ (T_{N_0} \mathscr{I}'_{N_0}\ \mathscr{O}'_{N_0}\ State_{N_0}\ State'_{N_0})\ (Main\ \mathscr{I}_{N_0}\ \mathscr{O}_{N_0}\ State_{N_0}))\ (Main\ \mathscr{I}'_{N_0}\ \mathscr{O}'_{N_0}\ State'_{N_0}))
\end{array}$$

(iii) characterizes the set of initial states while (iv) defines the induction step. Let P_L be the expression representing the safety property for the Lustre program $L = [N_0, N_1, \ldots, N_m]$. As specified above, P_L is a predicate defined over the signature of the main node N_0. Let P_H be its equivalent in Horn clauses format, i.e. $P_H = \phi(P_L)$. Then, we encode the checking of the property P_H on the Horn encoding as defined above in the following manner:

$$(v) \quad (\texttt{rule}\ \Rightarrow\ \ (\textbf{and}\ (Main\ \mathscr{I}_{N_0}\ \mathscr{O}_{N_0}\ State_{N_0})\ (\textbf{not}\ P_H))\ Error)$$

[2]By construction, circular definition of nodes are forbidden in Lustre: this recursive definition is well-founded.

where $Error$ is the predicate marking the error state. Such state is reachable if and only if the property is not valid. If its unreachable then the property is valid. Tools like Z3 [9] are able to give a certificate for (un)reachability. A certificate of reachability is in a nutshell a proof of unsatisfiability. In this case the certificate is presented as a trace. A certificate for un-reachability is in a nutshell a model for the recursive predicates. The following theorem establishes a correspondence relation between Lustre program and the compiled Horn clauses H using the function `lus2horn`.

Theorem 1. *The semantics of the Lustre program $L = [N_0, \ldots, N_m]$ and the semantics of its Horn clauses encoding $H =$ `lus2horn`(L) are in strong bisimulation.*

Proof: A classical strong bisimulation proof only sketched here: the two set of initial states coincide while each transition that could be performed on one side is computable on the other: (i) for every execution of L there is a derivation of H, and (ii) for every derivation of H there is an equivalent execution of L.

3.1 Example

As an example of the compilation process `lus2horn`, we will consider a simple Lustre program that compares two implementations of a 2-bit counter: a low-level Boolean implementation and a higher-level implementation using integers. The left hand side of Figure 2 illustrate the Lustre code. The `greycounter` node internally repeats the sequence $ab = \{00, 01, 11, 10, 00, ...\}$ indefinitely, while the `integercounter` node repeats the sequence $time = \{0, 1, 2, 3, 0, ...\}$. In both cases the counter returns a boolean value that is true if and only if the counter is in its third step or input variable *reset* is true. The top node test is an example of a synchronous observer. So we wish to verify the safety property that both implementations have the same observable behavior, i.e. that the stream *OK* is always true. On the right hand side of Figure 2 is the corresponding Horn clauses encoding [3]. The predicates IC, GC and T encode the transition relation of the nodes `intcounter`, `greycounter` and `top` respectively. While the predicate IC_Init, GC_Init and T_Init encode the initial states of the three nodes. The predicate M encode the main entry point of the two counters Lustre program; while $Error$ is a predicate used to mark the error states.

Using the PDR engine of Z3 [9] we are able to get the modular invariant of the Horn encoding for the predicate IC, GC, M and T. For example, for the predicates IC_Init, IC and GC_Init, GC we obtain the following invariants:

$$
\begin{array}{lcll}
IC\_Init(reset, out, time) & = & & (time = 0) \wedge \neg(out) \\
IC(reset, out, time, time') & = & & (time < 3 \rightarrow time' \geq 0) \\
 & & \wedge & (out \rightarrow time \leq 1) \\
 & & \wedge & (time' \leq 0 \vee \neg(time \leq 3) \vee \neg(time \geq 3)) \\
 & & & (\neg(time \geq 3) \vee time' \geq 0) \\
\hline
GC\_Init(reset, out, a, b) & = & & \neg(out) \wedge \neg(a) \wedge \neg(b) \\
GC(reset, out, a, b, a', b') & = & \wedge & b' \leftrightarrow a \\
 & & \wedge & \neg a' \leftrightarrow b \\
 & & \wedge & \neg out \leftrightarrow \neg a \vee a'
\end{array}
$$

For example, for the node `intcounter`, denoted by the predicates IC_Init and IC, we obtain that the variable *time* is bound in the interval $[0, 3]$.

[3] The variable *time* on the Lustre program is denoted by the variable t in the Horn encoded.

```
node greycounter (reset:bool)
    returns (out:bool);
var a,b:bool;
let
 a = false →
       (not reset and not pre(b));
 b = false → (not reset and pre(a));
 out = a and b;
tel

node intcounter (reset:bool)
     returns (out:bool);
var time: int;
let
 time = 0 →
      if reset or pre(time) = 3
                 then 0
            else pre time + 1;
  out = (time = 2);
tel

node top (reset:bool)
     returns (OK:bool);
var b,d:bool;
let
  b = greycounter(reset);
  d = intcounter(reset);
  OK = b = d;
  --!PROPERTY : OK=true;
tel
```

```
(declare-rel GC(Bool Bool Bool Bool Bool Bool))
(declare-rel IC(Bool Bool Int Int))
; ... predicate declartions ...
(declare-rel Error())
; ... variable declarations ... ;

(rule (⇒ (and (= a' (and (not reset) (not b)))
              (= b' (and (not reset) a))
              (= out (and b' a')))
         (GC reset out a b a' b')))
(rule (⇒ (and (= a' false)
              (= b' false)
              (= out (and b' a'))
         (GC_Init reset out a' b')))
(rule (⇒ (and (= t'
            (ite (or reset (= t 3)) 0 (+ t 1)))
           (= out (= t' 2)))
         (IC reset out t t')))
(rule (⇒ (and (= t' 0) (= out (= t' 2)))
         (IC_Init reset out t')))
(rule (⇒ (and (GC reset gout ga gb ga' gb')
              (IC reset iout it it')
             (= ok (= iout gout)))
        (T reset ok ga gb it ga' gb' it')))
(rule (⇒ (and (GC_init reset gout ga' gb')
              (IC_init reset iout it')
             (= ok (= iout gout)))
         (T_Init reset ok ga' gb' it')))

(rule (⇒ (T_Init reset ok ga gb it)
    (M reset ok ga gb it)))

(rule (⇒ (and (M reset' ok' ga gb it)
              (T reset tok ga gb it ga' gb' it'))
         (M reset tok ga' gb' it')))
(rule (⇒ (and (M reset ok ga gb it)
              (not (= ok true)))
         Error)
(query Error)
```

Figure 2: Two counters example.

4 From monolithic to modular invariants

In the last section we illustrated how we compile a Lustre program in a modular fashion and obtain a Horn clause representation. Such encoding allows to generate modular invariants by exploiting tools based on property-directed reachability such as Z3 [9]. In this section, we describe a technique to synthesize a modular invariants given a monolithic invariant. The latter is an invariant over an inlined version of the program, in which all the nodes (non-recurisve predicate) are inlined to the main Lustre node. Formally, it is defined as follows:

Definition 1 (Monolithic invariant)**.** *Let $L = [N_0, \ldots, N_m]$ be a Lustre program and $H = \texttt{lus2horn}(L)$ be the set of Horn clauses defined in the previous section. Let $K = \texttt{inline}(\texttt{H})$ an inlined version of H. That is all the non-recursive predicate are inlined by resolution. The function $\texttt{inline}(H)$, will generate a tuple $M = (I_{N_0}, T_{N_0})$, where I_{N_0} and T_{N_0} are the predicates for the initial states and the transition relation as defined in (i) and (ii) of Section 3. Let P_K be a safety property. Checking the property P_K over M as defined in (v) of Section 3, we obtain an invariant $\pi : M \to \mathscr{F}$, where $\mathscr{F}$ is a set of formulas valid in M.*

We call π a monolithic invariant.

Given a monolithic invariant π defined as above, we are interested in obtaining modular invariants. That is, given a modular encoding of the program we would like to reconstruct the modular invariant from π. The following theorem states that given a monolithic invariant we can obtain a modular invariant for a modularly defined Horn clauses.

Theorem 2. *Let $\pi : M \rightarrow \mathscr{F}$ be a monolithic invariant for $M = (I_{N_0}, T_{N_0})$ of an inlined Horn clause $K = \texttt{inline}(H)$. Then, π can be extended to a model π' for the Horn clause H; where H is a modular set of Horn clauses as defined in Section 3.*

Given a modular Horn clauses H as defined by the rules (i) and (ii) in Section 3, where I_{N_0} and T_{N_0} are the predicates for the initial and transition relation of the top node, we encode the following Horn rules in order to get a modular invariants:

$$
\begin{array}{lll}
(vi) & (\texttt{rule}\ (\Rightarrow & (I_{N_0}\, \mathscr{I}_{N_0}\, \mathscr{O}_{N_0}\, State_{N_0}) \\
 & & (Mono\ \mathscr{I}_{N_0}\, \mathscr{O}_{N_0}\, State_{N_0}))) \\
(vii) & (\texttt{rule}\ (\Rightarrow\ (\textbf{and} & (T_{N_0}\, \mathscr{I}'_{N_0}\, \mathscr{O}'_{N_0}\, State_{N_0}\, State'_{N_0}) \\
 & & (Mono\ \mathscr{I}_{N_0}\, \mathscr{O}_{N_0}\, State_{N_0}) \\
 & & (\textbf{not}\ (Mono\ \mathscr{I}'_{N_0}\, \mathscr{O}'_{N_0}\, State'_{N_0}))) \\
 & Error))
\end{array}
$$

where *Mono* is the predicate representing the monolithic invariant. Rule (vi) encode the rule for the initial states, while rule (vii) encode the reachability of the transition relation, where *Error* is the predicate that marks the error state. By checking the reachability of the *Error* state we can obtain, as expected, a certificate of un-reachability of it, producing a modular invariant for the predicates of in H.

4.1 Example (cont.)

Continuing the example from previous section, let `MONO` be the monolithic invariant for the inlined version of the `two_counters`, defined as follows:

```
(define-fun MONO ((reset Bool)(ok Bool) (ga Bool) (gb Bool) (it Int)) Bool
  (let ((tmp (not (or (not (<= it 0)) (not (>= it 0))))))
    (and ok
         (or tmp ga gb reset)
         (or (<= it 2) (not ga) (not reset))
         (<= it 3)
         (or (>= it 3) (not gb) ga)
         (or (>= it 2) (not gb) (not ga)))))
```

Given the modular definition of the Horn rules for IC, GC and T as defined in the sub-section 3.1, we encode the reachability challenge in the following way:

```
(rule (=> (and (T reset tok ga gb it ga' gb' it'))
               (MONO reset' ok ga gb it)
               (not (MONO reset' tok ga' gb' it')))
               Error))
(query Error)
```

Z3 produces a certificate for such queries over the predicates IC, GC and T.

5 Minimization of modular interface

Given a Horn clause H of the form $(\texttt{rule}\ (\Rightarrow\ Body\ B))$, where *Body* is a conjunction of expressions and B is a predicate, we can get an invariant $\pi : B \mapsto \mathscr{F}$, which is of the form:

$$B(\mathscr{I},\mathscr{O},State,State') = \bigwedge f$$

where $f \in \mathscr{F}$, and $\mathscr{F}$ is a set of formulas which makes the predicate B valid. Such invariant could be obtained using techniques from property-based reachability [9] or other techniques for invariant generators, e.g., template-based [10] or abstract interpretation-based [6]. In a nutshell, π represents a set of formulae which prescribe the behavior of that particular component. In other words, it represents an interface (or contract) of the component. We are interested in obtaining a smaller interface (or contract). Specifically, we are interested in minimizing the set of state variables for π. This means, minimizing the *bits* representing the state variables. For example, if the variables in *State* are represented by bit vectors, we are interested in minimizing the number of bits. More generally, we define the notion of *ranks of state variables* as follows:

Definition 2 (Rank of state variables)**.** *Let $\mathscr{V}$ be a set of state variables. We define the number of bits as a function* $\texttt{rank} : \mathscr{P}(\mathscr{V}) \to \mathbb{N}x\mathbb{N}$*, from the set of state variables V to a pair $(\bar{I},\bar{B})$, where $\bar{I}$ is the number of integers and $\bar{B}$ is the number of bits representing it.*

$$\begin{array}{rcl} \texttt{rank} : \mathscr{P}(\mathscr{V}) & \to & \mathbb{N}\times\mathbb{N} \\ \{v\} & \mapsto & \begin{cases} (0,1) & when\ type(v) = Bool \\ (1,0) & when\ type(v) = Int \\ (0,n) & when\ type(v) = BitVector^n \end{cases} \\ V & \mapsto & \sum_{v\in V} \texttt{rank}(v) \end{array}$$

The following are some examples of ranks of state variables:

$(*)$

State Variables	Rank
$P = \{v : Int, w : Int, b : Bool\}$	$\texttt{rank}(P) = (2,1)$
$Q = \{b_1 : Bool, b_2 : Bool\}$	$\texttt{rank}(Q) = (0,2)$
$R = \{w : BitVector^{1000}\}$	$\texttt{rank}(R) = (0,1000)$

Given an invariant we are interested in finding another invariant which is smaller, c.f. less rank of state variables. Formally, we define smaller invariant using lexicographic order as follows:

Definition 3 (Lexicographic order)**.** *Let $\mathscr{V}$ be a set of state variables. Let* $\texttt{rank} : \mathscr{V} \to \mathbb{N}x\mathbb{N}$ *be the function defined in Def. 2. Let $State_H, State_K \subseteq \mathscr{V}$ be two sets of state variables. There is a lexicographic order $<_{State}$ over the set $\mathscr{V}$ such that*

$$State_K <_{State} State_H \triangleq \texttt{rank}(State_K) <_{\mathbb{N}^2} \texttt{rank}(State_H)$$

where $\leq_{\mathbb{N}^2}$ denotes the usual lexicographic orderings of pairs of the classical order: $(a,b) \leq_{\mathbb{N}^2} (c,d)$ iff $a < c \vee (a = b \wedge b < d)$. The definition is lifted to invariants in the following way:

$$K(\mathscr{I},\mathscr{O},State_K,State'_K) <_{State} H(\mathscr{I},\mathscr{O},State_H,State'_H) \text{ iff } State_K <_{State} State_H.$$

For the predicates in example $(*)$ we have the ordering $Q <_{State} R <_{State} P$. We now sketch our idea of minimization of invariants. Let H be the set of Horn clauses as defined in Section 3, for which we have obtained a set of modular invariants $\pi_0,\ldots,\pi_m$ for the Horn rules $H_0,\ldots,H_m$ of H. Our aim is to generate a set of invariants $\pi'_0,\ldots,\pi'_n$ such that the predicates of the Horn rules $H_0,\ldots H_m$ have been minimized. That is, we generate predicates $Q_0,\ldots,Q_m$ such that $Q_0 <_{State} P_0,\ldots,Q_m <_{State} P_m$ where $P_0,\ldots,P_m$ are the predicates for the Horn rules $H_0,\ldots,H_m$.

In a nutshell, our approach of minimization is an iterative one which is based on a CEGAR [5] type technique. We start by abstracting the Horn rules by removing state variables from the signature of their predicates. Let H be the set of Horn rules and $H^\sharp$ the abstracted one. We check whether the safety property P is valid in $H^\sharp$ (c.f. (v) in Section 3). If the result is unsatisfiable then we are done, i.e., we have found certificate of unsatisfiability for smaller predicates, hence smaller invariants. If the result is satisfiable, it means we have a *spurious counterexamples*, traces for $H^\sharp$ that falsify the property P but are not legal traces of H. In this case we need to refine $H^\sharp$.

Let $t^\sharp$ be a trace of $H^\sharp$ that falsifies the property P. However, the corresponding trace t of H is not legal, hence is unsatisfiable. This means that there is a constraint Z in $t \backslash t^\sharp$. Therefore, we refine $H^\sharp$ by finding the smallest subset of Z that makes $t^\sharp$ unsatisfiable.

6 Conclusion

In this paper, we have described two techniques to synthesis modular invariants for synchronous code encoded as Horn clauses. Modular invariants are very useful for different aspects of analysis, synthesis, testing and program transformation. We have described two techniques to generate modular invariants for code written in the synchronous dataflow language Lustre. The first technique directly encodes the synchronous code in a modular fashion. While in the second technique, we synthesize modular invariants starting from a monolithic invariant. Both techniques take advantage of analysis techniques based on property-directed reachability. Both techniques have been implemented in a tool that targets the verification of safety properties specified in Lustre. We have also sketched a technique for minimizing invariants following a CEGAR-like approach. In the future, we plan to fully work the details of the minimization process and implement such techniques.

References

[1] Pascalin Amagbégnon, Loïc Besnard & Paul Le Guernic (1995): *Implementation of the Data-Flow Synchronous Language SIGNAL*. In: *PLDI*, pp. 163–173. Available at `http://doi.acm.org/10.1145/207110.207134`.

[2] Albert Benveniste & Gérard Berry (2002): *The Synchronous Approach to Reactive and Real-time Systems*. In Giovanni De Micheli, Rolf Ernst & Wayne Wolf, editors: *Readings in Hardware/Software Co-design*, Kluwer Academic Publishers, Norwell, MA, USA, pp. 147–159. Available at `http://doi.acm.org/10.1016/B978-155860702-6/50013-2`.

[3] Gérard Berry & Georges Gonthier (1992): *The Esterel Synchronous Programming Language: Design, Semantics, Implementation*. *Sci. Comput. Program.* 19(2), pp. 87–152. Available at `http://dx.doi.org/10.1016/0167-6423(92)90005-V`.

[4] Paul Caspi, Daniel Pilaud, Nicolas Halbwachs & John Plaice (1987): *Lustre: A Declarative Language for Programming Synchronous Systems*. In: *POPL*, pp. 178–188. Available at `http://doi.acm.org/10.1145/41625.41641`.

[5] Edmund M. Clarke, Orna Grumberg, Somesh Jha, Yuan Lu & Helmut Veith (2000): *Counterexample-Guided Abstraction Refinement*. In: *CAV*, pp. 154–169. Available at `http://dx.doi.org/10.1007/10722167_15`.

[6] Pierre-Loïc Garoche, Temesghen Kahsai & Cesare Tinelli (2013): *Incremental Invariant Generation Using Logic-Based Automatic Abstract Transformers*. In: *NASA Formal Methods*, pp. 139–154. Available at `http://dx.doi.org/10.1007/978-3-642-38088-4_10`.

[7] Ashutosh Gupta, Corneliu Popeea & Andrey Rybalchenko (2011): *Solving Recursion-Free Horn Clauses over LI+UIF*. In: *APLAS*, pp. 188–203. Available at http://dx.doi.org/10.1007/978-3-642-25318-8_16.

[8] Ashutosh Gupta, Corneliu Popeea & Andrey Rybalchenko (2011): *Threader: A Constraint-Based Verifier for Multi-threaded Programs*. In: *CAV*, pp. 412–417. Available at http://dx.doi.org/10.1007/978-3-642-22110-1_32.

[9] Krystof Hoder & Nikolaj Bjørner (2012): *Generalized Property Directed Reachability*. In: *SAT*, pp. 157–171. Available at http://dx.doi.org/10.1007/978-3-642-31612-8_13.

[10] Temesghen Kahsai, Yeting Ge & Cesare Tinelli (2011): *Instantiation-Based Invariant Discovery*. In: *NASA Formal Methods*, pp. 192–206. Available at http://dx.doi.org/10.1007/978-3-642-20398-5_15.

[11] John Rushby (2014): *The Versatile Synchronous Observer*. In: *Specification, Algebra, and Software*, pp. 110–128. Available at http://dx.doi.org/10.1007/978-3-642-54624-2_6.