On the Formalization of Signal-Flow-Graphs in HOL

Sidi Mohamed Beillahi, Umair Siddique, and Sofiène Tahar

Department of Electrical and Computer Engineering,
Concordia University, Montreal, Canada
{beillahi, muh_sidd, tahar}@ece.concordia.ca

Technical Report

November, 2014

Abstract

Signal-flow-graphs is a special kind of directed graph which is widely used to represent physical and engineering systems such as electrical networks, photonic and digital signal processing systems. Mathematically, signal-flow-graphs describes a linear relation among input and output of a system. Generally, Mason's gain formula is used to compute the transfer function of such systems using the signal-flow-graphs. Traditionally, the analysis of physical circuits based on signal-flow-graphs had been carried by paper-and-pencil based proofs and numerical techniques. Both of these methods have some known limitations of human-error proneness and incompleteness. In this report, we present the formalization of signal-flow-graphs and Mason's gain formula in higher-order logic. We also briefly describe one potential application of our work which is add-drop filter.

1 Introduction

In order to design physical or engineering systems, we need to analyze their behavior and to ensure that our design satisfies the system properties and specifications. To analyze those behaviours, we need first to obtain the system transfer function which relates the system input and output. Therefore based on the transfer function we will be able to investigate the system properties, such as stability for physical systems, which ensures that the output is bounded whenever the input in bounded. Generally those studies are curried by some tedious, time-consuming, and error-proneness methods, such as transfer matrix method or the successive substitutions of simultaneous equations. Particularly for large number of components and interconnections in some engineering systems such as space rocket, biomedical systems, etc. But with the simplicity of signal-flow-graphs (proposed by Mason in 1953 [9]) which is widely applied to linear engineering system analysis and design, we can overcome those limitations. A signal-flow-graph (SFG) is a block diagram that represents a set of simultaneous linear algebraic equations, the signal-flow-graphs depicts the flow of the signals from one point of a system to another and gives the relation-ships among the signals. The main benefit of using SFG is Mason's gain formula [9, 8] to determine the system transfer function, which can be applied on any linear signal-flow-graphs. Mason's gain formula (MGF) can easily handle the system transfer function and avoid the linear algebraic computations of simultaneous equations and reduce the work to the computation of forward paths and feedback loops. MGF comes up often in the context of control systems [11] and digital filters, because control systems and digital control systems are often represented by SFG [3].

An automated approach is to use well known computational tools such as MATLAB [7], which provides the facility to obtain transfer functions for an arbitrary systems (e.g., the program available at [4]). However, this tool is not efficient and only provide the facility to handle the systems with constant parameters and the generated output is string of characters; which is a complex-valued function. Theses mentioned inaccuracy problems of traditional analysis techniques are preventing their usage in designing safety-critical engineering systems, where minor bugs can lead to fatal consequences both in terms of cost and human safety. In order to build high assurance engineering systems, it is indispensable to develop a framework which is both accurate and scalable for analyzing complex engineering systems.

Formal methods [13] based techniques have the potential to improve the analysis accuracy of engineering systems. Particulary higher-order logic theorem proving [6] does not exhibit the above limitations, thus we propose a higher-order logic formalization of SFG. To the best of our knowledge, SFG theory and MGF have not been formalized. Our formalization is mainly based on existing theories of multivariate analysis in the HOL Light theorem prover [5]. In order to demonstrate the practical effectiveness of our formalization, we present a potential application for our work.

In this report we present our formalization of transfer function using SFG and MGF in HOL Light theorem prover. The report is organized as follows: Section 2 describes some fundamentals of signal-flow-graphs. Section 3 presents HOL Light theorem prover and some HOL Light symbol and functions. In Section 4 we present in details our elementary and forward algorithm, also in Section 5 we present the idea to compute the transfer function using Mason's gain formula. We give some preliminaries of our formalization in Section 6 which will be used in the next sections. Sections 7 and 8 describe our HOL formalization of elementary and forward algorithm and Mason's gain formula, respectively. In Section 9

we give some properties of our SFG formalization. In order to demonstrate the practical effectiveness and the use of our work, we present in Section 10 a potential application of our work, which is add-drop filter [10]. Finally, Section 11 concludes the report.

2 Signal-Flow-Graphs

A signal-flow-graphs (SFG) is a special type of block diagram and directed graph, consisting of nodes and branches, which connect at nodes. Its nodes are the variables of a set of linear algebraic relations. For example a branch jk originates at node j and terminates upon node k; its direction is indicated by an arrowhead.

The SFG may be interpreted as a signal transmission system in which each node is a tiny repeater station [9]. The station receives signals via the incoming branches, combines the information, and then transmits the result along each outgoing branch. If the nodes of a flow graph are numbered in a chosen order from 1 to n, then we will be able to specify the forward path, as any path along which the sequence of node numbers is increasing, and a backward path as one along which the numbers decrease [9]. A SFG can only represent multiplications and additions. Multiplications are represented by the weights of the branches; additions are represented by multiple branches going into one node [8].

2.1 Some Properties of Signal-Flow-Graphs

Source : A source is a node having only outgoing branches.
Sink : A sink is a node having only incoming branches.
Path : A path is any continuous succession of branches traversed in the indicated branch directions.
Forward Paths : A forward path is a path in SFG that connects the input to the output without touching any single node or path more than once. A single system can have multiple forward paths.
Loops : A loop is a structure in SFG that leads back to itself. A loop does not contain the beginning and ending points, and the end of the loop is the same node as the beginning of a loop. The loops are touching if they share a node or a line in common. The loop gain is the total gain of the loop, as we travel from one point, around the loop, back to the starting point.

2.2 Mason's Gain Formula

Mason's gain formula [2] is a method for finding the transfer function which relates the input and output of the corresponding system of a linear signal-flow-graphs. The formula was proposed by Samuel Jefferson Mason in 1953 [9], whom it is also named after. MGF is an alternate method to find the transfer function algebraically by labeling each signal, writing down the equation for how that signal depends on other signals, and then solving the multiple equations for the output signal in terms of the input signal. MGF provides a step by step method to obtain the transfer function from a SFG. Often, MGF can be determined by inspection of the SFG. The method can easily handle SFG with many variables and loops including loops with inner loops. MGF is often used in the context of control systems and

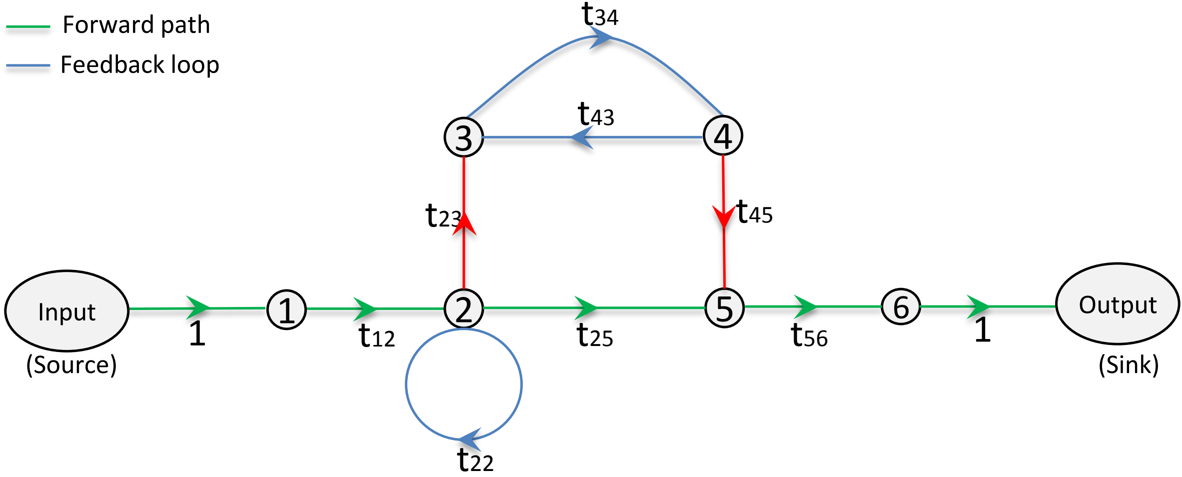

Fig. 1: An example of signal-flow-graphs

digital filters. The formula is given as follows :

$$MasonGain = \frac{Output\ node}{Input\ node} = \sum_{k=1}^{N} \frac{G_k \Delta_k}{\Delta} \tag{1}$$

$$\Delta = 1 - \sum L_i + \sum L_i L_j - \sum L_i L_j L_k + \ldots + (-1)^m \sum \ldots + \ldots$$

where Δ represents the determinant of the graph which has N forwards paths, y_{in} and y_{out} represent the input and output node variables. The parameters *Mason Gain* and G_k describe the complete gain between *Input node* and *Output node* and the gain of the kth forward path, respectively. The Loop gain of each closed loop is represented by L_i whereas the product of the loop gains of any two non-touching loops is given by $L_i L_j$ and this process continues for all non-touching loops in SFG. Finally, Δ_k represents the cofactor value of Δ for the kth forward path considering the elimination of all loops touching the kth forward path.

3 HOL Light Theorem Prover

HOL Light is an interactive high order theorem prover and member of the HOL theorem prover family, developed by John Harrison in 1996 at the University of Cambridge. HOL light conducts proofs formally in higher order logic, HOL light has been successfully used as a verification framework for both software and hardware verification as well as a platform for the formalization of pure mathematical theorems, HOL light also provides a number of automated tools and pre-proved theorems (e.g. arithmetic, set and list theory, and real and complex analysis). HOL light is written in the functional programming language Objective CAML [5](OCaml). So installing OCaml is required for using HOL Light.

3.1 HOL Light Symbols

Table 1 provides the mathematical interpretations of some HOL Light notations that will be used in this report.

HOL Symbol	Standard Symbol	Meaning
/\	and	Logical *and*
\/	or	Logical *or*
~	not	Logical *negation*
`T`	true	Logical true value
`F`	false	Logical false value
==>	$\longrightarrow$	Implication
<=>	$=$	Equality
`!x.t`	$\forall x.t$	for all $x : t$
`?x.t`	$\exists x.t$	for some $x : t$
`@x.t`	$\epsilon x.t$	an x such that : t
`λx.t`	$\lambda x.t$	Function that maps x to $t(x)$
`num`	$\{0, 1, 2, \ldots\}$	Positive Integers data type
`real`	All Real numbers	Real data type
`complex`	All complex numbers	Complex data type
`suc n`	$(n + 1)$	Successor of natural number
`abs x`	$\|x\|$	Absolute function
`&a`	$\mathbb{N} \rightarrow \mathbb{R}$	Typecasting from Integers to Reals
`EL k L`	$L[k]$	The kth element of list L
`CONS e L`	*cons*	Add the element e to the list L
`APPEND l1 l2`	*append*	Joins two lists together
`HD L`	*head*	Return the head of the list L
`TL L`	*tail*	tail of list L
`LENGTH L`	*length*	Length of list L
`(a, b)`	$a \times b$	A pair of two elements
`FST`	$fst(a, b) = a$	First component of a pair
`SND`	$snd(a, b) = b$	Second component of a pair
`{x|P(x)}`	$\{x\|P(x)\}$	Set of all x such that $P(x)$

Table 1: HOL Light Symbols and Functions

4 Elementary & Forward Algorithms

In this section we describe in details the algorithms for extraction the elementary and forward circuit (EC & FC algorithm) from the signal-flow-graphs using a common algorithm [12]. The algorithms are named, EC for "elementary circuit",and FC for "forward circuit". The function blocks of EC are named `EC_INITIALIZATION`, `EC_SEARCH`, etc. And of FC, `FC_VALIDATION`, `FC_MODIFICATION`, etc. In our algorithms we assigned to the nodes of the graph by order, the integer designators 1, 2, ..., n. The algorithms utilize three principal arrays: The first is a two-dimension-array that describes graph G, for which each row i contains all the nodes with which there is a branch starting from i. The second is a one-dimension-array P which contains the nodes of the present path under consideration. The third is a two-dimension-array H, in which each row i contains all the nodes which are closed to the node i (which means there is no loop or forward path starting from i and ending in the same node.)

4.1 Algorithm Description

In `EC_BUILDING`, we build the matrix G based on the list of branches given as input to the algorithm. `EC_INITIALIZATION` initializes the EC or FC algorithms parameters. In `EC_SEARCH`, we start the constitution of new path starting by the first node which will be in the first iteration of the algorithm node 1.

```
Data: L:path
Result: H:matrix of integer
for i = 1 to n do
    k = 0;
    for j = 1 to n do
        if L[j,1] = i then
            G[i,k] = L[j,3];
            k = k + 1;
        end
    end
end
```

Algorithm 1: `EC_BUILDING`

```
Data: n:integer
Result: H:matrix of integer , P:list of integer, k:integer
P = 0;
H = 0;
k = 1;
P[1] = 1;
```

Algorithm 2: `EC_INITIALIZATION`

The path will be extended from its end, with three conditions checked before tentative extension is performed:

- The extension node can not be in P.
- The extension node designation value should be larger than that of the previous node.
- The extension node can not be closed node to the first node of the path under consideration, which means it is not in H.

First condition assures that a path (elementary, or forward) is being formed. Second condition assures that each circuit will only be considered once. The search for a particular circuit will only occur when its lowest valued integer is at the path beginning. Third condition assures that no circuit is considered more than once.

```
Data: P:list of integer, k:integer
Result: G,H:Matrix of integer , P:list of integer, k:integer
for j = 1 to n do
    if ((G[P[k],j] > P[1]) ∧ (G[P[k],j] ∉ H[P[k]]) ∧ (G[P[k],j] ∉ P)) then
        k = k + 1;
        P[k] = G[P[k],j];
        Go to EC_SEARCH;
    end
end
```

Algorithm 3: `EC_SEARCH`

At some points no nodes will be available for extension. A circuit confirmation is then performed, i.e., is there an arc connecting the last node of P to the first node ? (`EC_VALIDATION`, for elementary circuit) or is the last node of P is the sink node of the graph ? (`FC_VALIDATION`, for forward circuit) if it does exist, a circuit is reported.

Data: Loops:matrix ($m_1 \times m_2$) of integer
Result: G:matrix of integer , P:list of integer, k:integer, Loops:matrix ($m_1 \times m_2$) of integer
if *(P[1] $\notin$ G[P[k]])* **then**
  Go to `EC_MODIFICATION` ;
else
  ADD P to Loops;
  Go to `EC_MODIFICATION` ;
end

Algorithm 4: `EC_VALIDATION`

Data: P:list of integer , k,n :integer
Result: H:matrix of integer , P:list of integer, k:integer, Loops:matrix (m1×m2) of integer
if *(stop $\notin$ G[P[k]])* **then**
  Go to `FC_MODIFICATION` ;
else
  ADD P to Forwards;
  Go to `FC_MODIFICATION` ;
end

Algorithm 5: `FC_VALIDATION`

In any case, node closure occurs unless there is only one node in the path. The node closure process consists of three steps (`EC_MODIFICATION` for elementary circuit and `FC_MODIFICATION` for forward circuit):

- Enter the last node of P into the row in H for the previous node of the last node in P.
- Clear the row in H for the last node.
- Shorten P by one arc eliminating the last node.

The first step assures that the path extension just performed will not be repeated. The second step allows correct forward continuation from the last node if it is reached by a

different path in the next iterations.

```
Data: P:list of integer, k:integer
Result: G, H:matrix of integer, P:list of integer, k:integer
if k = 1 then
    Go to EC_EXTEND ;
else
    for j = 1 to n do
        if H[P[k-1],j] = 0 then
            H[P[k-1],j] = P[k];
            exit;
        end
    end
    P[k] = 0;
    k = k - 1;
    Go to EC_SEARCH;
end
```

Algorithm 6: `EC_MODIFICATION`

```
Data: P:list of integer, k:integer
Result: G, H:matrix of integer, P:list of integer, k:integer
if k=1 then
    Go to FC_REWRITING;
else
    for j=1 to n do
        if H[P[k-1],j] = 0 then
            H[P[k-1],j] = P[k];
            exit;
        end
    end
    P[k] = 0;
    k = k - 1;
    Go to EC_SEARCH;
end
```

Algorithm 7: `FC_MODIFICATION`

The above extension process continues until eventually the path has been backed down to one node. At this point an initial node advancement occurs (`EC_EXTEND`, only for elementary circuit, the forward circuit algorithm ends here). In this step, the first node is incremented by one, H is cleared and the extension process starts anew. Now, however, no paths, and thus circuit containing node 1 will be considered. EC continues to extend paths and, on occasion, advance the initial node until P contains a path of one node, node n. Then we will move to the next step of EC & FC algorithm, which is rewriting the output in the format of branches like the format of the input (which is path) the output will be a list of paths, each under form of a list of branches.

```
Data: P:list of integer, n, k:integer
Result: H:matrix of integer, P:list of integer, k:integer, Loops:matrix (m1×m2) of
        integer
if P[1] = n then
|  Go to EC_REWRITING;
else
|  P[1] = P[1] + 1;
|  k = 1;
|  H = 0;
|  Go to EC_SEARCH;
end
```

Algorithm 8: EC_EXTEND

The second part of EC & FC algorithm (EC_REWRITING for elementary circuit, and FC_REWRITING for forward circuit) is to write the reported circuits in the format that each circuit is a list of branches, each is constituted of the start node, the gain, and the end node. Those functions utilize a list of branches which representing the graph, and a list of circuits (forward or elementary) given by the first part of the algorithm.

```
Data: Loops:matrix (m1×m2) of integer, SFG:matrix (nb×3)
Result: Loops_path:matrix
for i = 1 to m1 do
|  for j = 1 to m2-1 do
|  |  for k = 1 to nb do
|  |  |  if (Loops[i,j] = SFG[k,1]) ∧ (Loops[i,j+1] = SFG[k,3]) then
|  |  |  |  path[j,1] = SFG[k,1];
|  |  |  |  path[j,2] = SFG[k,2];
|  |  |  |  path[j,3] = SFG[k,3];
|  |  |  end
|  |  end
|  end
|  for k = 1 to nb do
|  |  if (Loops[i,n] = SFG[k,1]) ∧ (Loops[i,1] = SFG[k,3]) then
|  |  |  path[n,1] = SFG[k,1];
|  |  |  path[n,2] = SFG[k,2];
|  |  |  path[n,3] = SFG[k,3];
|  |  end
|  end
|  Loops_path[i] = path;
end
```

Algorithm 9: EC_REWRITING

Data: Forwards:matrix ($m_1 \times m_2$) of integer, SFG:matrix (nb×3)
Result: Forwards_path:matrix
for *i = 1 to* m_1 **do**
  for *j = 1 to* m_2*-1* **do**
    for *k = 1 to nb* **do**
      if *(Forwards[i,j] = SFG[k,1])* ∧ *(Forwards[i,j+1] = SFG[k,3])* **then**
        path[j,1] = SFG[k,1];
        path[j,2] = SFG[k,2];
        path[j,3] = SFG[k,3];
      end
    end
  end
  Forwards_path[i] = path;
end

Algorithm 10: `FC_REWRITING`

5 Mason's Gain Formula Algorithm

In order to implement the Mason's gain formula and to overcome the challenges, such as the number of loops and forward paths can be infinite, and also finding no touching loops, we implement the fowling procedures:

1. A list of all the forward paths and their gains
2. A list of all the loops and their gains
3. A list of list for non touching loops for each loop
4. A list of list as in 3 but only for the no touching loops with the forward path
5. Computing the determinant Δ and the cofactor Δ_k
6. Applying the formula

6 Preliminaries

In order to simplify our formalization of SFG and to make it more meaningful we defined three types of abbreviation: the first one is : branch which is composed of the start node (integer designation), the gain (complex), and the end node (integer designation). The second one is a path, which is a list of branches. The third one is : SFG which is composed of a list of branches, the size (integer), the sink node (integer), and the source node (integer).

Definition 1 (Type of Abbreviations).

```
new_type_abbrev ("branch", `:ℕ × ℂ × ℕ`)
new_type_abbrev ("path", `:(branch)list`)
new_type_abbrev ("SFG", `:(branch)list × ℕ × ℕ × ℕ`)
```

Definition 2 (Are Two Paths Touching).
```
⊢ ∀ (t1:path) (t2:path).
  IS_TOUCHING [ ] [ ] = F ∧
  IS_TOUCHING [ ] t1 = F ∧
  IS_TOUCHING t2 [ ] = F ∧
  IS_TOUCHING (CONS a t1) (CONS b t2) = if ((RIGHT (CONS a t1) b) ∨
   (LEFT a (CONS b t2))) then T else IS_TOUCHING t1 t2
```

The above function verifies if two paths are touching or not, two paths are touching if they have one common node or more. `IS_TOUCHING` accepts as parameters two paths (list of branches), and takes in each iteration one branch from each path and verifies if that branch does touch the other path. If the branch is from the path in the left, `IS_TOUCHING` calls the function `LEFT` [1], otherwise if the branch from the list in the right calls the function `RIGHT` [1]. By induction the function will go through the both given paths.

Definition 3 (Gain).
```
⊢ ∀ (t:path).
  GAIN [ ] = Cx(&0)
  GAIN (CONS h t) =
     if (t = [ ]) then snd_of_trpl h else (snd_of_trpl h) * (GAIN t)
```

The `GAIN` function calculates the gain of path, which in the case of empty path it returns 0, otherwise it returns the product of the transmittances of all the branches in the path. We used snd_of_trpl [1] to extract the second component of the branch which is the transmittance.

Definition 4 (Valid Path).
```
⊢ ∀ (l:path).
  IS_VALID_PATH (CONS e l) = if (l=[ ]) then T
  else (if((lst_of_trpl e) = (fst_of_trpl (HD l)))
  then (T ∧ (IS_VALID_PATH l)) else F)‘;;
```

To ensure that a path is a valid path which means that all the branches in the path are connected to each other. We define the function `IS_VALID_PATH` to fulfill this requirement. Where fst_of_trpl and last_of_trpl extract the first and third element of a triplet, respectively. The function `IS_VALID_PATH` accepts a path (which is a list of branches) and returns true (T) if last components of each i branch is equal to the first element of corresponding ($i + 1$) branch in the path.

7 Formalization of Elementary & Forward Circuit Algorithms

In this section, we give some definitions of our formalization of the elementary and forward circuits extraction from a given SFG. We start our formalization by defining a new type `KPH` which is a type of data we will need in our formalization, and represents the triple (k, P, H), as we described in the EC & FC algorithm in Section 4.

Definition 5 (Types of Abbreviations).
```
    new_type_abbrev ("KPH", ‘:ℕ × (ℕ list) × ((ℕ list) list)‘)
```

Definition 6 (Generates Matrix G).
```
⊢ ∀ (k:ℕ) (t1:ℕ list) (t2:ℕ list) (t3:ℕ list).
```

```
EC_G_creation k t1 t2 [ ] = [ ] ∧
EC_G_creation k (CONS e1 t1) (CONS e2 t2) (CONS e3 t3) =
(CONS (EC_G_creation_aid (k+1) (CONS e1 t1) (CONS e2 t2))
(EC_G_creation (k + 1) (CONS e1 t1) (CONS e2 t2) t3))
```

The matrix G here is represented as list of list and has the same size of the graph, each list i of G is the list of nodes where there are direct branches from the node i to those nodes. `EC_G_creation` takes in each iteration a node from the graph and calls the function `EC_G_creation_aid` which in each iteration searches in the list of start nodes (second parameter of the main function) if there is matching with the given node, it returns the element of the list of end nodes (third parameter of the main function) which is in the same place as the one from the list of start nodes, and by induction the `EC_G_creation_aid` will go through the list of start nodes, also by induction the `EC_G_creation` will go through all the nodes of the graph.

Definition 7 (Modifying Matrix H).

```
⊢ ∀ (e1:ℕ) (H:(ℕ list) list) (k1:ℕ) (k2:ℕ).
  EC_H_MODIFIE e1 H k1 k2 = EC_H_MODIFIE2 (EC_H_MODIFIE1 e1 H k1) k2 (LENGTH H)
```

When the path attains the closed node, matrix H, in which each row i contents the closed nodes of the node i in the graph, is modified as in `EC_MODIFICATION` and `FC_MODIFICATION` process explained above. The function `EC_H_MODIFIE1` with the help of the function `EC_H_MODIFIE0`, puts the closed node of the path under consideration as closed node for the previous node in the same path. The function `EC_H_MODIFIE2` initializes the row of the closed node to zero as in the process explained before. Thus the function `EC_H_MODIFIE` uses all those functions.

Definition 8 (Path Searching).

```
⊢ ∀ (n:ℕ) (m:ℕ) (t1:ℕ list) (t2:ℕ list) (l1:ℕ list) (l2:ℕ list).
  EC_next_node n m [ ] t2 l1 l2 e t = 0 ∧
  EC_next_node n (m + 1) (CONS e1 t1) (CONS e2 t2) l1 l2 =
  EC_next_node n m (CONS e1 t1) t2 l1 l2 ∧
  EC_next_node n 0 (CONS e1 t1) (CONS e2 t2) l1 l2 =
  if((e1 > HD l1) ∧ (NOT_IN_LIST e1 l1) ∧
 (NOT_IN_LIST e1 l2)) then (n+1) else
  EC_next_node (n+1) 0 t1 (CONS e2 t2) l1 l2
```

```
⊢ ∀ (k:ℕ) (P: ℕ list) (H: (ℕ list) list) (G: (ℕ list) list).
  EC_next_node_main k P H G =
  EC_next_node 0 k (EL (minus_one (EL k P)) G) P P
  (EL (minus_one (EL k P)) H)
```

The function `EC_next_node` is the core for searching to paths inside the graph, it verifies if the next node under consideration of a path does verify the three conditions explained in the `EC_SEARCH` procedure in the Elementary & Forward Algorithm Section 4. All the graph nodes are under consideration as the next node in the searching of path. After that the path will be extended until the closed node and path will be reported to the validation process to validate it.

Definition 9 (Path Searching Main).

```
⊢ ∀ (k:ℕ) (P:ℕ list) (H:(ℕ list) list) (G:(ℕ list) list) (j:ℕ).
```

```
EC_search_path k P H G j =
if(¬(j = 0)) then EC_search_path (k+1) (EC_replace_first_zero (k + 1)
(EL (minus_one j) (EL (minus_one (EL k P)) G)) P) H G
(EC_next_node_main (k + 1) (EC_replace_first_zero (k + 1)
(EL (minus_one j) (EL (minus_one (EL k P)) G)) P) H G) else (if(¬(k = 0))
then (k,P,(EC_H_MODIFIE (EL k P) H (minus_one (EL (minus_one k) P))
(minus_one (EL k P)) )) else (k,P,H))
```

To validate the result found by `EC_next_node_main` and to do the necessary modification for the next search parameters (matrix H and vector P), we use `EC_search_path` in which the two first parameters represent the path under consideration and its size, respectively. The next two parameters are matrix H for closed nodes and matrix G of the graph description, respectively.

Definition 10 (Searching All EC for Node).

```
⊢ ∀ (l:KPH) (G:(ℕ list) list) (t:ℕ list).
  EC_POINT l G [ ] = [ ] ∧
  EC_POINT l G (CONS e t) =
  if((fst_of_trpl l = 0) ∧ (¬(EC_TEST (snd_of_trpl l) (fst_of_trpl l) G)))
  then [ ] else (APPEND (if(EC_TEST (snd_of_trpl l) (fst_of_trpl l) G)
  then [(EC_add_node (snd_of_trpl l))] else [ ])
  (EC_POINT (EC_POINT_AID l G) G t3))
```

The function `EC_POINT` reports all the elementary circuits which start from a single node of the graph. If the path reported by `EC_search_path` is a valid loop, by testing it with the function `EC_TEST` [1], an elementary circuit will be reported and `EC_POINT` goes to the next iteration of the search by calling `EC_POINT_AID` [1] to update the search parameters. `EC_POINT_AID` makes call to the function `EC_search_path` to update the search parameters and to start new search as explained above.

Definition 11 (Skip Node).

```
⊢ ∀ (p:ℕ) (G:(ℕ list) list) (n:ℕ) (m:ℕ).
  ECSKIP p G m n = if(m=n) then F else if(IN_LIST p (EL m G)) then T
  else (ECSKIP p G (m+1) n)
```

For some nodes in the graph all the branches are outgoing (there is no incoming branches). From the definition of loops, we can not find a loop which contains a node which does not have incoming branch. So in order to enhance the performance of our algorithm, we define the function `ECSKIP` which skips the process of finding loops for each node of the graph which does not have an incoming branch.

Definition 12 (Searching All EC for All Nodes).

```
⊢ ∀ (k:ℕ) (P:ℕ list) (H:(ℕ list) list) (G:(ℕ list) list)
   (size:ℕ) (n:ℕ) (t:ℕ list).
  EC_ALL_POINTS k P H G size n [ ] = [ ] ∧
  EC_ALL_POINTS k P H G size n (CONS e t) = if(EECTEST P size)
  then [ ] else if(ECSKIP (EL 0 P) G (minus_one (EL 0 P)) n) then
  (APPEND (EC_POINT k P H G (EC_next_node_main k P H G)) G (REPLICATE (m*m) 0)
  (EC_ALL_POINTS k (EC_add_one_to_list_head P) H G n (minus_one m) t)) else
  (EC_ALL_POINT k (EC_add_one_to_list_head P) H G n (minus_one m) t)
```

As the function EC_POINT did report all the elementary circuits which start from a single node of the graph, the EC_ALL_POINTS calls in each iteration the function EC_POINT for each node of the graph by order, so to make sure that all the graph elementary circuits are reported. The end of this function process will be when the head of the vector P is equal to the graph size (EECTEST [1]).

Definition 13 (Feedback Paths, Output: List of Nodes).

```
⊢ ∀ (l:path) (size:ℕ).
  EC_MAIN l size =
  EC_ALL_POINTS 0 (EC_replace_vector_head (EC_create_vector size) 1)
  (EC_create_matrix size) (EC_G_creation 0
  (EC_start_nodes l) (EC_end_nodes l)
  (EC_create_vector size) size size (EC_create_vector size))
```

EC_MAIN function, which is the main function for elementary circuit algorithm, which starts by initializing the vector P and matrixes H and G (search parameters), thereafter it uses the function EC_ALL_POINTS to report all the graph's elementary circuits. EC_MAIN accepts as parameters the graph representation (as list of branches) and the graph size.

Definition 14 (Rewriting Feedback Paths Output).

```
⊢ ∀ (p:(ℕ list) list) (l:path).
  EC_REWRITING [ ] l = [ ] ∧
  EC_REWRITING (CONS t p) l = (CONS (EC_REWRITING0 t (EC_REWRITING1 (HD t) t l)
   (HD t)) (EC_REWRITING p l))
```

The EC_REWRITING function takes all the elementary circuits as list of nodes and returns them as list of branches (more meaningful form). In its process EC_REWRITING calls in each iteration two functions. First one, EC_REWRITING1 [1] which reports the list of gains of the circuit under consideration. The second function is EC_REWRITING0 [1] which takes each circuit in format of list nodes and returns it in list of branches with the help of the previous function. EC_REWRITING has two parameters, the list of elementary circuits and the graph representation (list of branches).

Definition 15 (Feedback Paths).

```
⊢ ∀ (l:path) (size:ℕ).
  EC l size = if(l=[ ]) then [ ] else EC_REWRITING (EC_MAIN l n) l
```

Above is EC, the main function which returns all the elementary circuits returned by EC_MAIN, in format of list of nodes, in format of list of branches (more meaningful) and that with the help of EC_REWRITING. It takes two parameters: the graph representation and its size, respectively.

Definition 16 (Finding Next Forward Path).

```
⊢ ∀ (k:ℕ) (P:ℕ list) (H:(ℕ list) list) (G:(ℕ list) list).
  FC_next_node_main k P H G = EC_next_node 0 0 (EL (minus_one (EL k P)) G)
  P P (EL (minus_one (EL k P)) H)
```

The next function is the core of searching for forward circuits inside the graph by using the function EC_next_node explained before. the difference between this function and EC_next_node_main is in the parameters of searching, this function is part of FC_SEARCH and FC_VALIDATION process in Section 4 . The first two parameters are the size of the path under

consideration and the path under consideration (list P), respectively. The next two parameters are the matrix H (the closed nodes matrix) and the matrix G (the graph description matrix).

Definition 17 (Removing Node From Search).

```
⊢ ∀ (k:ℕ) (P:ℕ list) (H:(ℕ list) list) (G:(ℕ list) list) (j:ℕ).
  FC_search_path k P H G j =
   if(¬(j = 0)) then (FC_search_path (k + 1)
   (EC_replace_first_zero(k + 1) (EL (minus_one j)
   (EL (minus_one (EL k P)) G)) P) H G (FC_next_node_main (k + 1)
   (EC_replace_first_zero(k + 1) (EL (minus_one j)
   (EL (minus_one (EL k P)) G)) P) H G)) else (if(¬(j = 0)) then
   (k,P,(EC_H_MODIFIE (EL k P) H (minus_one (EL (minus_one k) P))
   (minus_one (EL k P)))) else (k,P,H))
```

The `FC_search_path` function searches for forward circuits and reports the results given by `FC_next_node_main` to the validation process also it calls `EC_H_MODIFIE` to do the necessary modification to matrix H. Which is the main process of `EC_SEARCH` and `FC_VALIDATION`. The first four parameters are the size of the path under consideration, the path under consideration (list P), matrix H (the closed nodes matrix), and matrix G (the graph description matrix).

Definition 18 (Forward Paths, Output: List of Nodes (1/2)).

```
⊢ ∀ (l:KPH) (G:(ℕ list) list) (t:ℕ list) (stop:ℕ).
  FC_FORWARD l G [ ] stop = [ ] ∧
  FC_FORWARD l G (CONS e t) stop =
  if((fst_of_trpl l = 0) ∧ (¬(FCTEST (snd_of_trpl l) (fst_of_trpl l) stop)))
  then [ ] else (APPEND (if(FCTEST (snd_of_trpl l) (fst_of_trpl l) stop)
  then [(EC_add_node (snd_of_trpl l))] else [ ])
  (FC_FORWARD (FC_POINT_AID l G) G t stop))

⊢ ∀ (k:ℕ) (P:ℕ list) (H:(ℕ list) list) (G:(ℕ list) list)
  (stop:ℕ) (size:ℕ).
  FORWARD_CIRCUIT k P H G stop size = FC_FORWARD (FC_search_path k P H G
  (FC_next_node_main k P H G)) G (REPLICATE (size∗size) 0) stop
```

The function `FC_FORWARD` reports all the forward circuits of the graph. If the path reported by `FC_search_path` is validated by `FCTEST` [1], a forward circuit is reported and `FC_FORWARD` goes to the next iteration of search by using `FC_POINT_AID` [1] to update the search parameters. `FC_POINT_AID` uses the function `FC_search_path` to update the search parameters and to start new search as explained above.

Definition 19 (Forward Paths, Output: List of Nodes (2/2)).

```
⊢ ∀ (l:path) (size:ℕ) (stop:ℕ) (start:ℕ).
  FC_MAIN l size stop start =
  FORWARD_CIRCUIT 0 (EC_replace_vector_head (EC_create_vector size) start)
  (EC_create_matrix size) (EC_G_creation 0
  (EC_start_nodes l)
  (EC_end_nodes l) (EC_create_vector size)) stop size
```

FC_MAIN is the main function for forward circuit algorithm. Which starts by initializing the vector P and matrixes H and G (search parameters), then it calls the function FORWARD_CIRCUIT to report all the graph's forward circuits.

Definition 20 (Rewriting Forward Paths Output).

```
⊢ ∀ (p:(ℕ list) list) (l:path).
  FC_REWRITING [ ] l = [ ] ∧
  FC_REWRITING (CONS t p) l = (CONS (FC_REWRITING1 t (FC_REWRITING0 t l))
  (FC_REWRITING p l))
```

The FC_REWRITING function takes the forward circuits as list of nodes and returns them as list of branches in more meaningful forms. In its process FC_REWRITING uses in each iteration two functions. First one, FC_REWRITING0 [1], which reports the list of gains of the circuit under consideration. The second function is FC_REWRITING1 [1], which takes each circuit in format of list nodes and returns it in list of branches with the help of the previous function. The difference between FC_REWRITING and EC_REWRITING is that in elementary circuit the number of branches is equal to the number of nodes but in forward circuit it is equal to the number of nodes minus one. The plus branch in elementary circuit is the feedback branch from the last node to the start node.

Definition 21 (Main Forward Paths).

```
⊢ ∀ (l:path) (size:ℕ) (stop:ℕ) (start:ℕ).
  FC l size stop start = if(l = [ ]) then [ ] else
  FC_REWRITING (FC_MAIN l size stop start) l
```

FC, the main function which returns all the forward circuits returned by FC_MAIN (in format of list of nodes) in format of list of branches more meaningful and that with the help of FC_REWRITING. It takes four parameters, the graph representation (list of branches), the graph's size, and the graph's sink and source nodes, respectively.

8 Mason's Gain Formula Formalization

Definition 22 (Lists of Touching Loops).

```
⊢ ∀ (t1:path) (t:(path)list).
  TOUCHING_LOOP t1 [ ] = [ ] ∧
  TOUCHING_LOOP t1 (CONS t2 t) = if (IS_TOUCHING t1 t2)
  then (TOUCHING_LOOP t1 t) else CONS t2 (TOUCHING_LOOP t1 t)
```

```
⊢ ∀ (t:(path)list).
  TOUCHING_LOOP_MAIN [ ] = [ ] ∧
  TOUCHING_LOOP_MAIN (CONS h t) = (CONS (TOUCHING_LOOP h t)
  (TOUCHING_LOOP_MAIN t))
```

The function TOUCHING_LOOP_MAIN calculates for each loop, the list of loops which are; first have bigger rank than the loop under consideration in the list of loops given, second non touching with the loop under consideration. Therefore the function TOUCHING_LOOP does the work for each loop and that by testing each loop with the loops which have bigger rank than the loop under consideration in the list of loops given and that with the function IS_TOUCHING. Then it reports the result to TOUCHING_LOOP_MAIN and then the next iteration for the next loop. TOUCHING_LOOP_MAIN takes as parameter the list of loops in the graph.

Definition 23 (Common Paths).

```
⊢ ∀ (t1:(path)list) (t2:(path)list).
  COMMON_ELES_LISTS [ ] [ ] = [ ] ∧
  COMMON_ELES_LISTS [ ] t1 = [ ] ∧
  COMMON_ELES_LISTS (CONS a t1) t2 = if (LEFT_OF_LIST a t2) then
   (CONS a (COMMON_ELES_LISTS t1 t2))
   else (COMMON_ELES_LISTS t1 t2)
```

The function COMMON_ELES_LISTS extracts the common paths of two given lists of paths, it calls the function LEFT_OF_LIST [1] which takes a path and a list of paths and verifies if that path is an element of the list of paths and that by using ARE_SAME_PATH [1] which takes a path and tests it with each path of the list of paths.

Definition 24 (Products of Loops).

```
⊢ ∀ (t1:path) (t3:(path)list) (t:((path)list)list) (l:(path)list)
  (tl:((path)list)list).
  PRODUCT_THREE_MORE t1 (CONS t2 t3) [ ] l tl = Cx(&0) ∧
  PRODUCT_THREE_MORE t1 [ ] t l tl = Cx(&0) ∧
  PRODUCT_THREE_MORE t1 (CONS t2 t3) (CONS t4 t) l tl =
   if(LISTS_ARE_EMPTY (CONS t4 t))
   then Cx(&0) else ((PRODUCT_THREE_MORE t1 t3 t l tl) -
   ((GAIN t1) × ((GAIN_PATH_LISTPATHS t2 (COMMON_ELES_LISTS t3
   (EL (EXTRACT_PLACE_ELEMENT_OFLIST 0 t2 l) tl))) +
   (PRODUCT_THREE_MORE t2
   (COMMON_ELES_LISTS t4 t3) t l tl))))
```

PRODUCT_THREE_MORE calculates the infinite part of Mason's determinant which contents the product of three or more of non touching loops to a given loop which are already computed by the previous functions we defined.

Definition 25 (Delta Minus One).

```
⊢ ∀ (t2:(path) list) (t:((path) list) list) (tl:((path) list) list).
  DELTA_MINUS_ONE [ ] [ ] l tl = Cx(&0) ∧
  DELTA_MINUS_ONE (CONS t1 t2) [ ] l tl = Cx(&0) ∧
  DELTA_MINUS_ONE [ ] (CONS t3 t) l tl = Cx(&0) ∧
  DELTA_MINUS_ONE (CONS t1 t2) (CONS t3 t) l tl =
   ((GAIN_PATH_LISTPATHS t1 t3) + (PRODUCT_THREE_MORE t1 t3 t l tl)
   + (DELTA_MINUS_ONE t2 t l tl) - (GAIN t1))
```

The above function calculates the determinant of the graph minus one by using in each iteration for each loop; GAIN_PATH_LISTPATHS [1] to calculate the product between each loop and its non touching loops, GAIN to calculate the gain of the loop under consideration, and PRODUCT_THREE_MORE to calculate the product of three and more of non touching loops to the loop under consideration. the first two parameters of DELTA_MINUS_ONE are the list of loops and the list of non touching loops for each loop.

Definition 26 (MGF Denominator).

```
⊢ ∀ (t:(path) list).
  DETERMINANT t =
   Cx(&1) + (DELTA_MINUS_ONE t (TOUCHING_LOOP_MAIN t) t (TOUCHING_LOOP_MAIN t))
```

The function DETERMINANT above calculates the determinant of Mason's gain formula by calling TOUCHING_LOOP_MAIN to compute for each loop its non touching loops then it uses DELTA_MINUS_ONE to calculate the determinant minus one. It takes as parameter the list of loops.

Definition 27 (FP Delta Minus One).

```
⊢ ∀ (t1:path) (t:(path) list).
  FORWARD_DELTA_MINUS t1 t =  (DELTA_MINUS_ONE (TOUCHING_LOOP t1 t)
  (TOUCHING_LOOPS_FORWARD t1 t) (TOUCHING_LOOP t1 t)
  (TOUCHING_LOOPS_FORWARD t1 t))
```

FORWARD_DELTA_MINUS calculates for each forward path its determinant based on the non touching loops to the given forward path and that by using TOUCHING_LOOPS_FORWARD [1] to determine the list of loops and DELTA_MINUS_ONE to calculate the determinant minus one. It accepts as parameters a forward path, and the list of loops.

Definition 28 (MGF Numerator).

```
⊢ ∀ (t1:(path)list) (t2:(path)list).
  PRODUCT_FORWARD_DELTA [ ] t2 = Cx(&0) ∧
  PRODUCT_FORWARD_DELTA (CONS f t1) t2 = (((GAIN f) * (Cx(&1)
  + (FORWARD_DELTA_MINUS f t2))) + (PRODUCT_FORWARD_DELTA t1 t2))
```

PRODUCT_FORWARD_DELTA calculates in each iteration the product of the forward path under consideration and its determinant which is calculated by FORWARD_DELTA_MINUS, and adding the result to the terms which will be calculated in the next iterations. It takes two parameters, the list of forward paths and the list of loops in the graph.

Definition 29 (MGF).

```
⊢ ∀ (l:(path) list) (t:(path) list).
```

$$\texttt{MASON_GAIN l t} = \frac{\texttt{PRODUCT_FORWARD_DELTA l t}}{\texttt{DETERMINANT t}}$$

Above is MASON_GAIN, the main function of MGF formalization which returns the transfer function as in MGF for a given list of feedback loops and forward paths, and that by calling DETERMINANT to calculate the determinant of the graph and PRODUCT_FORWARD_DELTA to calculate the numerator.

Definition 30 (Transfer Function (TF) (1/2)).

```
⊢ ∀ (sfg:SFG) (P:ℕ list) (H:(ℕ list) list) (G:(ℕ list) list).
  SFG_Main_AID sfg P H G = MASON_GAIN (FC_REWRITING
  (FORWARD_CIRCUIT 0 (EC_replace_vector_head P (lst_of_four sfg))
  H G (thd_of_four sfg) (snd_of_four sfg)) (fst_of_four sfg))
  (EC_REWRITING (EC_ALL_POINT 0 (EC_replace_vector_head P 1)
  H G (snd_of_four sfg) (snd_of_four sfg) P) (fst_of_four sfg))
```

Definition 31 (Transfer Function (TF) (2/2)).

```
⊢ ∀ (sfg:SFG).
  SFG_Main sfg = if((fst_of_four sfg)=[]) then Cx(&0) else
  (SFG_Main_AID sfg (EC_create_vector (snd_of_four sfg))
  (EC_create_matrix (snd_of_four sfg))
  (EC_G_creation 0 (EC_start_nodes (fst_of_four sfg))
  (EC_end_nodes (fst_of_four sfg)) (EC_create_vector (snd_of_four sfg))))
```

SFG_Main, the main function of our formalization of SFG theory, which combines forward and elementary circuits algorithms and Mason's gain formula algorithm to calculate the transfer function of a given graph. SFG_Main initializes the matrix G and H, and the vector P, next it uses SFG_Main_AID in order to compute the transfer function of the given graph (sfg).

Definition 32 (Numerator of TF (1/2)).

```
⊢ ∀ (sfg:SFG) (P:ℕ list) (H:(ℕ list) list) (G:(ℕ list) list).
  NUMERATOR_AID sfg P H G = PRODUCT_FORWARD_DELTA (FC_REWRITING
  (FORWARD_CIRCUIT 0 (EC_replace_vector_head P (lst_of_four sfg))
  H G (thd_of_four sfg) (snd_of_four sfg)) (fst_of_four sfg))
  (EC_REWRITING (EC_ALL_POINT 0 (EC_replace_vector_head P 1)
  H G (snd_of_four sfg) (snd_of_four sfg) P) (fst_of_four sfg))
```

Definition 33 (Numerator of TF (2/2)).

```
⊢ ∀ (sfg:SFG).
  NUMERATOR sfg = if((fst_of_four sfg)=[]) then Cx(&0) else
  (NUMERATOR_AID sfg (EC_create_vector (snd_of_four sfg))
  (EC_create_matrix (snd_of_four sfg))
  (EC_G_creation 0 (EC_start_nodes (fst_of_four sfg))
  (EC_end_nodes (fst_of_four sfg)) (EC_create_vector (snd_of_four sfg))))
```

The function NUMERATOR calculates the numerator of SFG_Main, first by using EC_create_matrix, EC_G_creation, and EC_create_vector to calculate the matrix H, G, and the vector P, second the function NUMERATOR_AID computes list of feedback paths and forward paths and uses PRODUCT_FORWARD_DELTA to calculate the numerator of Mason's gain formula.

Definition 34 (Denominator of TF).

```
⊢ ∀ (sfg:SFG).
  DENOMINATOR sfg = DETERMINANT (EC (fst_of_four sfg) (snd_of_four sfg))
```

The function DENOMINATOR calculates the numerator of the SFG_Main, using EC to determine the list of feedback loops and DETERMINANT to calculate the determinant of the graph as in Mason's gain formula algorithm.

9 Some Basic Properties of Signal-Flow-Graphs

In this section we give some properties of our SFG formalization in HOL Light, that we have proved.

Theorem 1 (SFG Fundamental Theorem).

⊢ ∀ (sfg:SFG).

$$\texttt{SFG_Main sfg} = \frac{\texttt{NUMERATOR sfg}}{\texttt{DENOMINATOR sfg}}$$

This property proves that the transfer function of SFG (SFG_Main) can be written as fraction of the NUMERATOR and DENOMINATOR defined above of the same SFG. The proof of this theorem is based on rewriting the definitions of SFG_Main, DENOMINATOR, and NUMERATOR. This theorem is very useful for the analysis of physical circuits, because it helps for the extraction of the system zeroes and poles.

Theorem 2 (FC of Empty SFG).
⊢ `FC [ ] 0 0 = [ ]`

The property above proves that in an empty SFG (no branch and node, graph size = 0), there is no forward circuit. The proof is by rewriting the definition of `FC`.

Theorem 3 (FC of SFG Contents Single Branch).
⊢ `∀ (gain:ℂ). FC [(1,gain,2)] 2 3 = [ ]`

The above mentioned property describes that in a given SFG if the given stop node does not match with any node of the end nodes of the graph branches, the graph will not have any forward circuit. Note that the proof is based on some simplifiers (see appendix I) developed for the forward circuits which reduces the interaction with HOL Light during the proof.

Theorem 4 (EC of Empty SFG).
⊢ `EC [ ] 0 = [ ]`

This property proves that in an empty SFG, there is no feedback loop. The proof of this theorem is by rewriting the definition of `EC`.

Theorem 5 (EC of SFG contents single branch).
⊢ `∀ (gain:ℂ). EC [(1,gain,2)] 2 = [ ]`

The property above proves that in given SFG if there is no feedback loop the result will be an empty list. The proof of this theorem is based on some simplifiers [1] developed for the elementary circuits which reduces the interaction with HOL Light during the proof.

Next property proves that however the given list of feedback loops if the list of forward paths is empty, the transfer function will be zero. The proof is based on rewriting the definitions of `MASON_GAIN` and `PRODUCT_FORWARD_DELTA`. When we rewrite `PRODUCT_FORWARD_DELTA` we will get zero, because `PRODUCT_FORWARD_DELTA` is the sum of products of forward path gain and its determinant, so when there is no forward path the first term of each product is zero and the final result therefore is zero.

Theorem 6 (Mason's gain formula Property).
```
⊢ ∀ (l:(path) list).
  MASON_GAIN l [ ] = Cx(&0)
```

10 Add-Drop Filter

10.1 Introduction

As an application of our proposed signal-flow-graphs formalization, we formalize and verify an important photonic processor namely add-drop filter (or add-drop multiplexer) [10] which is widely used as a filtering element in biosensors and wavelength division multiplexing (WDM). Add-drop filter (ADF) is generally used for the construction of optical telecommunication networks. "add" and "drop" refer to the capability of the device to add one or more input wavelength channels to an existing multi-wavelength WDM signal, andor to drop (remove) one or more channels, passing those signals to another network path, respectively. An add-drop filter may be considered to be a specific type of optical cross-connect.

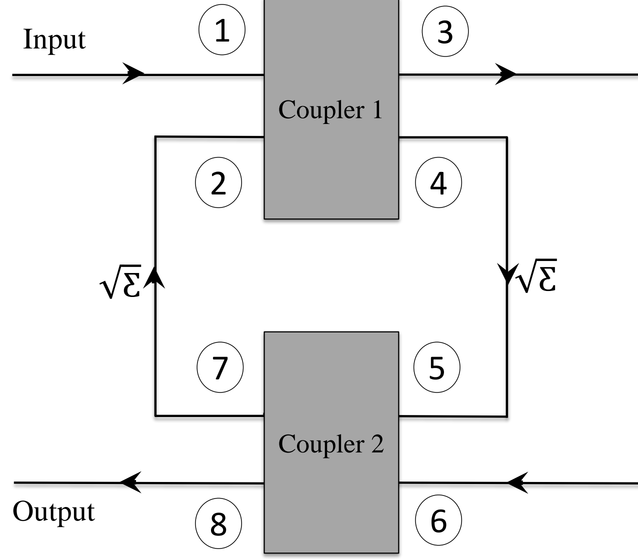

Fig. 2: Add-Drop Filter Schematic Architecture

10.2 Add-Drop Filter Modelling

A traditional add-drop filter consists of three stages: an optical demultiplexer, an optical multiplexer, and between them a method of reconfiguring the paths between the optical demultiplexer, the optical multiplexer and a set of ports for adding and dropping signals. The optical demultiplexer separates wavelengths in an input fiber onto ports. The schematic diagram of the add-drop filter circuit is shown in Figure 2 which consists of two directional couplers interconnected with two optical fiber forward and feedback path. The fiber path ④-⑤ is the forward path of the circuit, while path ⑦-② is the feedback path of the circuit. The signal-flow-graphs representation of the ADF circuit is shown in Figure 3 which consists of the same number of nodes as in the block diagram representation in Figure 2. Our main interest is to evaluate the circuit behavior at the output node which is represented by node ⑧, when the signal is applied at the input, i.e., node ①. We keep all above mentioned parameters in the general form which further can be used to model different ADF configurations.

Next, we formally define the SFG of the ADF in HOL as follows:

Definition 35 (ADF Model).

$\vdash \forall \xi\ \mathtt{S_1}\ \mathtt{S_2}\ \mathtt{C_1}\ \mathtt{C_2}\ \in \mathbb{C}.$

$\mathtt{ADF_model}\ \xi\ \mathtt{S_1}\ \mathtt{S_2}\ \mathtt{C_1}\ \mathtt{C_2}\ =\ [(1, -\mathtt{j} * \mathtt{S1}, 4); (4, \sqrt{\xi}, 5); (5, \mathtt{C_2}, 7); (1, \mathtt{C_1}, 3);$
$(5, -\mathtt{j} * \mathtt{S_2}, 8); (6, \mathtt{C_2}, 8); (6, -\mathtt{j} * \mathtt{S2}, 7); (7, \sqrt{\xi}, 2); (2, \mathtt{C_1}, 4); (2, -\mathtt{j} * \mathtt{S_1}, 3)], 8, 8, 1$

where `ADF_model` accepts complex-valued transmittances and coupling coefficients, and returns the signal-flow-graphs which has a total number of `8` nodes and the output node is ⑧ as shown in Figure 2.
Next, we verify the transfer function of the ADF circuit as follows:

Theorem 7 (Transfer Function of ADF).

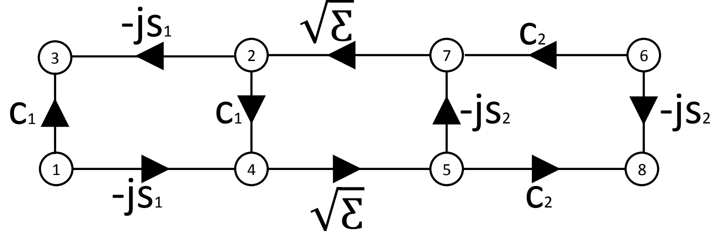

Fig. 3: Signal-Flow-Graph Model of Add-Drop Filter

$\vdash \forall \xi \ \mathtt{S_1} \ \mathtt{S_2} \ \mathtt{C_1} \ \mathtt{C_2} \ \in \mathbb{C}.$

$$\mathtt{transfer_function}\ (\mathtt{ADF_model}\ \xi\ \mathtt{S_1}\ \mathtt{S_2}\ \mathtt{C_1}\ \mathtt{C_2}) = \frac{-\mathtt{S_1} * \mathtt{S_2} * \sqrt{\xi}}{1 - \mathtt{C_1} * \mathtt{C_2} * \xi}$$

11 Conclusions

In this report, we presented the formalization of algorithms for calculating the transfer function using HOL Light theorem prover. In particular, we presented the formalization of the signal-flow-graphs alone with the Mason's gain formula. We also verified some important properties which ensures the correctness of our definitions. Finally, we gave an application of our formalization in the field of photonic signal processors, in order to demonstrate the effectiveness of our proposed formalization of signal-flow-graphs. Note that our proposed formalization can be used in other field other than photonic, such as: command theory, electronic, and digital signal processing.

Appendix I: Proof Tactics

Tactic	Description
SIMPLIFIED_FH_TAC0	It simplifies the function which confirms if a possible feedback path is a real feedback path and simplifies the function which updates the matrix H and the vector P for the next search
SIMPLIFIED_FC_TAC0	It simplifies the function which searches for one possible forward path
TAC_FC	It simplifies the function which computes the entire forward path by applying the previous two tactics randomly
SIMPLIFIED_EC_TAC0	It simplifies the function which searches for one possible feedback path
SIMPLIFIED_H_TAC0	It simplifies the function which confirms if a possible feedback path is a real feedback path and simplifies the function which updates the matrix H and the vector P for the next search
SIMPLIFIED_H_TAC1	It combines the two previous tactics
SIMPLIFIED_ALL_TAC0	It simplifies the function EC_ALL_POINTS in order to change the search starting node
TAC_EC	It simplifies the function which computes the entire feedback paths by applying the previous four tactics randomly
MASON_SIMP_TAC	It simplifies the function which calculates the transfer function using Mason's gain formula (MASON_GAIN)
COMPLEX_DIV_TAC	It simplifies a goal of comparing two fractional expressions
ECFEC_SIMP_TAC2	Simplifies the functions which normalize the output of the feedback path's searching functions
ECFEC_SIMP_TAC1	Simplifies the functions which initialize the matrix G and H and the vector P used for extracting the feedback paths and the forward paths
ECFEC_SIMP_TAC3	Simplifies the functions which normalize the output of the forward path's searching functions
SFG_TAC	This is the main tactics for simplifying the transfer function of the given graph
NUMERATOR_SIMP_TAC	Simplifies the functions which initialize the matrix G and H and the vector P used for extracting the feedback paths and the forward paths for computing the numerator
DENOMINATOR_SIMP_TAC2	Simplifies the functions which normalize the output of the forward path's searching functions for calculating the denominator
DENOMINATOR_SIMP_TAC1	Simplifies the functions which initialize the matrix G and H and the vector P used for extracting the feedback paths and the forward paths for computing the denominator
NUMERATOR_TAC	This is the main tactics for simplifying the numerator of transfer function of the given graph
DENOMINATOR_TAC	This is the main tactics for simplifying the denominator of transfer function of the given graph

References

[1] S. M. Beillahi and U. Siddique. Formal Analysis of Photonic Signal Processing Systems. `http://hvg.ece.concordia.ca/projects/optics/psp.html`, 2014.

[2] L. N. Binh. *Photonic Signal Processing : Techniques and Applications (Optical Science and Engineering)*. CRC Press, 2010.

[3] H. T. Nagle C. L. Phillips. *Digital Control System Analysis and Design*. Prentice Hall, Inc. USA, 1995.

[4] Signal Flow Graph Similification Program for MATLAB. `http://www.mathworks.com/matlabcentral/fileexchange/22-mason-m`, 2014.

[5] J. Harrison. The HOL Light Theorem Prover. `http://www.cl.cam.ac.uk/~jrh13/hol-light/`.

[6] J. Harrison. Theorem Proving for Verication, Princeton, July 9 2008. `http://www.cl.cam.ac.uk/~jrh13/slides/cav-09jul08/slides.pdf`.

[7] MATLAB. `http://www.mathworks.com/products/matlab/`, 2014.

[8] S.J. Mson. Feedback Theory, Further Properties of Signal Flow Graphs. In *Proc. IRE*, volume 44, pages 920–926, July 1956.

[9] S.J. Mson. Feedback Theory, Some Properties of Signal Flow Graphs. In *Proc. IRE*, volume 41, pages 1144–1156, September 1953.

[10] C. Li P.P. Yupapin, P. Saeung. Characteristics of Complementary Ring-Resonator Add/Drop Filters Modeling by Using Graphical Approach. In *Optics Communications*, volume 272, pages 81–86, 2007.

[11] R. H. Bishop R. C. Drof. *Modern Control Systems*. AEEIZH, USA, 2002.

[12] J.C. Tiernan. An Efficient Search Algorithm to Find Elementary Circuits of a Graph. In *Comm ACM*, volume 13, pages 722–726, 1970.

[13] J. Woodcock, P. G. Larsen, J. Bicarregui, and J. Fitzgerald. Formal Methods: Practice and Experience. *ACM Computing Survey*, 41(4):19:1–19:36, 2009.