Mechanically Proving Determinacy of Hierarchical Block Diagram Translations*

Viorel Preoteasa[1,3(✉)], Iulia Dragomir[2(✉)], and Stavros Tripakis[3,4(✉)]

[1] Space Systems Finland, Espoo, Finland
viorel.preoteasa@gmail.com
[2] Univ. Grenoble Alpes, CNRS, Grenoble INP**,VERIMAG, 38000 Grenoble, France
iulia.dragomir@univ-grenoble-alpes.fr
[3] Aalto University, Espoo, Finland
stavros.tripakis@aalto.fi
[4] Northeastern University, Boston, USA

Abstract. Hierarchical block diagrams (HBDs) are at the heart of embedded system design tools, including Simulink. Numerous translations exist from HBDs into languages with formal semantics, amenable to formal verification. However, none of these translations has been proven correct, to our knowledge.
We present in this paper the first mechanically proven HBD translation algorithm. The algorithm translates HBDs into an algebra of terms with three basic composition operations (serial, parallel, and feedback). In order to capture various translation strategies resulting in different terms achieving different tradeoffs, the algorithm is nondeterministic. Despite this, we prove its *semantic determinacy*: for every input HBD, all possible terms that can be generated by the algorithm are semantically equivalent. We apply this result to show how three Simulink translation strategies introduced previously can be formalized as determinizations of the algorithm, and derive that these strategies yield semantically equivalent results (a question left open in previous work). All results are formalized and proved in the Isabelle theorem-prover and the code is publicly available.

1 Introduction

Dozens of tools, including Simulink [28], the most widespread embedded system design environment, are based on *hierarchical block diagrams* (HBDs). Being a graphical notation (and in the case of Simulink a "closed" one in the sense that the tool is not open-source), such diagrams need to be translated into other formalisms more amenable to formal analysis. Several such translations exist, e.g., see [2,39,29,41,34,12,24,43,44,30] and the related discussion in Section 2. To our knowledge, none of these translations has been formally verified. This paper aims to remedy this fact.

* This work has been partially supported by the Academy of Finland and the U.S. National Science Foundation (awards #1329759 and #1801546). V. Preoteasa – Partially supported by the ECSEL JU MegaM@Rt2 project under grant agreement #737494. I. Dragomir – Partially supported by the the European Union's Horizon 2020 research and innovation programme under grant agreement #730080 (ESROCOS).

** Institute of Engineering Univ. Grenoble Alpes

Our work builds upon the Refinement Calculus of Reactive Systems (RCRS), a publicly available compositional framework for modeling and reasoning about reactive systems [17,18]. RCRS is itself implemented on top of the Isabelle theorem prover [31].

RCRS uses Simulink as one of its front-ends, and includes a tool that translates Simulink diagrams to RCRS theories [16]. This *Translator* implements three translation strategies from HBDs to an algebra of components with three basic composition operators: serial, parallel, and feedback. The several translation strategies are motivated by the fact that each strategy has its own pros and cons. For instance, one strategy may result in shorter and/or easier to understand algebra terms, while another strategy may result in terms that are easier to simplify by manipulating formulas in a theorem prover. But a fundamental question is left open in [16]: are these translation strategies *semantically equivalent*, meaning, do they produce semantically equivalent terms? This is the question we study and answer (positively) in this paper.

The question is non-trivial, as we seek to prove the equivalence of three complex algorithms which manipulate a graphical notation (hierarchical block diagrams) and transform models in this notation into a different textual language, namely, the algebra mentioned above. Terms in this algebra have intricate formal semantics, and formally proving that two given specific terms are equivalent is already a non-trivial exercise. Here, the problem is to prove that a number of translation strategies $T_1, T_2, ..., T_k$ are equivalent, meaning that for *any* given graphical diagram D, the terms resulting from translating D by applying these strategies, $T_1(D), T_2(D), ..., T_k(D)$, are all semantically equivalent.

This equivalence question is important for many reasons. Just like a compiler has many choices when generating code, a HBD translator has many choices when generating algebraic expressions. Just like a correct compiler must guarantee that all possible results are equivalent (independently of optimization or other flags/options), the translator must also guarantee that all possible algebraic expressions are equivalent. Moreover, the algebraic expressions constitute the formal semantics of HBDs, and hence also those of tools like Simulink. Therefore, this determinacy principle is also necessary in order for the formal Simulink semantics to be well-defined.

In order to formulate the equivalence question precisely, we introduce an *abstract* and *nondeterministic* algorithm for translating HBDs into an abstract algebra of components with three composition operations (serial, parallel, feedback) and three constants (split, switch, and sink). By *abstract algorithm* we understand an algorithm that produces terms in this abstract algebra. Concrete versions for this algorithm are obtained when using it for concrete models of the algebra (e.g., *constructive functions*). The algorithm is *nondeterministic* in the sense that it consists of a set of basic operations (transformations) that can be applied in any order. This allows to capture various deterministic translation strategies as determinizations (*refinements* [5]) of the abstract algorithm.

The main contributions of the paper are the following:

1. We formally and mechanically define a translation algorithm for HBDs.
2. We prove that despite its internal nondeterminism, the algorithm achieves deterministic results in the sense that all possible algebra terms that can be generated by the different nondeterministic choices are semantically equivalent.

3. We formalize two translation strategies introduced in [16] as refinements of the abstract algorithm.
4. We formalize also the third strategy (feedbackless) introduced in [16] as an independent algorithm.
5. We mechanically prove the equivalence of these three translation strategies.
6. We make our results publicly available at https://github.com/hbd-translation/TranslateHBD.

To our knowledge, our work constitutes the first and only mechanically proven hierarchical block diagram translator. Moreover, our method is compositional and our abstract algorithm can be instantiated in many different ways, encompassing not just the three translation strategies of [16], but also any other HBD translation strategy that can be devised by combining the basic composition operations defined in the abstract algorithm.

2 Related Work

Model transformation and the verification of its correctness is a long standing line of research, which includes classification of model transformations [3] and the properties they must satisfy with respect to their intent [26], verification techniques [1], frameworks for specifying model transformations (e.g., ATL [19]), and various implementations for specific source and target meta-models. Extensive surveys of the above can be found in [3,11,1].

Several translations from Simulink have been proposed in the literature, including to Hybrid Automata [2], BIP [39], NuSMV [29], Lustre [41], Boogie [34], Timed Interval Calculus [12], Function Blocks [24], I/O Extended Finite Automata [43], Hybrid CSP [44], and SpaceEx [30]. It is unclear to what extent these approaches provide formal guarantees on the determinism of the translation. For example, the order in which blocks in the Simulink diagram are processed might a-priori influence the result. Some works fix this order, e.g., [34] computes the control flow graph and translates the model according to this computed order. In contrast, we prove that the results of our algorithm are equivalent for any order. To the best of our knowledge, the abstract translation proposed hereafter for Simulink is the only one formally defined and mechanically proven correct.

The focus of several works is to validate the preservation of the semantics of the original diagram by the resulting translation (e.g., see [24,35,9,36]). In contrast, our goal is to prove equivalence of all possible translations. Given that Simulink semantics is informal ("what the simulator does"), ultimately the only way to gain confidence that the translation conforms to the original Simulink model is by simulation (e.g., as in [16]).

In general, our approach can be considered as a means in the certification and qualification of compilers by mechanical formal verification. Several works tackle the formal verification of compilers for programming languages: CompCert [25] is a verified compiler for a subset of C with the Coq interactive theorem prover [40], while the verification of a compiler for Lustre with Coq is considered in [4,10]. The aim of these works is to show that the semantics of the original program is preserved during

the different compilation phases until the generated assembly code, while we provide a semantics for HBDs and we prove it correct with respect to the different translation choices.

Further comparison of our approach to additional related works and in particular works on category theory such as [6,13,21,22,23,37,38,42] is included in [32] and is omitted from here due to space limitations. To our knowledge, none of these works has been mechanically formalized nor verified.

3 Preliminaries

For a type or set X, X^* is the type of finite lists with elements from X. We denote the empty list by ϵ, $(x_1, \ldots, x_n)$ denotes the list with elements $x_1, \ldots, x_n$, and for lists x and y, $x \cdot y$ denotes their concatenation. The length of a list x is denoted by $|x|$. The list of common elements of x and y in the order occurring in x is denoted by $x \otimes y$. The list of elements from x that do not occur in y is denoted by $x \ominus y$. We define $x \oplus y = x \cdot (y \ominus x)$, the list of x concatenated with the elements of y not occurring in x. A list x is a *permutation* of a list y, denoted $\mathsf{perm}(x, y)$, if x contains all elements of y (including multiplicities) possibly in a different order. For a list x, $\mathsf{set}(x)$ denotes the set of all elements of x.

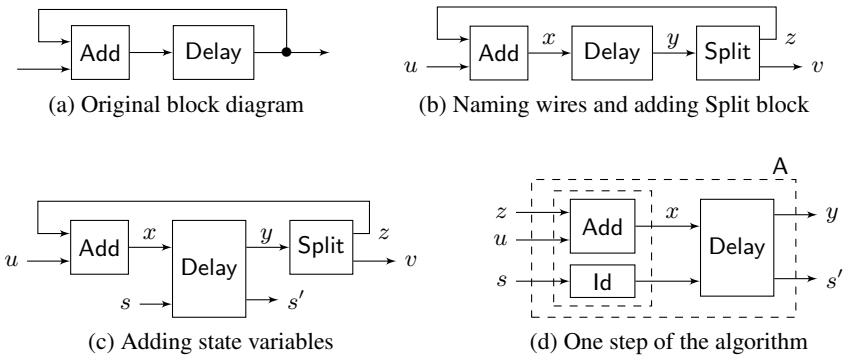
In the sequel we refer to *constructive functions* as used in the *constructive semantics* literature [27,8,20]. Constructive functions enjoy important properties, in particular with respect to feedback composition, and are one of the concrete models for the abstract algebra of HBDs introduced in Section 5. The formal definition of constructive functions is omitted due to lack of space and the reader is referred to [32].

4 Overview of the Translation Algorithm

A *block diagram* N is a network of interconnected blocks. A block may be a basic (*atomic*) block, or a *composite* block that corresponds to a *sub-diagram*. If N contains composite blocks then it is called a *hierarchical block diagram* (HBD); otherwise it is called *flat*. An example of a flat diagram is shown in Fig. 1a. The connections between blocks are called wires, and they have a source block and a target block. For simplicity, we will assume that every wire has a single source and a single target. This can be achieved by adding extra blocks. For instance, the diagram of Fig. 1a can be transformed as in Fig. 1b by adding an explicit block called *Split*.

Let us explain the idea of the translation algorithm. We first explain the idea for flat diagrams, and then we extend it recursively for hierarchical diagrams.

A diagram is represented in the algorithm as a list of elements corresponding to the basic blocks. One element of this list is a triple containing a list of input variables, a list of output variables, and a *function*. The function computes the values of the outputs based on the values of the inputs, and for now it can be thought of as a constructive function. Later this function will be an element of an abstract algebra modeling HBDs. Wires are represented by matching input/output variables from the block representations.



(a) Original block diagram (b) Naming wires and adding Split block

(c) Adding state variables (d) One step of the algorithm

Fig. 1: Running example: diagram for summation.

A block diagram may contain *stateful* blocks such as delays or integrators. We model these blocks using additional state variables (wires). In Fig. 1, the only stateful block is the block Delay. We model this block as an element with two inputs (x, s), two outputs (y, s') and function $(y, s') := (s, x)$ (Fig. 1c). More details about this representation can be found in [16].

In summary, the list representation of the example of Fig. 1 is the following:

$$\big(\mathsf{Add}, \mathsf{Delay}, \mathsf{Split}\big), \text{ where:} \qquad \mathsf{Add} = ((z, u),\ x,\ [z, u \rightsquigarrow z + u]),$$
$$\mathsf{Delay} = ((x, s),\ (y, s'),\ [x, s \rightsquigarrow s, x]), \quad \mathsf{Split} = (y,\ (z, v),\ [y \rightsquigarrow y, y]).$$

The algorithm works by choosing nondeterministically some elements from the list and replacing them with their appropriate composition (serial, parallel, or feedback). The composition must connect all the matching variables. Let us illustrate how the algorithm may proceed on the example of Fig. 1; for the full description of the algorithm see Section 6. Symbols $\circ$, $\|$ and feedback used below denote serial, parallel and feedback compositions, respectively, and they will be formally introduced in Section 5.1.

Suppose the algorithm first chooses to compose Add and Delay. The only matching variable in this case is x, between the output of Add and the first input of Delay. The appropriate composition to use here is serial composition. Because Delay also has s as input, Add and Delay cannot be directly connected in series. This is due to the number of outputs of Add that need to match the number of inputs of Delay. To compute the serial composition, Add must first be composed in parallel with the identity block Id, as shown in Fig. 1d. Doing so, a new element A is created:

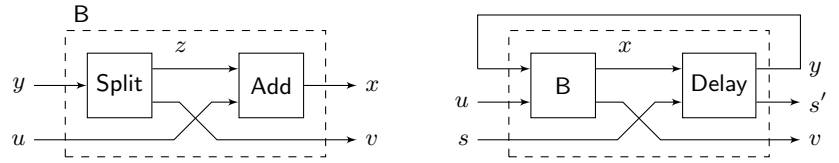
$$\mathsf{A} = ((z, u, s),\ (y, s'),\ \mathsf{Delay} \circ (\mathsf{Add} \parallel \mathsf{Id}))$$

Next, A is composed with Split. In this case we need to connect variable y (using serial composition), as well as z (using feedback composition). The resulting element is A′:

$$\mathsf{A}' = \Big((u, s),\ (v, s'),\ \mathsf{feedback}\big((\mathsf{Split} \parallel \mathsf{Id}) \circ \mathsf{Delay} \circ (\mathsf{Add} \parallel \mathsf{Id})\big)\Big)$$

where we need again to add the Id component for variable s'.

As a different nondeterministic choice, the algorithm may first compose Split and Add into B:



(a) First step: composing Split and Add (b) Second step: composing B and Delay

Fig. 2: A different composition order for the example from Fig. 1.

$$\mathsf{B} = ((y,u),\ (x,v),\ (\mathsf{Add} \parallel \mathsf{Id}) \circ (\mathsf{Id} \parallel [v, u \rightsquigarrow u, v]) \circ (\mathsf{Split} \parallel \mathsf{Id}))$$

In this composition, shown in Fig. 2a, we now need in addition to the Id components, a switch ($[v, u \rightsquigarrow u, v]$) for wires v and u. Next the algorithm composes B and Delay (Fig. 2b):

$$\mathsf{B}' = \Big((u,s),\ (s',v),\ \mathsf{feedback}\big((\mathsf{Delay} \parallel \mathsf{Id}) \circ (\mathsf{Id} \parallel [v, s \rightsquigarrow s, v]) \circ (\mathsf{B} \parallel \mathsf{Id})\big).\Big)$$

As we can see from this example, by considering the blocks in the diagram in different orders, we obtain different expressions. On this example, expression A' is simpler (it has less connectors) than B'. In general, a diagram, being a graph, does not have a predefined canonical order, and we need to show that the result of the algorithm is *the same* regardless of the order in which the blocks are considered.

We make two remarks here. First, the final result of the algorithm is a triple with the same structure as all elements on the original list: (input variables, output variables, function), where the function represents the computation performed by the entire diagram. Therefore, the algorithm can be applied recursively on HBDs. Second, the variables in the representation occur at most twice, once as input, and once as output. The variables occurring only as inputs are the inputs of the resulting final element, and variables occurring only as outputs are the outputs of the resulting final element. This is true in general for all diagrams, due to the representation of splitting of wires. This fact is essential for the correctness of the algorithm as we will see in Section 6.

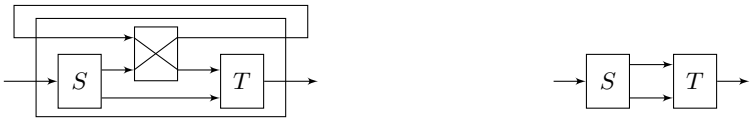
5 An Abstract Algebra for Hierarchical Block Diagrams

We assume that we have a set of Types. We also assume a set of *diagrams* Dgr. Every element $S \in \mathsf{Dgr}$ has input type $t \in \mathsf{Types}^*$ and output type $t' \in \mathsf{Types}^*$. If $t = t_1 \cdots t_n$ and $t' = t'_1 \cdots t'_m$, then S takes as input a tuple of the type $t_1 \times \ldots \times t_n$ and produces as output a tuple of the type $t'_1 \times \ldots \times t'_m$. We denote the fact that S has input type $t \in \mathsf{Types}^*$ and output type $t' \in \mathsf{Types}^*$ by $S : t \xrightarrow{\circ} t'$. The elements of Dgr are abstract.

5.1 Operations of the Algebra of HBDs

Constants. Basic blocks are modeled as constants on Dgr. For types $t, t' \in \mathsf{Types}^*$ we assume the following constants:

$$\mathsf{Id}(t) : t \xrightarrow{\circ} t \quad \mathsf{Split}(t) : t \xrightarrow{\circ} t \cdot t \quad \mathsf{Sink}(t) : t \xrightarrow{\circ} \epsilon \quad \mathsf{Switch}(t,t') : t \cdot t' \xrightarrow{\circ} t' \cdot t$$



$\mathsf{feedback}(\mathsf{Id}(a) \parallel S \;;\; \mathsf{Switch}(a,a) \parallel \mathsf{Id}(t) \;;\; \mathsf{Id}(a) \parallel T) \quad = \quad S \;;\; T$

Fig. 3: Two flat diagrams and their corresponding terms in the abstract algebra.

Id corresponds to the identity block. It copies the input into the output. In the model of constructive functions $\mathsf{Id}(t)$ is the identity function. $\mathsf{Split}(t)$ takes an input x of type t and outputs $x \cdot x$ of type $t \cdot t$. $\mathsf{Sink}(t)$ returns the empty tuple ϵ, for any input x of type t. $\mathsf{Switch}(t, t')$ takes an input $x \cdot x'$ with x of type t and x' of type t' and returns $x' \cdot x$. In the model of constructive functions these diagrams are total functions and they are defined as explained above. In the abstract model, the behaviors of these constants is defined with a set of axioms (see below).

Composition operators. For two diagrams $S : t \xrightarrow{\circ} t'$ and $S' : t' \xrightarrow{\circ} t''$, their *serial composition*, denoted $S \;;\; S' : t \xrightarrow{\circ} t''$ is a diagram that takes inputs of type t and produces outputs of type t''. In the model of constructive functions, the serial composition corresponds to function composition ($S \;;\; S' = S' \circ S$). Please note that in the abstract model we write the serial composition as $S \;;\; S'$, while in the model of constructive functions the first diagram that is applied to the input occurs second in the composition.

The *parallel composition* of two diagrams $S : t \xrightarrow{\circ} t'$ and $S' : r \xrightarrow{\circ} r'$, denoted $S \parallel S' : t \cdot r \xrightarrow{\circ} t' \cdot r'$, is a diagram that takes as input tuples of type $t \cdot r$ and produces as output tuples of type $t' \cdot r'$. This parallel composition corresponds to the parallel composition of constructive functions.

Finally we introduce a *feedback composition*. For $S : a \cdot t \xrightarrow{\circ} a \cdot t'$, where $a \in \mathsf{Types}$ is a single type, the feedback of S, denoted $\mathsf{feedback}(S) : t \xrightarrow{\circ} t'$, is the result of connecting in feedback the first output of S to its first input. Again this feedback operation corresponds to the feedback of constructive functions.

We assume that parallel composition operator binds stronger than serial composition, i.e. $S \parallel T \;;\; R$ is the same as $(S \parallel T) \;;\; R$.

Graphical diagrams can be represented as terms in the abstract algebra, as illustrated in Fig. 3. This figure depicts two diagrams, and their corresponding algebra terms. As it turns out, these two diagrams are equivalent, in the sense that their corresponding algebra terms can be shown to be equal using the axioms presented below.

5.2 Axioms of the Algebra of HBDs

In the abstract algebra, the behavior of the constants and composition operators is defined by a set of axioms, listed below (f^n denotes n applications of function f, so for example $\mathsf{feedback}^2(\cdot) = \mathsf{feedback}(\mathsf{feedback}(\cdot))$):

1. $S : t \xrightarrow{\circ} t' \implies \mathsf{Id}(t) \;;\; S = S \;;\; \mathsf{Id}(t') = S$

2. $S : t_1 \xrightarrow{\circ} t_2 \wedge T : t_2 \xrightarrow{\circ} t_3 \wedge R : t_3 \xrightarrow{\circ} t_4 \Longrightarrow S \,;\, (T \,;\, R) = (S \,;\, T) \,;\, R$
3. $\mathsf{Id}(\epsilon) \parallel S = S \parallel \mathsf{Id}(\epsilon) = S$
4. $S \parallel (T \parallel R) = (S \parallel T) \parallel R$
5. $\mathsf{Id}(t) \parallel \mathsf{Id}(t') = \mathsf{Id}(t \cdot t')$
6. $S : s \xrightarrow{\circ} s' \wedge S' : s' \xrightarrow{\circ} s'' \wedge T : t \xrightarrow{\circ} t' \wedge T' : t' \xrightarrow{\circ} t''$
   $\Longrightarrow\ (S \parallel T) \,;\, (S' \parallel T') = (S \,;\, S') \parallel (T \,;\, T')$
7. $\mathsf{Switch}(t, t' \cdot t'') = \mathsf{Switch}(t, t') \parallel \mathsf{Id}(t'') \,;\, \mathsf{Id}(t') \parallel \mathsf{Switch}(t, t'')$
8. $S : s \xrightarrow{\circ} s' \wedge T : t \xrightarrow{\circ} t' \Longrightarrow \mathsf{Switch}(s, t) \,;\, T \parallel S \,;\, \mathsf{Switch}(t', s') = S \parallel T$
9. $\mathsf{feedback}(\mathsf{Switch}(a, a)) = \mathsf{Id}(a)$
10. $S : a \cdot s \xrightarrow{\circ} a \cdot t \Longrightarrow \mathsf{feedback}(S \parallel T) = \mathsf{feedback}(S) \parallel T$
11. $S : a \cdot s \xrightarrow{\circ} a \cdot t \wedge A : s' \xrightarrow{\circ} s \wedge B : t \xrightarrow{\circ} t'$
    $\Longrightarrow\ \mathsf{feedback}(\mathsf{Id}(a) \parallel A \,;\, S \,;\, \mathsf{Id}(a) \parallel B) = A \,;\, \mathsf{feedback}(S) \,;\, B$
12. $S : a \cdot b \cdot s \xrightarrow{\circ} a \cdot b \cdot t$
    $\Longrightarrow\ \mathsf{feedback}^2(\mathsf{Switch}(b, a) \parallel \mathsf{Id}(s) \,;\, S \,;\, \mathsf{Switch}(a, b) \parallel \mathsf{Id}(t)) = \mathsf{feedback}^2(S)$
13. $\mathsf{Split}(t) \,;\, \mathsf{Sink}(t) \parallel \mathsf{Id}(t) = \mathsf{Id}(t)$
14. $\mathsf{Split}(t) \,;\, \mathsf{Switch}(t, t) = \mathsf{Split}(t)$
15. $\mathsf{Split}(t) \,;\, \mathsf{Id}(t) \parallel \mathsf{Split}(t) = \mathsf{Split}(t) \,;\, \mathsf{Split}(t) \parallel \mathsf{Id}(t)$
16. $\mathsf{Sink}(t \cdot t') = \mathsf{Sink}(t) \parallel \mathsf{Sink}(t')$
17. $\mathsf{Split}(t \cdot t') = \mathsf{Split}(t) \parallel \mathsf{Split}(t') \,;\, \mathsf{Id}(t) \parallel \mathsf{Switch}(t, t') \parallel \mathsf{Id}(t')$

Due to space limitations, the intuition behind these axioms is explained and illustrated with figures in [32].

6 The Abstract Translation Algorithm and its Determinacy

6.1 Diagrams with Named Inputs and Outputs

The algorithm works by first transforming the graph of a HBD into a list of basic components with named inputs and outputs as explained in Section 4. For this purpose we assume a set of names or variables Var and a function $\mathsf{T} : \mathsf{Var} \to \mathsf{Types}$. For $v \in \mathsf{Var}$, $\mathsf{T}(v)$ is the type of variable v. We extend T to lists of variables by $\mathsf{T}(v_1, \ldots, v_n) = (T(v_1), \ldots, T(v_n))$.

Definition 1. *A* diagram with named inputs and outputs *or* io-diagram *for short is a tuple* (in, out, S) *such that* $in, out \in \mathsf{Var}^*$ *are lists of distinct variables, and* $S : \mathsf{T}(in) \xrightarrow{\circ} \mathsf{T}(out)$.

In what follows we use the symbols $A, A', B, \ldots$ to denote io-diagrams, and $\mathsf{I}(A)$, $\mathsf{O}(A)$, and $\mathsf{D}(A)$ to denote the input variables, the output variables, and the diagram of A, respectively.

Definition 2. *For io-diagrams A and B, we define* $\mathsf{V}(A,B) = \mathsf{O}(A) \otimes \mathsf{I}(B) \in \mathsf{Var}^*$.

$\mathsf{V}(A,B)$ is the list of common variables that are output of A and input of B, in the order occurring in $\mathsf{O}(A)$. We use $\mathsf{V}(A,B)$ later to connect for example in series A and B on these common variables, as we did for constructing A from Add and Delay in Section 4.

6.2 General Switch Diagrams

We compose diagrams when their types are matching, and we compose io-diagrams based on matching names of input and output variables. For example if we have two io-diagrams A and B with $\mathsf{O}(A) = u \cdot v$ and $\mathsf{I}(B) = v \cdot u$, then we can compose in series A and B by switching the output of A and feeding it into B, i.e., $(A \mathbin{;} \mathsf{Switch}(\mathsf{T}(u), \mathsf{T}(v)) \mathbin{;} B)$.

In general, for two lists of variables $x = (x_1 \cdots x_n)$ and $y = (y_1 \cdots y_k)$ we define a *general switch diagram* $[x_1 \cdots x_n \rightsquigarrow y_1 \cdots y_k] : \mathsf{T}(x_1 \cdots x_n) \xrightarrow{\circ} \mathsf{T}(y_1 \cdots y_k)$. Intuitively this diagram takes as input a list of values of type $\mathsf{T}(x_1 \cdots x_n)$ and outputs a list of values of type $\mathsf{T}(y_1 \cdots y_k)$, where the output value corresponding to variable y_j is equal to the value corresponding to the first x_i with $x_i = y_j$ and it is arbitrary (unknown) if there is no such x_i. For example in the constructive functions model $[u, v \rightsquigarrow v, u, w, u]$ for input (a, b) outputs $(b, a, \bot, a)$.

To define $[\_ \rightsquigarrow \_]$ we use Split, Sink, and Switch, but we need also an additional diagram that outputs an arbitrary (or unknown) value for an empty input. For $a \in \mathsf{Types}$, we define $\mathsf{Arb}(a) : \epsilon \xrightarrow{\circ} a$ by $\mathsf{Arb}(a) = \mathsf{feedback}(\mathsf{Split}(a))$. The diagram Arb is represented in Fig. 4.

We define now $[x \rightsquigarrow y] : \mathsf{T}(x) \xrightarrow{\circ} \mathsf{T}(y)$ in two steps. First for $x \in \mathsf{Var}^*$ and $u \in \mathsf{Var}$, the diagram $[x \rightsquigarrow u] : \mathsf{T}(x) \xrightarrow{\circ} \mathsf{T}(u)$, for input $a_1, \ldots, a_n$ outputs the value a_i where i is the first index such that $x_i = u$. Otherwise it outputs an arbitrary (unknown) value.

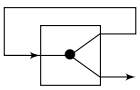


Fig. 4: The diagram Arb.

$$
\begin{aligned}
[\epsilon \rightsquigarrow u] &= \mathsf{Arb}(\mathsf{T}(u)) \\
[u \cdot x \rightsquigarrow u] &= \mathsf{Id}(\mathsf{T}(u)) \parallel \mathsf{Sink}(\mathsf{T}(x)) \\
[v \cdot x \rightsquigarrow u] &= \mathsf{Sink}(\mathsf{T}(v)) \parallel [x \rightsquigarrow u] \qquad (\text{if } u \neq v) \\
[x \rightsquigarrow \epsilon] &= \mathsf{Sink}(\mathsf{T}(x)) \\
[x \rightsquigarrow u \cdot y] &= \mathsf{Split}(\mathsf{T}(x)) \mathbin{;} ([x \rightsquigarrow u] \parallel [x \rightsquigarrow y])
\end{aligned}
$$

6.3 Basic Operations of the Abstract Translation Algorithm

The algorithm starts with a list of io-diagrams and repeatedly applies operations until it reduces the list to only one io-diagram. These operations are the extensions of serial, parallel and feedback from diagrams to io-diagrams.

Definition 3. *The named serial composition of two io-diagrams A and B, denoted $A \;;;\; B$ is defined by $A \;;;\; B = (in, out, S)$, where $x = \mathsf{I}(B) \ominus \mathsf{V}(A,B)$, $y = \mathsf{O}(A) \ominus \mathsf{V}(A,B)$, $in = \mathsf{I}(A) \oplus x$, $out = y \cdot \mathsf{O}(B)$ and*

$$S = [in \rightsquigarrow \mathsf{I}(A) \cdot x] \;;\; \mathsf{D}(A) \parallel [x \rightsquigarrow x] \;;\; [\mathsf{O}(A) \cdot x \rightsquigarrow y \cdot \mathsf{I}(B)] \;;\; [y \rightsquigarrow y] \parallel \mathsf{D}(B).$$

The construction of A from Section 4 can be obtained by applying the named serial composition to Add and Delay.

Fig. 5a illustrates an example of the named serial composition. In this case we have $\mathsf{V}(A,B) = u$, $x = (a,b)$, $y = (v,w)$, $in = (a,c,b)$, and $out = (v,w,d,e)$. The component A has outputs u, v, w, and u is also input of B. Variable u is the only variable that is output of A and input of B. Because the outputs v, w of A are not inputs of B they become outputs of $A \;;;\; B$. Variable a is input for both A and B, so in $A \;;;\; B$ the value of a is split and fed into both A and B. The diagram for this example is:

$$[a,c,b \rightsquigarrow a,c,a,b] \;;\; A \parallel \mathsf{Id}(\mathsf{T}(a,b)) \;;\; [u,v,w,a,b \rightsquigarrow v,w,a,u,b] \;;\; \mathsf{Id}(\mathsf{T}(v,w)) \parallel B$$

The result of the named serial composition of two io-diagrams is not always an io-diagram. The problem is that the outputs of $A \;;;\; B$ are not distinct in general. The next lemma gives sufficient conditions for $A \;;;\; B$ to be an io-diagram.

Lemma 1. *If A, B are io-diagrams and $(\mathsf{O}(A) \ominus \mathsf{I}(B)) \otimes \mathsf{O}(B) = \epsilon$ then $A \;;;\; B$ is an io-diagram. In particular if $\mathsf{O}(A) \otimes \mathsf{O}(B) = \epsilon$ then $A \;;;\; B$ is an io-diagram.*

The named serial composition is associative, expressed by the next lemma.

Lemma 2. *If A, B, C are io-diagrams such that $(\mathsf{O}(A) \ominus \mathsf{I}(B)) \otimes \mathsf{O}(B) = \epsilon$ and $(\mathsf{O}(A) \otimes \mathsf{I}(B)) \otimes \mathsf{I}(C) = \epsilon$, then $(A \;;;\; B) \;;;\; C = A \;;;\; (B \;;;\; C)$.*

Next we introduce the corresponding operation on io-diagrams for the parallel composition.

Definition 4. *If A, B are io-diagrams, then the named parallel composition of A and B, denoted $A \,|||\, B$ is defined by*

$$A \,|||\, B = (\mathsf{I}(A) \oplus \mathsf{I}(B),\ \mathsf{O}(A) \cdot \mathsf{O}(B),\ [\mathsf{I}(A) \oplus \mathsf{I}(B) \rightsquigarrow \mathsf{I}(A) \cdot \mathsf{I}(B)] \;;\; (A \parallel B)).$$

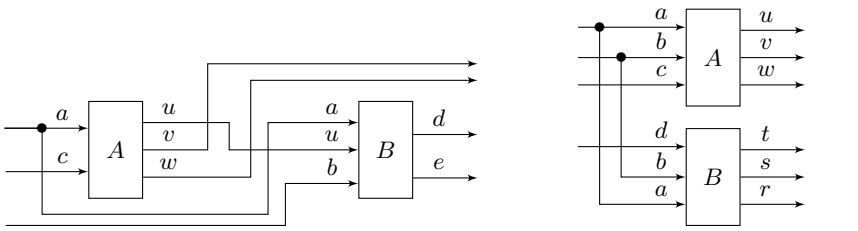
Fig. 5b presents an example of a named parallel composition. The named parallel composition is meaningful only if the outputs of the two diagrams have different names. However, the inputs may not necessarily be distinct as shown in Fig. 5b.

As in the case of named serial composition, the parallel composition of two io-diagrams is not always an io-diagram. Next lemma gives conditions for the parallel composition to be io-diagram and also states that the named parallel composition is associative.

Lemma 3. *Let A, B, and C be io-diagrams, then*

1. $\mathsf{O}(A) \otimes \mathsf{O}(B) = \epsilon \;\Rightarrow\; A \,|||\, B$ *is an io-diagram.*
2. $(A \,|||\, B) \,|||\, C = A \,|||\, (B \,|||\, C)$

Next definition introduces the feedback operator for io-diagrams.



(a) A named serial composition. (b) A named parallel composition.

Fig. 5: Examples of named compositions.

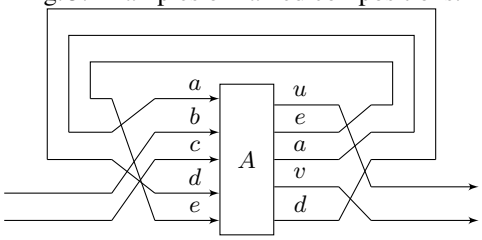


Fig. 6: Example of named feedback composition.

Definition 5. *If A is an io-diagram, then the named feedback of A, denoted $\mathsf{FB}(A)$ is defined by (in, out, S), where $in = \mathsf{I}(A) \ominus \mathsf{V}(A, A)$, $out = \mathsf{O}(A) \ominus \mathsf{V}(A, A)$ and*

$$S = \mathsf{feedback}^{|\mathsf{V}(A,A)|}([\mathsf{V}(A, A) \cdot in \rightsquigarrow \mathsf{I}(A)] \ ; \ S \ ; \ [\mathsf{O}(A) \rightsquigarrow \mathsf{V}(A, A) \cdot out]).$$

The named feedback operation of A connects all inputs and outputs of A with the same name in feedback. Fig. 6 illustrates an example of named feedback composition. The named feedback applied to an io-diagram is always an io-diagram.

Lemma 4. *If A is an io-diagram then $\mathsf{FB}(A)$ is an io-diagram.*

6.4 The Abstract Translation Algorithm

We have now all elements for introducing the abstract translation algorithm. The algorithm starts with a list $\mathcal{A} = (A_1, A_2, \ldots, A_n)$ of io-diagrams, such that for all $i \neq j$, the inputs and outputs of A_i and A_j are disjoint respectively ($\mathsf{I}(A_i) \otimes \mathsf{I}(A_j) = \epsilon$ and $\mathsf{O}(A_i) \otimes \mathsf{O}(A_j) = \epsilon$). We denote this property by $\mathsf{io{-}distinct}(\mathcal{A})$. The algorithm is given in Alg. 1. Formally the algorithm is represented as a monotonic predicate transformer [15], within the framework of refinement calculus [5].

Computing $\mathsf{FB}(A)$ in the last step of the algorithm is necessary only if $\mathcal{A}$ contains initially only one element. However, computing $\mathsf{FB}(A)$ always at the end does not change the result since, as we will see later in Theorem 1, the FB operation is idempotent, i.e. $\mathsf{FB}(\mathsf{FB}(A)) = \mathsf{FB}(A)$. In the presentation of the algorithm, we have used the keyword choose for the nondeterministic choice $\sqcap$, to emphasize the two alternatives.

Note that, semantically, choice (b) of the algorithm is a special case of choice (a), as shown later in Theorem 1. But syntactically, choices (a) and (b) result in different

input: $\mathcal{A} = (A_1, A_2, \ldots, A_n)$ (list of io-diagrams)
while $|\mathcal{A}| > 1$:
choose between options (a) and (b) :
(a) $[\, \mathcal{A} := \mathcal{A}' \mid \exists\, k, B_1, \ldots, B_k, \mathcal{C} : k > 1 \ \wedge$
$\mathsf{perm}(\mathcal{A},\ (B_1, \ldots, B_k) \cdot \mathcal{C}) \wedge \mathcal{A}' = \mathsf{FB}(B_1 \,|||\, \ldots \,|||\, B_k) \cdot \mathcal{C}\,]$
(b) $[\, \mathcal{A} := \mathcal{A}' \mid \exists\, A, B, \mathcal{C} : \mathsf{perm}(\mathcal{A},\ (A, B) \cdot \mathcal{C}) \wedge$
$\mathcal{A}' = \mathsf{FB}(\mathsf{FB}(A)\ ;;\ \mathsf{FB}(B)) \cdot \mathcal{C}\,]$
$A := \mathsf{FB}(A')$ (where A' is the only remaining element of $\mathcal{A}$)

Alg. 1: Nondeterministic algorithm for translating HBDs.

expressions that achieve different performance tradeoffs as observed in Section 4 and as further discussed in [16]. The point of the Translator is to be indeed able to generate semantically equivalent but syntactically different expressions, which achieve different performance tradeoffs [16].

The result for the running example from Section 4 can be obtained by applying the second choice of the algorithm twice for the initial list of io-diagrams ([Add, Delay, Split]), first to Add and Delay to obtain A, and next to A and Split to obtain

$$\Big((u, s),\ (v, s'),\ \mathsf{feedback}\big((\mathsf{D}(\mathsf{Add}) \parallel \mathsf{Id})\ ;\ \mathsf{D}(\mathsf{Delay})\ ;\ ((\mathsf{Split}) \parallel \mathsf{Id})\big)\Big)$$

As opposed to the example from Section 4, the elements are composed serially in the order occurring in the diagram.

6.5 Determinacy of the Abstract Translation Algorithm

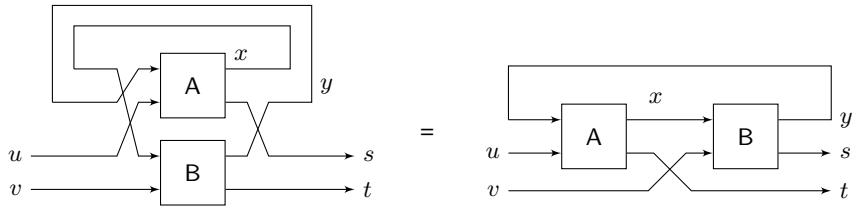
The result of the algorithm depends on how the nondeterministic choices are resolved. However, in all cases the final io-diagrams are equivalent modulo a permutation of the inputs and outputs. To prove this, we introduce the concept *io-equivalence* for two io-diagrams.

Definition 6. *Two io-diagrams A, B are* io-equivalent*, denoted $A \sim B$ if they are equal modulo a permutation of the inputs and outputs, i.e., $\mathsf{I}(B)$ is a permutation of $\mathsf{I}(A)$, $\mathsf{O}(B)$ is a permutation of $\mathsf{O}(A)$ and*

$$\mathsf{D}(A) = [\mathsf{I}(A) \rightsquigarrow \mathsf{I}(B)]\ ;\ \mathsf{D}(B)\ ;\ [\mathsf{O}(B) \rightsquigarrow \mathsf{O}(A)]$$

Lemma 5. *The relation io-equivalent is a congruence relation, i.e, for all io-diagrams A, B, C:*

1. $A \sim A$
2. $A \sim B \Rightarrow B \sim A$
3. $A \sim B \wedge B \sim C \Rightarrow A \sim C$.
4. $A \sim B \Rightarrow \mathsf{FB}(A) \sim \mathsf{FB}(B)$.
5. $\mathsf{O}(A) \otimes \mathsf{O}(B) = \epsilon \Rightarrow A \,|||\, B \sim B \,|||\, A$.
6. *If* $\mathsf{io{-}distinct}(A_1, \ldots, A_n)$ *and* $\mathsf{perm}((A_1, \ldots, A_n),\ (B_1, \ldots, B_n))$ *then*
$$A_1 \,|||\, \ldots A_n \sim B_1 \,|||\, \ldots B_n.$$



(a) Composing A and B in parallel and then in feedback

(b) Composing A and B in series and then in feedback

Fig. 7: Named feedback of parallel composition is equivalent to named feedback of serial composition.

To prove correctness of the algorithm we also need the following results:

Theorem 1. *If A, B are io-diagrams such that $\mathsf{I}(A) \otimes \mathsf{I}(B) = \epsilon$ and $\mathsf{O}(A) \otimes \mathsf{O}(B) = \epsilon$ then*

$$(1) \quad \mathsf{FB}(A \,|||\, B) = \mathsf{FB}(\mathsf{FB}(A) \;;; \mathsf{FB}(B)) \qquad \textit{and} \qquad (2) \quad \mathsf{FB}(\mathsf{FB}(A)) = \mathsf{FB}(A).$$

The proof of Theorem 1 is quite involved and requires several properties of diagrams (see the RCRS formalization [18] for details). Fig. 7 illustrates a simplified application of Theorem 1 (1). In the general case of this theorem there are possibly multiple wires between A and B. There may also be wires between the outputs and inputs of A, and B, and these wires may also be inter-mixed.

We can now state and prove one of the main results of this paper, namely, determinacy of Alg. 1.

Theorem 2. *If $\mathcal{A} = (A_1, A_2, \ldots, A_n)$ is the initial list of io-diagrams satisfying $\mathsf{io{-}distinct}(\mathcal{A})$, then Alg. 1 terminates, and if A is the io-diagram computed by the algorithm, then*

$$A \sim \mathsf{FB}(A_1 \,|||\, \ldots \,|||\, A_n)$$

7 Proving Equivalence of Two Translation Strategies

To demonstrate the usefulness of our framework, we return to our original motivation, namely, the open problem of how to prove equivalence of the translation strategies introduced in [16]. Two of the translation strategies of [16], called *feedback-parallel* and *incremental* translation, can be seen as a determinizations (or refinements) of the abstract algorithm of Section 6, and therefore can be shown to be equivalent and correct with respect to the abstract semantics. (The third strategy proposed in [16], called *feedbackless*, is significantly different and is presented in the next section.)

The feedback-parallel strategy is the implementation of the abstract algorithm where we choose $k = |\mathcal{A}|$. Intuitively, all diagram components are put in parallel and the common inputs and outputs are connected via feedback operators. On the running example from Fig. 1c, this strategy will generate the following component:

$$\begin{aligned}&((u,s),\ (v,s'),\ \mathsf{feedback}^3([z,x,y,u,s \rightsquigarrow z,u,x,s,y]\\ &\quad ;\ \mathsf{D(Add)} \parallel \mathsf{D(Delay)} \parallel \mathsf{D(Split)}\ ;\ [x,y,s',z,v \rightsquigarrow z,x,y,v,s']))\end{aligned}$$

The switches are ordering the variables such that the feedback variables are first and in the same order in both input and output lists.

The incremental strategy is the implementation of the abstract algorithm where we use only the second choice of the algorithm and the first two components of the list $\mathcal{A}$. This strategy is dependent on the initial order of $\mathcal{A}$, and we order $\mathcal{A}$ topologically (based on the input - output connections) at the beginning, in order to reduce the number of switches needed.

Again on the running example, assume that this strategy composes first Add with Delay, and the result is composed with Split. The following component is then obtained:

$$((u, s),\ (v, s'),\ \mathsf{feedback}(\mathsf{D}(\mathsf{Add}) \parallel \mathsf{Id}\ ;\ \mathsf{D}(\mathsf{Delay})\ ;\ \mathsf{D}(\mathsf{Split}) \parallel \mathsf{Id})$$

The Add and Split components are put in parallel with Id for the unconnected input and output state respectively. Next all components are connected in series with one feedback operator for the variable z.

The next theorem shows that the two strategies are equivalent, and that they are independent of the initial order of $\mathcal{A}$.

Theorem 3. *If A and B are the result of the feedback-parallel and incremental strategies on $\mathcal{A}$, respectively, then A and B are input - output equivalent ($A \sim B$). Moreover both strategies are independent of the initial order of $\mathcal{A}$.*

Since both strategies are refinements of the nondeterministic algorithm, they both satisfy the same correctness conditions of Theorem 2.

8 Proving Equivalence of A Third Translation Strategy

The abstract algorithm for translating HBDs, as well as the two translation strategies presented in Section 7, use the feedback operator when translating diagrams. As discussed in [16], expressions that contain the feedback operator are more complex to process and simplify. For this reason, we wish to avoid using the feedback operator as much as possible. Fortunately, in practice, diagrams such as those obtained from Simulink are *deterministic* and *algebraic loop free*. As it turns out, such diagrams can be translated into algebraic expressions that do not use the feedback operator at all [16]. This can be done using the third translation strategy proposed in [16], called *feedbackless*.

While the two translation strategies presented in Section 7 can be modeled as refinements of the abstract algorithm, the feedbackless strategy is significantly more complex, and cannot be captured as such a refinement. We therefore treat it separately in this section. In particular, we formalize the feedbackless strategy and we show that it is equivalent to the abstract algorithm, namely, that for the same input, the results of the two algorithms are io-equivalent.

8.1 Deterministic and Algebraic-Loop-Free Diagrams

Before we introduce the feedbackless strategy, we need some additional definitions.

Definition 7. *A diagram S is* deterministic *if* $[x \rightsquigarrow x, x]\ ;\ (S \parallel S) = S\ ;\ [y \rightsquigarrow y, y]$. *An io-diagram A is* deterministic *if* $\mathsf{D}(A)$ *is deterministic.*

The definition of deterministic diagram corresponds to the following intuition. If we execute two copies of S in parallel using the same input value x, we should obtain the same result as executing one S for the same input value x.

The deterministic property is closed under the serial, parallel, and switch operations of the HBD algebra.

Lemma 6. *If $S, T \in \mathsf{Dgr}$ are deterministic and x, y are lists of variables such that x is distinct and $\mathsf{set}(y) \subseteq \mathsf{set}(x)$, then $[x \rightsquigarrow y]$, and $S\ ;\ T$, and $S \parallel T$ are also deterministic.*

It is not obvious whether we can deduce from the axioms that the deterministic property is closed under the feedback operation. However, since we do not use the feedback operation in this algorithm, we do not need this property.

Definition 8. *The* output input dependency relation *of an io-diagram A is defined by*

$$\mathsf{oi\_rel}(A) = \mathsf{set}(\mathsf{O}(A)) \times \mathsf{set}(\mathsf{I}(A))$$

and the output input dependency relation *of a list $\mathcal{A} = [A_1, \ldots, A_n]$ of io-diagrams is defined by*

$$\mathsf{oi\_rel}(\mathcal{A}) = \mathsf{oi\_rel}(A_1) \cup \ldots \cup \mathsf{oi\_rel}(A_n)$$

A list $\mathcal{A}$ of io-diagrams is algebraic loop free*, denoted* $\mathsf{loop\_free}(\mathcal{A})$*, if*

$$(\forall x : (x, x) \notin (\mathsf{oi\_rel}(\mathcal{A}))^+)$$

where $(\mathsf{oi\_rel}(\mathcal{A}))^+$ *is the reflexive and transitive closure of relation* $(\mathsf{oi\_rel}(\mathcal{A}))$.

If we apply this directly to the list of io-diagrams from our example $\mathcal{A} = [\mathsf{Add}, \mathsf{Delay}, \mathsf{Split}]$ we obtain

$$\mathsf{oi\_rel}(\mathcal{A}) = \{(x, u), (x, z), (y, x), (y, s), (s', x), (s', s), (z, y), (v, y)\}$$

and we have that $(z, z) \in (\mathsf{oi\_rel}(\mathcal{A}))^+$ because $(z, y), (y, x), (x, z) \in \mathsf{oi\_rel}(\mathcal{A})$, therefore $\mathcal{A}$ is not algebraic loop free. However, the diagram from the example is accepted by Simulink, and it is considered algebraic loop free. In our treatment $\mathsf{oi\_rel}(\mathcal{A})$ contains pairs that do not represent genuine output input dependencies. For example output y of Delay depends only on the input s, and it does not depend on x. Similarly, output s' of Delay depends only on x.

Before applying the feedbackless algorithm, we change the initial list of blocks into a new list such that the output input dependencies are recorded more accurately, and all elements in the new list have one single output. We split a basic block A into a list of blocks $A_1, \ldots, A_n$ with single outputs such that $A \sim A_1 \,|||\, \ldots \,|||\, A_n$. Basically every block with n outputs is split into n single output blocks.

We can do the splitting systematically by composing a block A with all projections of the output. For example if $A = (x, (u_1, \ldots, u_n), S)$, then we can split A into $A_i = (x, u_i, S\ ;\ [u_1, \ldots, u_n \rightsquigarrow u_i])$. Such splitting is always possible:

Lemma 7. *If A is deterministic, then $A_1, \ldots, A_n$ is a splitting of A, i.e.*

$$A \sim A_1 \,|||\, \ldots \,|||\, A_n.$$

However, this will still introduce unwanted output input dependencies. We solve this problem by defining the splitting for every basic block, such that it accurately records the output input dependency. For example, we split the delay block into Delay_1 and Delay_2:

$$\mathsf{Delay}_1 = (s, y, [s \rightsquigarrow s]) = (s, y, \mathsf{Id}) \quad \text{and} \quad \mathsf{Delay}_2 = (x, s', [x \rightsquigarrow x]) = (x, s', \mathsf{Id})$$

The Split block is split into Split_1 and Split_2:

$$\mathsf{Split}_1 = (y, z, [y \rightsquigarrow y]) = (y, z, \mathsf{Id}) \quad \text{and} \quad \mathsf{Split}_2 = (y, v, [y \rightsquigarrow y]) = (y, v, \mathsf{Id})$$

The blocks Delay_1, Delay_2, Split_1, and Split_2 are all the same, except the naming of the inputs and outputs. The Add block has one single output that depends on both inputs, so it remains unchanged.

After splitting, the list of single output blocks for our example becomes

$$\mathcal{B} = \big(\mathsf{Add}, \mathsf{Delay}_1, \mathsf{Delay}_2, \mathsf{Split}_1, \mathsf{Split}_2\big)$$

and we have

$$\mathsf{oi\_rel}(\mathcal{B}) = \{(x, u), (x, z), (y, s), (s', x), (z, y), (v, y)\}.$$

Now $\mathcal{B}$ is algebraic loop free.

Definition 9. *A block diagram is* algebraic loop free *if, after splitting, the list of blocks is algebraic loop free.*

We assume that every splitting of a block A into $B_1, \ldots, B_k$ is done such that $A \sim B_1 \,|||\, \ldots \,|||\, B_k$.

Lemma 8. *If a list of blocks $\mathcal{A} = (A_1, \ldots, A_n)$ is split into $\mathcal{B} = (B_1, \ldots, B_m)$, then we have*

$$A_1 \,|||\, \ldots \,|||\, A_n \sim B_1 \,|||\, \ldots \,|||\, B_m.$$

For the feedbackless algorithm, we assume that $\mathcal{A}$ is algebraic loop free, all io-diagrams in $\mathcal{A}$ are single output and deterministic, and all outputs are distinct. We denote this by $\mathsf{ok\_fbless}(\mathcal{A})$.

Definition 10. *For $\mathcal{A}$, such that* $\mathsf{ok\_fbless}(\mathcal{A})$*, a variable u is* internal *in $\mathcal{A}$ if there exist A and B in $\mathcal{A}$ such that $\mathsf{O}(A) = u$ and $u \in \mathsf{set}(\mathsf{I}(B))$. We denote the set of internal variables of $\mathcal{A}$ by* $\mathsf{internal}(\mathcal{A})$.

Definition 11. *If A and B are single output io-diagrams, then their* internal serial composition *is defined by*

$$A \rhd B = \mathsf{if}\ \mathsf{set}(\mathsf{O}(A)) \subseteq \mathsf{set}(\mathsf{I}(B))\ \mathsf{then}\ A\ ;;\ B\ \mathsf{else}\ B$$

and

$$A \rhd (B_1, \ldots, B_n) = (A \rhd B_1, \ldots, A \rhd B_n)$$

We use this composition when all io-diagrams have a single output, and for an io-diagram A, we connect A in series with all io-diagrams from $B_1, \ldots, B_n$ that have $\mathsf{O}(A)$ as an input.

The internal serial composition satisfies some properties that are used in proving the correctness of the algorithm.

Lemma 9. *If* $\mathsf{ok\_fbless}(A, B, C)$ *then* $((A \rhd B) \rhd (A \rhd C)) \sim ((B \rhd A) \rhd (B \rhd C))$

Lemma 10. *If* $\mathsf{ok\_fbless}(\mathcal{A})$ *and* $A \in \mathsf{set}(\mathcal{A})$ *such that* $\mathsf{O}(A) \in \mathsf{internal}(\mathcal{A})$ *then*

$$\mathsf{ok\_fbless}(A \rhd (\mathcal{A} \ominus A)) \quad \textit{and} \quad \mathsf{internal}(A \rhd (\mathcal{A} \ominus A)) = \mathsf{internal}(\mathcal{A}) - \{\mathsf{O}(A)\}.$$

8.2 Functional Definition of the Feedbackless Strategy

Definition 12. *For a list x of distinct internal variables of $\mathcal{A}$, we define by induction on x the function* $\mathsf{fbless}(x, \mathcal{A})$ *by*

$$\mathsf{fbless}(\epsilon, \mathcal{A}) = \mathcal{A} \quad \textit{and} \quad \mathsf{fbless}(u \cdot x, \mathcal{A}) = \mathsf{fbless}(x, A \rhd (\mathcal{A} \ominus A))$$

where A is the unique io-diagram from $\mathcal{A}$ with $\mathsf{O}(A) = u$.

Lemma 10 shows that the function fbless is well defined.

The function fbless is the functional equivalent of the feedbackless iterative algorithm that we introduce in Section 8.3.

Theorem 4. *If $\mathcal{A} = (A_1, \ldots, A_n)$ is a list of io-diagrams satisfying* $\mathsf{ok\_fbless}(\mathcal{A})$*, x is a distinct list of all internal variables of $\mathcal{A}$ (*$\mathsf{set}(x) = \mathsf{internal}(\mathcal{A})$*), and $(B_1, \ldots, B_k) =$* $\mathsf{fbless}(x, \mathcal{A})$ *then*

$$\mathsf{FB}(A_1 \,|||\, \ldots \,|||\, A_n) \sim (B_1 \,|||\, \ldots \,|||\, B_n).$$

This theorem together with Lemma 8 show that the result of the fbless function is io-equivalent to the results of the nondeterministic algorithm. This theorem also shows that the result of fbless is independent of the choice of the order of the internal variables in x.

The proof of Theorem 4 is based on Lemmas 9 and 10, and is available in the RCRS formalization – `https://github.com/hbd-translation/TranslateHBD`.

8.3 The Feedbackless Translation Algorithm

The recursive function fbless calculates the feedbackless translation, but it assumes that the set of internal variables is given at the beginning in a specific order. We want an equivalent iterative version of this function, which at every step picks an arbitrary io-diagram A with internal output, and performs one step:

$$\mathcal{A} := A \rhd (\mathcal{A} \ominus A)$$

The feedbackless algorithm is given in Alg. 2.

input: $\mathcal{A} = (A_1 \ldots, A_n)$ (list of io-diagrams satisfying $\mathsf{ok\_fbless}(\mathcal{A})$)
while $\mathsf{internal}(\mathcal{A}) \neq \emptyset$:
$[\, \mathcal{A} := \mathcal{A}' \mid \exists\, A \in \mathsf{set}(\mathcal{A}) : \mathsf{O}(A) \in \mathsf{internal}(\mathcal{A}) \wedge \mathcal{A}' = A \rhd (\mathcal{A} \ominus A) \,]$
$A := B_1 \,|||\, \ldots \,|||\, B_k$ (where $\mathcal{A} = (B_1, \ldots, B_k)$)

Alg. 2: Feedbackless algorithm for translating HBDs.

The feedbackless algorithm is also nondeterministic, because it allows choosing at every step one of the available io-diagrams with internal output. As we will see in Section 8.4, this nondeterminism allows for different implementations regarding the complexity of the generated expressions.

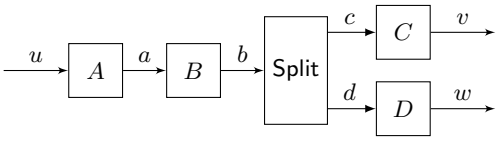


Fig. 8: Example for efficient implementation of feedbackless.

Theorem 5. *If $\mathcal{A} = (A_1 \ldots, A_n)$ is a list of io-diagrams satisfying* $\mathsf{ok\_fbless}(\mathcal{A})$*, then the feedbackless algorithm terminates for input $\mathcal{A}$, and if A is the output of the algorithm on $\mathcal{A}$, then*

$$\mathsf{FB}(A_1 \,|||\, \ldots \,|||\, A_n) \sim A.$$

Theorem 6. *For a deterministic and algebraic loop free block diagram, the feedbackless algorithm and the nondeterministic algorithm are equivalent.*

8.4 On the Nondeterminism of the Feedbackless Translation

We have seen already that different choices in the nondeterministic abstract algorithm result in different algebraic expressions, e.g., with different numbers of composition operators. We show in this section that the same is true for the feedbackless translation algorithm. In particular, consider a framework like the Refinement Calculus of Reactive Systems [16], where the intermediate results of the algorithm are symbolically simplified at every translation step. Different choices of the order of internal variables could result in different complexities of the simplification work. We illustrate this with the example from Fig. 8.

After the splitting phase, the list of blocks for this example is

$$\mathcal{A} = \big((u, a, A),\ (a, b, B),\ (b, c, \mathsf{Id}),\ (b, d, \mathsf{Id}),\ (c, v, C),\ (d, w, D)\big)$$

and the set of internal variables is

$$\mathsf{internal}(\mathcal{A}) = \{a, b, c, d\}.$$

If we choose the order (c, d, b, a), then after first two steps (including intermediate simplifications) we obtain the list:

$$\big((u, a, A),\ (a, b, B),\ (b, v, C),\ (b, w, D)\big)$$

After another step for internal variable b we obtain:

$$\big((u, a, A),\ (a, v, \mathsf{simplify}(B\ ;\ C)),\ (a, w, \mathsf{simplify}(B\ ;\ D))\big)$$

where the function simplify models the symbolic simplification. Finally, after applying the step for the internal variable a we obtain:

$$\big((u, v, \mathsf{simplify}(A\ ;\ \mathsf{simplify}(B\ ;\ C))),\ (u, w, \mathsf{simplify}(A\ ;\ \mathsf{simplify}(B\ ;\ D)))\big) \quad (1)$$

In this order, we end up simplifying A serially composed with B twice. This is especially inefficient if A and B are complex. If we choose the order (c, d, a, b), then in the first three steps we obtain:

$$\big((u, b, \mathsf{simplify}(A\ ;\ B)),\ (b, v, C),\ (b, w, D)\big)$$

At this point the term $A \ ; \ B$ is simplified, and the simplified version is composed with C and D to obtain:

$$\big((u, v, \mathsf{simplify}(\mathsf{simplify}(A \ ; \ B) \ ; \ C)),\ (u, w, \mathsf{simplify}(\mathsf{simplify}(A \ ; \ B) \ ; \ D))\big) \quad (2)$$

If we compare relations (1) and (2) we see the same number of occurrences of simplify, but in relation (2) there are two occurrences of the common subterm $\mathsf{simplify}(A \ ; \ B)$, and this is simplified only once.

As this example shows, different choices of the nondeterministic feedbackless translation strategy result in expressions of different quality, in particular with respect to simplification. It is beyond the scope of this paper to examine efficient deterministic implementations of the feedbackless translation. Our goal here is to prove the correctness of this translation, by proving its equivalence to the abstract algorithm. It follows that every refinement/determinization of the feedbackless strategy will also be equivalent to the abstract algorithm, and therefore a correct implementation of the semantics. Once we know that all possible refinements give equivalent results, we can concentrate in finding the most efficient strategy. In general, we remark that this way of using the mechanisms of nondeterminism and refinement are standard in the area of correct by construction program development, and are often combined to separate the concerns of correctness and efficiency, as is done here.

9 Implementation in Isabelle

Our implementation in Isabelle uses locales [7] for the axioms of the algebra. We use locale interpretations to show that these axioms are consistent. In Isabelle locales are a powerful mechanism for developing consistent abstract theories (based on axioms). To represent the algorithm we use monotonic predicate transformers. To prove correctness of the algorithm we use Hoare total correctness rules.

The formalization contains the locale for the axioms, a theory for constructive functions, and one for proving that such functions are a model for the axioms. An important part of the formalization is the theory introducing the diagrams with named inputs and outputs, and their operations and properties. The formalization also includes a theory for monotonic predicate transformers, refinement calculus, Hoare total correctness rules for programs, and a theory for the nondeterministic algorithm and its correctness.

In total the formalization contains 14797 lines of Isabelle code of which 13587 lines of code for the actual problem, i.e., excluding the code for monotonic predicate transformers, refinement calculus, and Hoare rules. The formalization is available at `https://github.com/hbd-translation/TranslateHBD`.

10 Conclusions and Future Work

We introduced an abstract algebra for hierarchical block diagrams, and an abstract algorithm for translating HBDs to terms of this algebra. We proved that this algorithm is correct in the sense that no matter how its nondeterministic choices are resolved, the results are semantically equivalent. As an application, we closed a question left open

in [16] by proving that the Simulink translation strategies presented there yield equivalent results. Our HBD algebra is reminiscent of the algebra of flownomials [14] but our axiomatization is more general, in the sense that our axioms are weaker. This implies that all models of flownomials are also models of our algebra. Here, we presented constructive functions as one possible model of our algebra.

Our work applies to hierarchical block diagrams in general, and the de facto predominant tool for embedded system design, Simulink. Proving the HBD translator correct is a challenging problem, and as far as we know our work is the only one to have achieved such a result.

We believe that our results are reusable in other contexts as well, in at least two ways. First, every other translation that can be shown to be a refinement/special case of our abstract translation algorithm, is automatically correct. For example, [34,44] impose an order on blocks such that they use mostly serial composition and could be considered an instance of our abstract algorithm. Second, our algorithms translate diagrams into an abstract algebra. By choosing different models of this algebra we obtain translations into these alternative models.

As future work we plan to investigate further HBD translation strategies, in addition to those studied above. Currently the RCRS Translator can only partially handle diagrams with algebraic loops, i.e., with instantaneous circular dependencies. Fully dealing with diagrams with algebraic loops is a non-trivial problem, because of the subtleties of instantaneous feedback for non-deterministic and non-input-receptive systems [33]. For deterministic and input-receptive systems, however, the model of constructive functions should be sufficient. Another future research goal is to unify the proof of the third translation strategy with that of the other two which are currently modeled as refinements of the abstract translation algorithm.

This work covers hierarchical block diagrams in general and Simulink in particular. Any type of diagram can be handled, however, we do assume a *single-rate* (i.e., synchronous) semantics. Handling multi-rate or event-triggered diagrams is left for future work. Handling hierarchical state machine models such as Stateflow is also left for future work.

As mentioned in Section 2, there are many existing translations from Simulink to other formalisms. It is beyond the scope of this paper to define and prove correctness of those translations, but this could be another future work direction. In order to do this, one would first need to formalize those translations. This in turn requires detailed knowledge of the algorithms or even access to their implementation, which is not always available. Our work and source code are publicly available and we hope can serve as a good starting point for others who may wish to provide formal correctness proofs of diagram translations.

Acknowledgments

We would like to thank Gheorghe Ştefănescu for his help with the algebra of flownomials.

References

1. Lukman Ab. Rahim and Jon Whittle. A survey of approaches for verifying model transformations. *Software & Systems Modeling*, 14(2):1003–1028, 2015.
2. Aditya Agrawal, Gyula Simon, and Gabor Karsai. Semantic Translation of Simulink/Stateflow Models to Hybrid Automata Using Graph Transformations. *Electronic Notes in Theoretical Computer Science*, 109:43 – 56, 2004.
3. Moussa Amrani, Benoît Combemale, Levi Lúcio, Gehan M. K. Selim, Jürgen Dingel, Yves Le Traon, Hans Vangheluwe, and James R. Cordy. Formal verification techniques for model transformations: A tridimensional classification. *Journal of Object Technology*, 14(3):1:1–43, August 2015.
4. Cédric Auger. *Compilation certifiée de SCADE/LUSTRE. (Certified compilation of SCADE/LUSTRE)*. PhD thesis, University of Paris-Sud, Orsay, France, 2013. In French.
5. Ralph-Johan Back and Joakim von Wright. *Refinement Calculus. A Systematic Introduction*. Springer, 1998.
6. John C. Baez and Jason Erbele. Categories in control. *CoRR*, abs/1405.6881, 2015.
7. Clemens Ballarin. Locales: A module system for mathematical theories. *Journal of Automated Reasoning*, 52(2):123–153, 2014.
8. G. Berry. The Constructive Semantics of Pure Esterel, 1999.
9. Olivier Bouissou and Alexandre Chapoutot. An Operational Semantics for Simulink's Simulation Engine. *SIGPLAN Not.*, 47(5):129–138, June 2012.
10. Timothy Bourke, Lélio Brun, Pierre-Évariste Dagand, Xavier Leroy, Marc Pouzet, and Lionel Rieg. A formally verified compiler for lustre. *SIGPLAN Not.*, 52(6):586–601, June 2017.
11. Daniel Calegari and Nora Szasz. Verification of model transformations. *Electronic Notes in Theoretical Computer Science*, 292:5 – 25, 2013.
12. Chunqing Chen, Jin Song Dong, and Jun Sun. A formal framework for modeling and validating Simulink diagrams. *Formal Aspects of Computing*, 21(5):451–483, 2009.
13. Bruno Courcelle. *A representation of graphs by algebraic expressions and its use for graph rewriting systems*, pages 112–132. Springer Berlin Heidelberg, Berlin, Heidelberg, 1987.
14. Gheorghe Ştefănescu. *Network Algebra*. Springer, 2000.
15. E.W. Dijkstra. Guarded commands, nondeterminacy and formal derivation of programs. *Commun. ACM*, 18(8):453–457, 1975.
16. Iulia Dragomir, Viorel Preoteasa, and Stavros Tripakis. Compositional semantics and analysis of hierarchical block diagrams. In Dragan Bosnacki and Anton Wijs, editors, *Model Checking Software - 23rd International Symposium, SPIN 2016, Co-located with ETAPS 2016, Eindhoven, The Netherlands, April 7-8, 2016, Proceedings*, volume 9641 of *Lecture Notes in Computer Science*, pages 38–56. Springer, 2016.
17. Iulia Dragomir, Viorel Preoteasa, and Stavros Tripakis. The Refinement Calculus of Reactive Systems Toolset. In Dirk Beyer and Marieke Huisman, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, pages 201–208, Cham, 2018. Springer International Publishing.
18. Iulia Dragomir, Viorel Preoteasa, and Stavros Tripakis. The Refinement Calculus of Reactive Systems Toolset - Feb 2018. figshare. https://doi.org/10.6084/m9.figshare.5900911, February 2018.
19. Eclipse. ATL - a model transformation technology. http://www.eclipse.org/atl/.
20. S. Edwards and E.A. Lee. The semantics and execution of a synchronous block-diagram language. *Sci. Comp. Progr.*, 48:21–42(22), July 2003.
21. Dan R. Ghica and Achim Jung. Categorical semantics of digital circuits. In Ruzica Piskac and Muralidhar Talupur, editors, *2016 Formal Methods in Computer-Aided Design, FMCAD 2016, Mountain View, CA, USA, October 3-6, 2016*, pages 41–48. IEEE, 2016.

22. Dan R. Ghica, Achim Jung, and Aliaume Lopez. Diagrammatic semantics for digital circuits. In Valentin Goranko and Mads Dam, editors, *26th EACSL Annual Conference on Computer Science Logic, CSL 2017, August 20-24, 2017, Stockholm, Sweden*, volume 82 of *LIPIcs*, pages 24:1–24:16. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2017.
23. Dan R. Ghica and Aliaume Lopez. A structural and nominal syntax for diagrams. In Bob Coecke and Aleks Kissinger, editors, *Proceedings 14th International Conference on Quantum Physics and Logic, QPL 2017, Nijmegen, The Netherlands, 3-7 July 2017.*, volume 266 of *EPTCS*, pages 71–83, 2017.
24. Chia han Yang and Valeriy Vyatkin. Transformation of Simulink models to IEC 61499 Function Blocks for verification of distributed control systems. *Control Engineering Practice*, 20(12):1259–1269, 2012.
25. Xavier Leroy, Sandrine Blazy, Daniel Kästner, Bernhard Schommer, Markus Pister, and Christian Ferdinand. CompCert - A Formally Verified Optimizing Compiler. In *ERTS 2016: Embedded Real Time Software and Systems, 8th European Congress*, Toulouse, France, January 2016. SEE.
26. Levi Lúcio, Moussa Amrani, Juergen Dingel, Leen Lambers, Rick Salay, Gehan M. Selim, Eugene Syriani, and Manuel Wimmer. Model transformation intents and their properties. *Softw. Syst. Model.*, 15(3):647–684, July 2016.
27. S. Malik. Analysis of cyclic combinational circuits. *IEEE Trans. Computer-Aided Design*, 13(7):950–956, 1994.
28. MathWorks. Simulink. https://www.mathworks.com/products/simulink.html.
29. B. Meenakshi, Abhishek Bhatnagar, and Sudeepa Roy. Tool for Translating Simulink Models into Input Language of a Model Checker. In *ICFEM*, volume 4260 of *LNCS*, pages 606–620. Springer, 2006.
30. Stefano Minopoli and Goran Frehse. SL2SX Translator: From Simulink to SpaceEx Models. In *Proceedings of the 19th International Conference on Hybrid Systems: Computation and Control*, HSCC '16, pages 93–98, New York, NY, USA, 2016. ACM.
31. Tobias Nipkow, Lawrence C. Paulson, and Markus Wenzel. *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*. LNCS 2283. Springer, 2002.
32. Viorel Preoteasa, Iulia Dragomir, and Stavros Tripakis. Mechanically Proving Determinacy of Hierarchical Block Diagram Translations. *CoRR*, abs/1611.01337, 2018.
33. Viorel Preoteasa and Stavros Tripakis. Towards compositional feedback in non-deterministic and non-input-receptive systems. In *Proceedings of the 31st Annual ACM/IEEE Symposium on Logic in Computer Science*, LICS '16, pages 768–777, New York, NY, USA, 2016. ACM.
34. Robert Reicherdt and Sabine Glesner. Formal Verification of Discrete-Time MATLAB/Simulink Models Using Boogie. In Dimitra Giannakopoulou and Gwen Salaün, editors, *Software Engineering and Formal Methods: 12th International Conference, SEFM 2014, Grenoble, France, September 1-5, 2014. Proceedings*, pages 190–204, Cham, 2014. Springer International Publishing.
35. Michael Ryabtsev and Ofer Strichman. Translation Validation: From Simulink to C. In Ahmed Bouajjani and Oded Maler, editors, *Computer Aided Verification: 21st International Conference, CAV 2009, Grenoble, France, June 26 - July 2, 2009. Proceedings*, pages 696–701, Berlin, Heidelberg, 2009. Springer Berlin Heidelberg.
36. Sebastian Schlesinger, Paula Herber, Thomas Göthel, and Sabine Glesner. Proving transformation correctness of refactorings for discrete and continuous simulink models. In *ICONS 2016, The Eleventh International Conference on Systems, EMBEDDED 2016, International Symposium on Advances in Embedded Systems and Applications*, pages 45–50. IARIA XPS Press, 2016.
37. Hartmut Schmeck. Algebraic characterization of reducible flowcharts. *Journal of Computer and System Sciences*, 27(2):165 – 199, 1983.

38. P. Selinger. *A Survey of Graphical Languages for Monoidal Categories*, pages 289–355. Springer Berlin Heidelberg, Berlin, Heidelberg, 2011.
39. Vassiliki Sfyrla, Georgios Tsiligiannis, Iris Safaka, Marius Bozga, and Joseph Sifakis. Compositional translation of Simulink models into synchronous BIP. In *SIES*, pages 217–220, July 2010.
40. The Coq Development Team. *The Coq proof assistant reference*. INRIA, 2016. Version 8.5.
41. Stavros Tripakis, Christos Sofronis, Paul Caspi, and Adrian Curic. Translating Discrete-time Simulink to Lustre. *ACM Trans. Embed. Comput. Syst.*, 4(4):779–818, November 2005.
42. Fabio Zanasi. *Interacting Hopf Algebras- the Theory of Linear Systems. (Interacting Hopf Algebras - la théorie des systèmes linéaires)*. PhD thesis, École normale supérieure de Lyon, France, 2015.
43. Changyan Zhou and Ratnesh Kumar. Semantic Translation of Simulink Diagrams to Input/Output Extended Finite Automata. *Discrete Event Dynamic Systems*, 22(2):223–247, 2012.
44. Liang Zou, Naijun Zhany, Shuling Wang, Martin Franzle, and Shengchao Qin. Verifying Simulink diagrams via a Hybrid Hoare Logic Prover. In *EMSOFT*, pages 9:1–9:10, Sept 2013.