A Graph Rewriting Approach for Transformational Design of Digital Systems

Corrie Huijs
University of Twente, P.O. Box 217, 7500 AE Enschede, The Netherlands
tel.: +31 53 893697, fax: +31 53 4894590, e-mail: chuijs@cs.utwente.nl

Abstract

Transformational design integrates design and verification. It combines "correctness by construction" and design creativity by the use of pre-proven behaviour preserving transformations as design steps. The formal aspects of this methodology are hidden in the transformations. A constraint is the availability of a design representation with a compositional formal semantics. Graph representations are useful design representations because of their visualisation of design information. In this paper graph rewriting theory, as developed in the last twenty years in mathematics, is shown to be a useful basis for a formal framework for transformational design. The semantic aspects of graphs which are no part of graph rewriting theory are included by the use of attributed graphs. The used attribute algebra, table algebra, is a relation algebra derived from database theory. The combination of graph rewriting, table algebra and transformational design is new.

1. Introduction

High-level synthesis deals with the derivation of RTL (Register Transfer Level) implementations from behavioural specifications. The correctness of high-level synthesis is of importance and needs to be guaranteed in order to eliminate costly design iterations. A design methodology based on "correctness by construction" is, because of the complexity of designs, preferred above design methodologies in which either simulation or verification is used to guarantee the design correctness. In high-level synthesis the creativity of the designer is of great influence and therefore exploitation of the designer's experience and insights need to be possible. Transformational design incorporates "correctness by construction" and "interactive design" and therefore it not only leaves room for the designer's creativity but even stimulates it. The "correctness by construction" in transformational design is achieved by building up the design process out of small design steps, transformations, which are proven to be correct previously. The designer gets a large set of correctness preserving transformations and selects, possibly supported by a transformational design system, which transformations will be used and in what order. In this design approach decisions and their alternatives become clear which increases the insight of the designer [1], stimulating his creativity. The feasibility of transformational design, especially for DSP (Digital Signal Processing) design applications, becomes more and more clear [1, 2, 3]. Until now transformational design is used for area, time as well as power optimisation objectives. Figure 1 shows informally how the critical path in a specification can be reduced just by using a transformation based on the associativity of addition. The transformation rule is specified informally by figure 1b. The delay nodes, @, in figure 1 are assumed to be implemented by registers. Therefor the critical path in figure 1a is the path along the nodes 3, 5, 8 and 7 (critical path = maximum number of nodes between two @ nodes, @ node and input or @ node and output). In figure 1c there are several parallel paths of 3 nodes which together form the critical path.

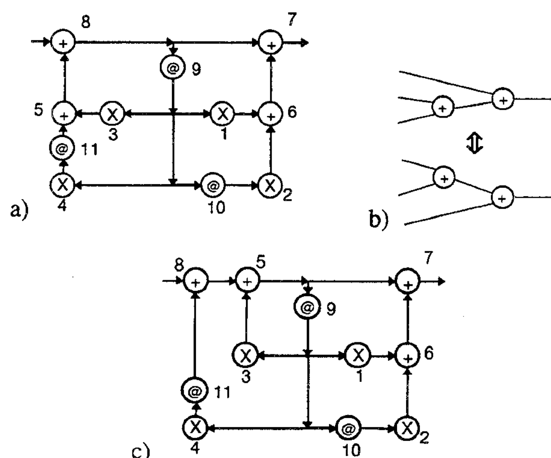


Figure 1:
a) A signal flow graph specification of a biquadratic filter (@ represents a delay node)
b) Transformation rule based on associativity of +
c) Result of applying the transformation rule on nodes 5 and 8 of the graph in a)

1089-6503/96 $5.00 © 1996 IEEE
Proceedings of EUROMICRO-22

Although transformational design is assumed to be formally well founded more attention often seems to be given to show its feasibility than to its formal aspects [2, 3, 4]. Correctness is related to the semantics of a design representation, the specified behaviour. To proof the behaviour preserving characteristic of transformations the representation(s) on which the transformations are defined need to have a formal semantics. Only a few of the commonly used specification languages have such a formal semantics. Different specification languages all have their own characteristics and often it is useful to combine several specification languages in one design process. The best approach for transformational design seems to be to define the transformations on an intermediate design representation to which the different specification languages can be converted [5, 6]. Often graph representations are used as internal or intermediate design representations [6, 7, 8, 9] because of the visualisation of design information they offer. Therefore it seems to be a good choice to base transformational design on graph representations. A formal semantic model for such graph representations, based on a relation algebra, is discussed in [10].

As also can be seen from the example in figure 1 transformations defined on graph representations are just replacements of selected subgraphs by new subgraphs. Graph rewriting is a specific theory developed during the last twenty years to describe the allowed replacements of subgraphs. In this paper the definitions and correctness aspects of transformations are related to graph rewriting.

2. Correctness aspects of Transformational design

The here presented research is part of a project that is aimed to show the feasibility of transformational design. The transformational design system TRADES [1] developed in this project is based on the signal flow graph representation SIL [6]. Case studies [1] have shown the usefulness of the transformational design aprroach. The design of a part of a HDTV system, the direction detector, could be improved by the use of the TRADES system mainly because design decisions became clear.

The research related to the formal aspects of transformational design is an essential part of this project Formal aspects are part of the principles founding this design approach. Moreover research related to these formal aspects is assumed to support the understanding of both the transformational design process and of the applicability of the various transformations. As discussed by McFarland [4], the formal aspects of transformational design methodologies for digital systems are not always well defined and the correctness of transformations is often based on intuition instead of proofs. Especially typing can cause problems. Proving the correctness of transformations is an essential element of the transformational design approach. Because of the large number of transformations a tool that supports the proofs is needed. PVS (Prototype Verification System of SRI) [11] has appeared to be useful support for correctness proofs of our transformations. Because PVS is based on interactive proving its use makes clear what are the problems in the proofs. Proofs that could not be given resulted in more insight then those easily been given. The strict typechecking of PVS was helpful in finding errors in the definition of some transformations. Beside errors unnecessary preconditions for some of the transformations were found.

The correctness proofs of transformations consist of:

1. the proof of the semantic correctness: the transformation has to preserve the observable behaviour
2. the proof of the syntactic correctness: the result of a transformation has to be a correct graph

The syntactic correctness can be based on graph rewriting theory and is being discussed in section 4 while the behaviour preserving characteristics are discussed in section 5. In both parts of the proofs the way transformations are defined is of importance. Where transformations need to be proven previously they need to be split in a transformation rule that can be proven at forehand and constraints for the application of these transformation rules. The constraints need to guarantee that the applications of transformation rules preserve correctness. From graph rewriting a general constraint is derived.

3. Graph representations

Several kinds of graph representations are used in design of digital systems, especially because of their visualisation of information. In the transformational design system CAMAD an extended Petri-net representation is used [9]. Several kinds of mixed or separate data flow and control flow representations are defined [7, 8, 12]. In respect with DSP design several kinds of signal flow graph representations are used [1, 3, 6, 13].

All graph representations are modelling hardware as hierarchical networks of processes or operational/relational units. This modelling reflects the concurrent nature of hardware blocks. The choice of operational/relational units and the modelling of their communication determines the class of systems which can be described. The use of relational units gives the

important possibility to describe (data) non-determinism. Ruby [14, 15] is a language based on relational structures for which a graphical representation is combined with a textual representation.

Modelling of communication is mainly based on the description of the temporal behaviour of systems. Petri-nets and data flow graphs are based on asynchronous communication represented by token passing. The temporal behaviour is described by the order of actions and no timing model is specified. Similarly in dependence graphs, often used in DSP design [16], modelling of timing is postponed and only a data dependency relation is specified. A dependence graph is based on single assignment. Each node represents an operation which is 'executed' only ones and therefore all operations are related to their own node. Time is explicitly introduced by folding of a dependence graph into a signal flow graph. The folding direction corresponds with the time axis. This means that signal flow graphs model synchronous systems by assuming a virtual clock. Each node in a signal flow graph is "executed" ones every clock period.

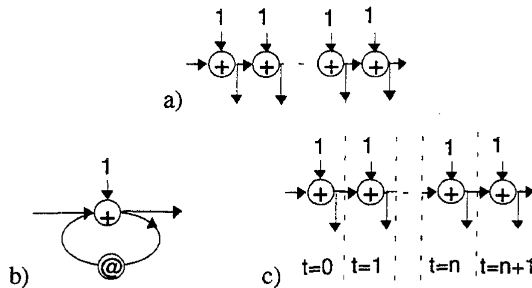


Figure 2:
a) A dependence graph for a counter
b) The signal flow graph received by projection and scheduling in horizontal direction.
c) Dependence graph with equi-time areas corresponding to the signal flow graph in b)

Communication of values between clockperiods is represented by delays which in [16] are given by annotation of edges. In figures 1 and 2 delay nodes (@) are used for modelling this communication of values between nodes "executing" in different clock periods. Delay nodes therefor are related to state. Their output can be said to correspond with the current state while their input correspond with a future state (which future state depends on the period of delay). The advantage of explicit conversion between dependence graphs and signal flow graphs is the explicit choice between time and space. Surprisingly this combination of dependence graphs and signal flow graphs is used only in DSP design while it seems more general applicable in design of synchronous digital systems. A signal flow graph (SFG) directly corresponds with a FSM. The difference between the two models is the integration of state variables and data variables in the SFG model and the separation of them in the FSM model. The advantage of the signal flow graph model is the possibility of hierarchy. Each node in a SFG can be a SFG itself. The FSM model is not hierarchical.

The CDFG (=Control Data Flow Graph) representation used in our transformational design approach, SIL, is based on the principles of dependence graphs and signal flow graphs. A special construction is defined for conditional execution of nodes, control. By the use of this construct control flow and data flow can be integrated. The formal definition of our CDFG representation is based on mathematical (hyper)graphs in order to be able to exploit graph rewriting theory. Our CDFG and its formal semantics as well as theorems of graph rewriting are modelled in the specification language of PVS in order to proof correctness of transformations.

4. Graph rewriting

In this section a short and (partly) informal introduction to graph rewriting theory is given. Special attention is given to the way in which transformation rules are defined and the preconditions that have to be satisfied before these rules can be applied to graphs.

A graph is a composition of two kinds of objects: nodes (or vertices) and edges (or arcs). Where a graph is composed of two kinds of objects two schools can be recognised in graph rewriting theory: one concentrating on replacements of nodes and one concentrating on the replacement of edges. Here only edge replacement, as originating from the Berlin school [17,18] is discussed.

Hypergraphs are graphs in which the edges, called hyperedges, are allowed to have more than one source and more than one target. Sources and targets of a hyperedge are given by lists of nodes. The ordering of sources and targets is explicitly part of the structure of hypergraphs. It nicely corresponds with parameter lists for function.

Definitions:

1. When X is a set, X^* is the set of all finite tuples (lists) of elements of X
2. A ***directed hypergraph*** is a 4 tuple $G = < N, E, s, t >$ in which
   N is the set of nodes and
   E the set of directed edges
   $s : E \rightarrow N^*$ a function that gives lists of sources of the edges
   $t : E \rightarrow N^*$ a function that gives lists of targets of the edges

Using arrows to graphically represent hyperedges can be confusing therefor hyperedges are graphically represented by ϒ in this paper. Nodes are represented by λ while elements of s and t are represented graphically, by $\rightarrow$. Functions are considered as sets of tuples and therefor can be said to have elements. To differ between s and t the elements of s are given by arrows starting at a node and elements of t by arrows pointing to a node (see figure 3b).

To define transformations (called derivations in graph rewriting) a formal definition is needed for the selection of a copy of a (hyper)graph in another (hyper)graphs. For this purpose graphmorfisms are defined. Graphmorfisms preserve structural properties. The sources of the image of an hyperedge are the images of the sources of the hyperedge and similar for targets:

Definition

A ***graphmorfism*** of (hyper)graph G to (hyper)graph G' is a tuple $m = \langle m_N, m_E \rangle$ such that

$m_N : N \rightarrow N'$,
$m_E : E \rightarrow E'$,
$s' \circ m_E = m_N \circ s$
$t' \circ m_E = m_N \circ t$

$$\begin{array}{ccccc} E & \overset{s}{\underset{t}{\rightrightarrows}} & N & \} & G \\ \downarrow m_E & & \downarrow m_N & = & \downarrow m \\ E' & \overset{s'}{\underset{t'}{\rightrightarrows}} & N' & \} & G' \end{array}$$

Transformation rules, productions in graph rewriting theory, are defined by 3 graphs L, K, R and 2 graphmorfisms l and r. L corresponds with the graph to be replaced and is called *left hand side*, R corresponds with the replacing graph and is called *right hand side* and K is called the interface graph and corresponds to the "boundary" along which the copy of graph L is cut out of a graph and along which the copy of R is glued into the same graph. Figure 4 gives an example of the graphical representation of such a transformation rule: the transformation rule describing associativity of addition.

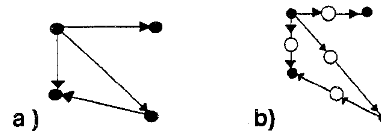


Figure 3:
a) "usual" representation of a graph
b) used representation of the same graph defined as hypergraph

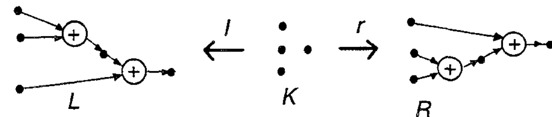


Figure 4: Graphical representation of a transformation rule describing associativity of addition.

The application of a transformation rule on hypergraph G is given by specifying a graphmorfism $g: L \rightarrow G$ and requiring that G can be viewed as the gluing of L and a context graph D along the nodes in $l(K)$. The result of such application of a transformation rule is the gluing of the context graph D and R along the nodes $r(K)$.

Figure 6 gives a graphical representation of a concrete transformation rule and its application specified by 6 graphs and 7 graphmorfisms which form a diagram like that in figure 4. Labels are used to describe the graphmorfisms: nodes (edges) with similar labels are mapped on each other by the graphmorfisms. The example is based on the transformation that was

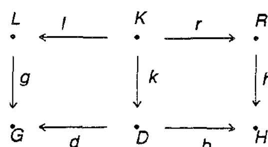


Figure 5: Diagram in which •'s represent (hyper)graphs and →'s represent graphmorfisms

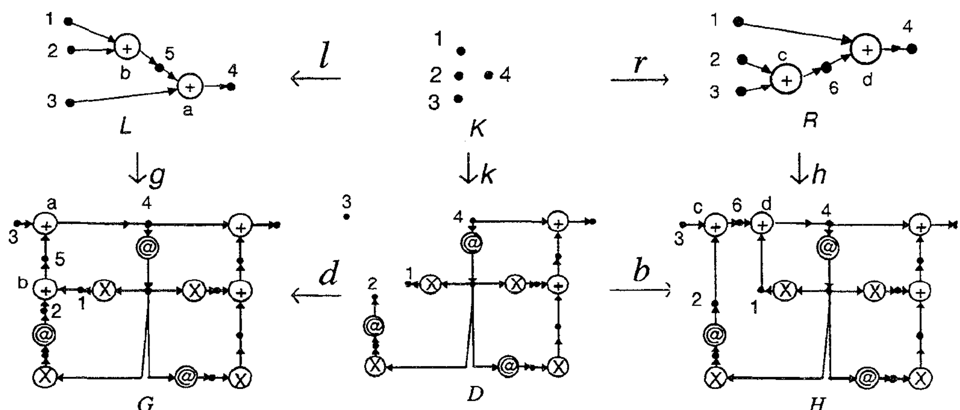


Figure 6: The transformation of figure 1 formally specified according to graphrewriting (All graphmorfisms are represented by labeling of the nodes and edges. Nodes (edges) with the same label are mapped onto each other).

already informally been given in figure 1.The algebraic specification of graph rewriting as defined by Ehrig [17] is useful in proofs. Of special importance is the "gluing condition". The idea behind this condition is that G can be interpreted as the gluing, specified by g and d, of the graph L and the context graph D along specified gluing items. These gluing items are specified by the interface graph K. The gluing items in L are elements of $l(K)$: $l(K) = GLUING$. The gluing items of D are the elements of $k(K)$ (see the diagram above). An element of L and an element of D are glued together if they are images of the same element of K, for respectively l and k. When $g(L)$ is removed from G several edges in the remaining graph, those which were connected to nodes that are removed, have dangling connections to sources or targets. Nodes of L that cause edges to get dangling connections are called "dangling". These nodes of L have to be gluing items in order to have D to be well formed. Elements of L which are together mapped, by g, onto the same element of G. are said to be identified and also need to gluing items. The gluing condition specifies sufficient conditions for G to be a well-formed graph that results from the gluing of L and D along K. Figure 7 shows two situations in which the gluing condition is not satisfied (the number labels specify l, the identity on these numbers):

Definition: gluing condition[17]

Lets L, K, G be (hyper)graphs, $l: L \rightarrow K$ and $g: L \rightarrow G$ be graphmorfisms and let :

$L \xleftarrow{l} K$, $g: L \downarrow G$, $k: K \downarrow D$, $G \xleftarrow{d} D$

$GLUING \equiv l(K)$

$DANGLING \equiv^{1} \{n \in N_L | \exists a \in (G - g(L)) : (g(n) = s_G(a)) \vee (g(n) = t_G(a))\}$

$IDENTIFICATION \equiv^{2} \{x \in L | \exists y \in L : (x \neq y) \wedge g(x) = g(y)\}$

then is G the gluing of L and $D = G - g(L)$ *along K if:*

$$DANGLING \cup IDENTIFICATION \subseteq GLUING$$

5. Transformational design based on graph rewriting

Signal Flow Graphs used as design representations specify behaviour but also include implementation suggestions. In the design process the specified behaviour needs to be preserved and the implementation suggestion needs to be refined. A signal flow graph can be modelled by a hypergraph together with a function

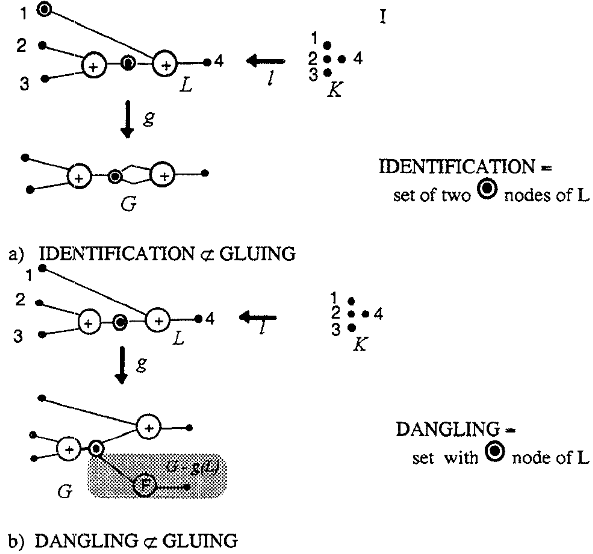


Figure 7: Two situations in which the gluing condition is not satisfied

that maps each hyperedge to the operation/relation it represents, its behaviour. A mathematical representation of the specified behaviour is necessary to proof design steps to be behaviour preserving. This mathematical representation has to correspond with the formal semantics of the signal flow graphs. Compositionality of this formal semantics is very important in transformational design: the behaviour of a composition, a graph, needs to be the composition of the behaviour of the components, nodes and edges.

In the above discussion of graph rewriting only attention was given to syntactic aspects. The *gluing condition* defines preconditions for syntactic correctness of transformations. Moreover this gluing condition appears also to be useful in proofs of behaviour preserving properties [19]. The semantical aspects are integrated in the above discussed model by the use of *attributed multi-ported graphs* [20]. Attributes of a graph are used to define its semantics in a denotational way which offers the needed compositionality. The attribute functions correspond to valuation functions of a denotational semantics [21]. Attributes correspond with the semantics of the nodes and/or edges.

Definition:

An ***attributed graph*** is a graph G together with one or more (atribute)functions of type[3] $E_G \rightarrow \Sigma_1$ or type $N_G \rightarrow \Sigma_2$ for some algebras Σ_i

[1] If G is a graph a reference to its elements is given by the use of the index: $_G$. For example N_G is the set of nodes of G

[2] The used notation is a bit sloppy. For a graph G the meaning of $x \in G$ is $(x \in N_G) \vee (x \in E_G)$

[3] The type of a function specifies its domain and codomain. The domain is given before the $\rightarrow$ and the codomain behind it.

5.1 Table Algebra as attribute algebra

The behaviour specified by a graph is a data-relation on its external nodes, its boundary. Tables are defined as representations of relations and can be viewed as a generalisation of the truth table concept. A table is a set of functions with a common domain [22]. This common domain is called the heading of the table. If a table is used as representation of the behaviour of a hyperedge then the heading of the table consists of the nodes to which this hyperedge is connected. A table corresponds with a predicate on functions (specifying if the function represents an allowed combination of data-values to be observed at the nodes). For example the behaviour of hyperedges instantiating addition operations, *add* and *add1* of figure 8, can be given by:

$$Table(add) = \{ f \mid [dom(f) = \{a,b,e\}] \wedge [f(a) \in [1,2]] \wedge [f(b) \in [1,2]] \wedge [f(e) \in INT] \wedge [f(e) = f(a) + f(b)] \}$$

$$Table(add1) = \{ g \mid [dom(g) = \{c,e,d\}] \wedge [g(c) \in [1,2]] \wedge [g(e) \in [1,2]] \wedge [g(d) \in INT] \wedge [g(d) = g(c) + g(e)] \}$$

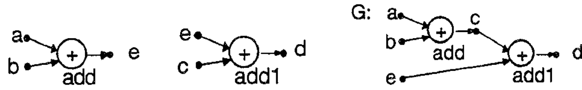


Figure 8: graph *G* as composition of two subgraphs

In case of *add* the types, the set of allowed observable values, of the nodes a and b is restricted to the interval [1, 2] of integers. These tables can also be given graphical which makes clear why these sets of functions are called tables. Each function in the set corresponds with a row in the graphical table:

Table(add)=

a	b	c
1	1	2
1	2	3
2	1	3
2	2	4

Table(add1)=

c	e	d
1	1	2
1	2	3
2	1	3
2	2	4

The behaviour, semantics, of a graph needs to be a composition of the behaviour of its components (in order to use the graph representation for transformational design). The natural join which is known from database theory is used as the composition operator. The natural join corresponds with the conjunction of predicates. The natural join combines function elements, rows, of tables into a function element, row, of a new table. Functions can only be combined if for shared domain elements the original functions deliver the same function value. In case of the natural join of *Table(add)* and *Table(add1)* the function value of c needs to be 2. The first row of *Table(add)* can be combined with the last two rows of *Table(add1)*:

$Table(G) = Table(add) \bowtie Table(add1) =$

a	b	c	e	d
1	1	2	1	3
1	1	2	2	4

The observable behaviour of G is the data relation on its external nodes: $\{a, b, e, d\}$. This observable behaviour is received by projection ⫦, restricting each element of *Table(G)* to the external nodes:

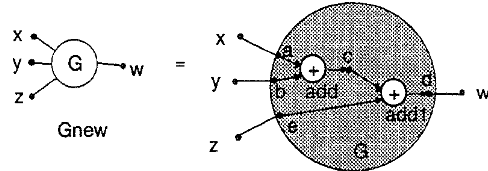
observable behaviour of G=

$Table(G)$ ⫦ $\{a, b, e, d\}$ =

a	b	e	d
1	1	1	3
1	1	2	4

By the use of a *table* to represent the data-relation on the (external)nodes of G no order on these nodes is used. No meaning is given to the geometrical placement of the hyperedges. There is a difference between hyperedges and graphs: the interface points of a graph are not ordered the interface points of a hyperedge are ordered. Relations given as sets of tuples can be used to represent behaviour of hyperedges. Tables are more suitable to represent the behaviour of graphs if no meaning has to be given to the geometrical placement of the components.

In case a graph is used as specification of the behaviour of a hyperedge an order on its external nodes need to be given. If G, of figure 8, is used as specification of the behaviour of a hyperedge the attributes of the above table, $\{a, b, e, d\}$ become formal parameters and are replaced by the sources and targets of the hyperedge (In the example below: $\{x, y, z, w\}$).



Table(Gnew)=

$Table(G) \infty \{\langle x,a\rangle, \langle y,b\rangle, \langle z,e\rangle, \langle w,d\rangle\}$ =

x	y	z	w
1	1	1	3
1	1	2	4

A renaming operation, ∞, is used to derive the behaviour of a hyperedge, from the table of its specifying graph. The renaming function is defined by the ordering of the external nodes of the specifying graph and the source and target functions.

Definitions of Table Algebra:

1. A ***function*** f is a set of tuples such that: $\forall a,x,y$: $(\langle a, x\rangle \in f) \wedge (\langle a, y\rangle \in f) \Rightarrow (x = y)$. The predicate *Function* therefore is defined by:
   $Function(f) \Leftrightarrow$
   $[\forall a,x,y: (\langle a, x\rangle \in f) \wedge (\langle a, y\rangle \in f) \Rightarrow (x = y)]$

2. For a set A a function f is said to be a ***function on** A* iff $[\forall a \in A: \exists x: <a, x> \in f]$. The predicate *Function-on* therefore is defined by: $[Function(f) \wedge \forall a \in A: \exists x: <a, x> \in f] \Leftrightarrow Function_on(f, A)$
3. The *restriction of a function f* to a set X, $f \upharpoonright X$, is defined by: $f \upharpoonright X = \{<x, y> \mid (<x, y> \in f) \wedge (x \in X)\}$
4. A ***Table on** A* is a set T of functions on A. A is called the **set of attributes of T**, $Att(T)$.
5. The ***natural join***, $\bowtie$, of two tables, T_1 and T_2, is a table on the union of the sets of attributes of *T1* and *T2* defined by:
   $T1 \bowtie T2 = \{f \mid Function_on(f, (Att(T1) \cup Att(T1))) \wedge (f \upharpoonright Att(T1) \in T1) \wedge (f \upharpoonright Att(T2) \in T2)\}$
6. The ***natural join over a set of tables*** $\{T_a \mid a \in A\}$ is defined by:
   $$\bowtie_{a \in A} T_a = \{t \mid Function_on(t, \Upsilon_{a \in A} Att(T_a)) \wedge \forall a \in A\ [(t \upharpoonright Att(T_a) \in T_a)\}$$
7. For a set B and a table T ***the restriction of T to B***, $T \Vert\!\upharpoonright B$, is defined by:
   $T \Vert\!\upharpoonright B = \{f \mid \exists t: [(t \in T) \wedge (f = t \upharpoonright B)]\}$
8. For a function r and a table T the ***renaming of T by** r*, $T \infty r$, is defined by:
   $T \infty r = \{f \mid \exists t: [(t \in T) \wedge (f = t \circ r)]\}$

5.2 Attributed graphs as design representation

Tables can be seen as instantiations of relations, just as hyperedges in a hypergraph. An attribute function on a hypergraph which maps each hyperedge to a table is also a valuation function which maps hyperedges on their semantics. The semantics of a hypergraph, the observable behaviour, is the natural join of the tables of its hyperedges restricted to the external nodes. Hypergraphs attributed by tables are useful design representations in transformational design becaus of the above defined *Table Algebra.* They combine representation of behaviour and structure and their formal semantics which is based on table algebra is compositional. Dependence graphs as well as signal flow graphs can be specified as hypergraphs attributed by tables. The difference between these two kind of graphs is the set of allowed relations for the hyperedges. In signal flow graphs a delay relation is allowed in dependence graphs not. The timing model used is represented by the allowed relations for the hyperedges. The types of values allowed at a node are mainly determined by the hyperedges to which the node is connected. Types of (external) nodes which are part of the specification can also be given by hyperedges.

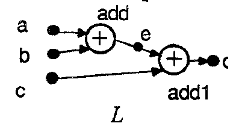
In order to achieve a CDFG representation we have extended the definition of hypergraphs with a control structure: conditions. Only if at all nodes connected with the condition input of a hyperedge the value "true" can be observed the relation of the hyperedge holds (its operation is executed). Besides this the definition of hypergraphs is extended by the explicit definition of the external nodes, *In* and *Out*, and the mapping of hyperedges onto the relation they instantiate. Relations and tables are sets defined by predicates and therefor can be easily specified in PVS. The allowed relations for hyperedges are assumed to be given by the set *REL*:

Definition:

A *spec_graph* over *REL* is a 8-tuple
$<Ops, Vars, s, t, cond, rel, In, Out>$ in which

Ops	is the set of operations (the hyperedges)
Vars	is the set of communicated variables (nodes of a hypergraph)
$s: Ops \rightarrow Vars^*$	a function that gives the source variables of the operations
$t: Ops \rightarrow Vars^*$	a function that gives the target variables of the operations
$cond: Ops \rightarrow Vars$	a function that gives the condition variables of the operations
$rel: Ops \rightarrow REL$	a function that gives a reference to relations defining operations
$In \subset Vars$	set of input variables of the graph
$Out \subset Vars$	set of output variables of the graph

The *lefthand side L* of the transformation rule of figure 3 is in our representation given by ($+ \in REL$; $\mapsto$ specifies the actual values of tuple-elements):



L

$$
\begin{aligned}
L = \{\ & Ops && \mapsto \{add, add1\},\\
& Vars && \mapsto \{a, b, c, d, e\},\\
& s && \mapsto \{add \mapsto <a, b>, add1 \mapsto <e, c>\},\\
& t && \mapsto \{add \mapsto <e>, add1 \mapsto <d>\},\\
& cond && \mapsto \emptyset,\\
& rel && \mapsto \{add \mapsto +, add1 \mapsto +\},\\
& In && \mapsto \{a, b, c\},\\
& Out && \mapsto \{d\}\}
\end{aligned}
$$

A sufficient condition for transformations to be behaviour preserving can be based on the graphs and graphmorfisms of the transformation rule:

$$Table(L) \infty\, l_N = Table(R) \infty\, r_N$$

which means that both tables are equal after renaming to the interface graph. The definition of graphmorfisms is slightly changed in order to preserve also condition structures related to hyperedges.

6. Conclusions

A formal framework for transformational design based on graph rewriting theory is presented. Transformations are defined as applications of transformation rules, graph rewriting rules. The gluing condition from graph rewriting theory unifies constraints for syntactic correctness of transformations. Semantical aspects are embedded in graph rewriting by the use of attributed multi-ported graphs. Interface graphs, as defined in graph rewriting theory, appear to be useful in the definition of transformation rules and their behaviour preserving characteristic. Table algebra is used as atribute algebra and therefor also as semantic algebra of the denotational semantics of the design representation. PVS could be used to support the proofs of transformations defined on the defined design representation.

References

[1] P.F.A. Middelhoek, G. E. Mekenkamp, E. Molenkamp, Th. Krol, *A Transformational Approach to VHDL and CDFG Based High-Level Synthesis: a Case Study*, Proc. of CICC 95, pp. 37-40, Santa Clara, Ca, May 1995. (also: http//wwwspa.cs.utwente.nl/aid/trades/trades.html)

[2] M. Potkonjak, J. Rabaey, *Optimizing resource utilization using transformations,* IEEE Trans. on CAD, Vol. 13. No.3 March 1994, pp 277-292.

[3] A. P Chandrakasan, M. Potkonjak, et al. *Optimizing power using transformations*, IEEE Trans. on CAD, Vol. 14. No.1 January 1995, pp. 12-31.

[4] M.C. McFarland SJ., *Formal Analysis of Correctness of Behavioral Transformations*, in; Formal Methods in System Design Vol. 2, Number 3, June 1993, pp 231-257

[5] J.T.J van Eijndhoven, G.G. de Jong, L. Stok, *The ASCIS data Flow Graph: Semantics and Textual Format,* Research report of Eindhoven University of Technology no 91-E-251, 1991.

[6] Th. Krol et al, *The SPRITE Input Language, An intermediate format for High Level Synthesis*, in: Proc. of EDAC 92, Brussels, 16-19 March 1992, pp 186-192.

[7] R. Camposano, *Behaviour-preserving transformations for high-level synthesis*, in: M. Leeser and G. Brown *Hardware Specification, Verification and Synthesis: Mathematical Aspects*, LNCS 408, (Springer-Verlag, Berlin Heidelberg,1989) pp.106-127.

[8] R. Camposano and R.M Tabet, *Design representation for the synthesis of behavioral VHDL models,* Proc. 9th Int.'l Conf on CHDL, J.A Darringer and F.J. Rammig (eds.), Elsevier Science Publishers B.V, June 1989, pp 49-58.

[9] Z.Peng and K. Kuchinski, *Automated Transformation of Algorithms into Register-Transfer Level Implementations,* IEEE Trans. on CAD Vol. 13, No. 2. Februari 1994, pp 150-166.

[10] C. Huijs and Th Krol, *Relational Semantics for Flow Graph Representations as basis for Transformational Design of Digital Systems,* in: Proceedings of the workshop on Design Methodologies for Microelectronics and Signal Processing, Gliwice-Cracow Poland, 20-23 October 1993, pp 21- 29. (also:http//wwwspa.cs.utwente.nl/aid/trades/trades.html)

[11] S.Owre, N. Shankar and J.M. Rushby, *Userguide for the PVS specification and Verification system*, Computer Science Laboratory, SRI, Menlo Park, CA, Feb. 1993.

[12] D. Gajski, *Introduction to High-Level Synthesis,* IEEE Design & Test, Vol. 11, Nr 4, Winter 1994, pp.44-54

[13] H. Samson, L. Claesen and H. DeMan, *Synguide: An environment for doing interactive Correctnesss preserving transformations,* Eggermont et al (eds) Proc. of IEEE VLSI Signal processing VI-workshop 1993.

[14] G. Jones and M.Sheeran, *Circuit design in Ruby*, in J. Staunstrup(Eds), *Formal Methods for VLSI design,* North-Holland, pp13-70

[15] W. Luk, *Systematic pipelining of processor array,* in G.M. Megson(Eds) *Transformational approaches to systolic design* Chapman&Hall 1994, pp77-98.

[16] S.Y. Kung, *VLSI Array processors*, Prentice Hall 1988.

[17] H. Ehrig, M. Korff, M. Löwe, *Tutorial introduction to the algebraic approach of graph grammars based on double and single pushouts,* in H. Ehrig, H.-J. Kreowski, G. Rozenberg(Eds) *Graph-Grammars and their Application to Computer Science* , LNCS532, (Springer-Verlag, Berlin Heidelberg,1991) pp.24-37.

[18] A. Habel H.-J. Kreowski, *May we introduce to you: Hyperedge replacement*, in H. Ehrig, M. Nagl, G. Rozenberg, A. Rosenfeld(Eds) *Graph-Grammars and their Application to Computer Science* , LNCS 291, (Springer-Verlag, Berlin Heidelberg,1987) pp.15-26.

[19] C. Huijs, *"Transformational Design of Digital Systems related to Graph Rewriting"*, Proceedings of the Workshop on Design Methodologies for Microelectronics, Smolenice Castle Slovakia September 11-13 1995:pp 297-305. (also: http//wwwspa.cs.utwente.nl/aid/trades/trades.html)

[20] M. Löwe, Martin Korff, A. Wagner, *An algebraic framework for the transformation of attributed graphs,* in M.R. Sleep, M.J. Plasmeijer and M.C.J.D. van Eekelen(Eds), *Term Graph Rewriting*, John Wiley& Sons Ltd, 1993, pp 185-199.

[21] Mosses, P.D., *Denotational Semantics*, in: J. van Leeuwen eds.*Formal Models and Semantics, Handbook of Theoretical Comp. Sc. Volume B* (Elsevier Science Publishers, North-Holland,1990) pp. 577- 631.

[22] E.O. de Brock, *Foundations of Semantic Databases*, Prentice Hall, 1995