Abstract. For high-level synthesis transformational design is a promising design methodology which combines correctness by construction and interactive design. In this design methodology the design steps are behaviour preserving transformations of one design representation into another. Because of the importance of visualisation of design-information several kinds of graphs are used as design representations. Transformational design based on graph representation is closely related to rewriting of (sub)graphs. In this paper the formal aspects of transformational design are related to graph rewriting theory. It is shown how a formal framework for transformational design can benefit from graph rewriting theory. Especially preconditions for the application of transformation rules can be based on generally formulated preconditions from graph rewriting theory. Moreover a general graph concept unifies graph representations and a formal framework for transformational design based on this general graphs and their rewriting unifies several transformational design approaches based on different graph representations. Unfortunately semantical aspects, the behaviour preserving aspects, which are of great importance for transformational design are not a part of graph rewriting theory. A consistent embedding of these behaviour preserving aspects in the formal framework based on graph rewriting is defined and presented in this paper. The discussion of formal aspects is accompanied by an informal discussion.

Keywords: specification and verification, transformational design, graph representations, graph rewriting, behaviour preserving transformations

1. Introduction

High-level synthesis deals with the derivation of “RTL level implementations from behavioural specifications. The correctness of high-level synthesis is of importance and need 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 designers experience and insight need to be possible. Transformational design incorporates “correctness by construction” and “interactivity” and therefore it not only leaves room for the designers creativity but even stimulate 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 these correctness preserving transformations and, possibly supported by a transformational design system, he selects 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 [Middelhoek95], stimulating his creativity. The feasibility of transformational design, especially for DSP design applications, becomes more and more clear[Middelhoek95, Potkonjak94, Chandrakasan95]. Transformational design until now 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.
Correctness preserving transformations or behaviour preserving transformations are the core of transformational design. Correctness is related to the semantics of a design representation, the described/specified behaviour. To prove 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. Choosing just a specification language with a formal semantics as basis for transformational design is not the best solution. The different specification languages all have their specific 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 for which a formal semantics is defined and to which the different specification languages can be converted [Eijndhoven91, Kro92]. Often graph representations are used as internal or intermediate design representations, [Camposano89a+b, Kro92, Peng94] because the visualisation of design information they offer. Therefore it seems to be a good choice to base transformational design on graph representations. A general concept for the formal semantics of such graph representations, based on a relational algebra, is discussed in [Huijs93].

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. For this graph rewriting a specific theory is developed during the last twenty years. From this theory we can learn how transformations can be defined as applications of transformation rules and how these transformation rules and rules for their application can be defined. Moreover the formal framework of graph rewriting can be used as basis for the formal framework of transformational design based on graph representations. Because such a formal framework for transformational design is based on a general graph it unifies the transformational design approaches based on different graph representations.

Several of the transformations used until now in transformational design, see also figure 1, are based on well-known algebraic laws (associativity, distributivity, etc.). These transformations can be said to be based on term rewritings, replacements of (sub)expressions. The proofs of the correctness of transformations on graphs which are based on these term rewritings need to be trivial consequences of the correctness of term rewriting rules. It is shown in [Sleep93] that term rewriting can be embedded in graph rewriting by term graph rewriting.

The here presented research is part of a project that is aimed to show the feasibility of transformational design. The transformational design system TRADES [Middelhoek95] developed in this project is based on the signal flow graph representation SIL [Kro92]. The research related to the formal aspects of transformational design is assumed to support both the understanding of the transformational design process and the understanding of the applicability of the various transformations.

2. Graph representations

Several kinds of graph representations are used in design of digital systems. In the transformational design system CAMAD an extended Petrinet representation is used [Peng94]. Several kinds of mixed or separate data flow and control flow representations are defined [Gayski, Camposanoa+b]. In respect with DSP design several kinds of signal flow graph representations are used [Samson93, Potkonjak94]. What all these representations have in common is that they define a kind of graph, not necessarily a mathematical graph, in which nodes represent operations and the composition of these operations is described by
edges. A more general approach is got when instead of nodes representing operations nodes are used which represent relations. This gives the possibility to describe non-determinism. In this approach graphs can be said to be relational structures. Ruby [Jones90, Luk94] is a language in which graphical representation of relational structures is combined with textual representation. Ruby is a nice relational formalism especially useful for regular structures but mixes up structural and behavioural information. Most of the used graph representations can easily been reformulated in such a way that they are based on mathematical (hyper)graphs. An advantage of this is that mathematical theories can be applied on these representations. But it also results in unification of the different graph representations.

3. Transformational design

In showing the feasibility of transformational design often the correctness of the transformations is based on intuition instead of on proof. As shown by McFarland[McFarland93] transformations which are intuitively correct do not have to be correct in all occasions they are used. Especially typing can cause problems. Proving the correctness of transformations is an essential element of the transformational design approach. But even more important is that it supports the understanding of the transformational design process and the understanding of the applicability of the various transformations. For these reasons the proofs of transformations are part of our research. Correctness proofs of transformations consist of:

1. the proof of the syntactical correctness: the result of a transformation has to be a correct graph
2. the proof of the semantical correctness: the transformation has to be behaviour preserving

In section 4 the syntactical correctness will be discussed while in section 5 the behaviour preserving characteristics are discussed. 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 a rule, constraint, for the application of these transformation rules which guarantees that their applications preserves the correctness. Moreover the way transformation rules are defined have to include the possibilities of:

- refinement
- abstraction
- parameterisation.

These requirements are illustrated by transformations based on common subexpression elimination, a well known transformation of compiler optimisation. Some examples of different appearances of common subexpressions in graphs are given in figure 2 and are depending on the number of inputs and/or outputs and on the difference of the relation/operation represented by the nodes.

![Figure 2: Some examples of graphs with “common subgraphs”](image)

By abstraction is meant that we want to reduce all these graphs by a rule in which only one input and one output is shown and which informally can be given as:

Parametrisation is also demonstrated by this example: the relation/operation is given by the parameter \( f \). Refinement can be compared with the replacement of a bitstring addition by a number of Boolean additions.

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 which 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). Were 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[Habel87,Ehrig91] is discussed.
**Definition**

A **directed graph** is a 4 tuple $G = \langle N, E, s, t \rangle$ in which

- $N$ is the set of nodes and
- $E$ the set of directed edges
- $s : E \to N$ gives the source of the edges
- $t : E \to N$ give the target of the edges

**Figure 3:** a) “usual” representation of a graph  
   b) the here used representation of the same graph

Usually nodes are graphically represented by $\bullet$ and directed edges by $\rightarrow$ (figure 3 a). Here we use a slightly different representation which can be easier generalised: nodes are also represented by $\bullet$ but edges are represented by $\circ$. Also $s$ and $t$ are represented graphically, by $\rightarrow$. 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). In fact for simple graphs the usual notation can be seen as a short hand for the here chosen representation.

A more general kind of graphs are hypergraphs in which edges can have more than one source (a list of sources) and more than one edge (a list of edges).

**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 $G = \langle N, E, s, t \rangle$ in which
   - $N$ is the set of nodes and
   - $E$ is the set of hyperedges
   - $s : E \to N^*$ gives the sources of the hyperedges
   - $t : E \to N^*$ gives the targets of the hyperedges

To define transformations (called derivations in graph rewriting) a formal definition is needed for the selection of a copy of a graph in another graph (embedding) is needed. For this purpose graph morphisms are defined. Graph morphisms preserve structural properties.

**Definition**

A **graph morphism** of graph $G$ to graph $G'$ is a tuple $m = \langle m_N, m_E \rangle$ such that $m_N : N \to N'$, $m_E : E \to E'$, $s' \circ m_E = m_N \circ s$, and $t' \circ m_E = m_N \circ t$.

Transformation rules, productions in graph rewriting theory, are defined by 3 graphs $L$, $K$, $R$ and 2 graph morphisms $l$ 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 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 graph morphism $g : L \to G$ 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 5:** Diagram in which $\bullet$’s represent (hyper)graphs and $\rightarrow$’s represent graph morphisms

---

1. Functions are considered as sets of tuples and therefore they can be said to have elements
Figure 6 gives a graphical representation of a concrete transformation rule and its application specified by 6 graphs and 7 graphmorphisms which form a diagram like that in figure 5. Labels are used to describe the graphmorphisms: nodes (edges) with similar labels are mapped on each other by the graphmorphisms. The example is based on the transformation which was already informally been given in figure 1.

Diagram like the one given in figure 5 are well known from category theory. The algebraic specification of graph rewriting as defined by Ehrig is based on this kind of diagrams and their properties [Ehrig91]. This algebraic specification of graph rewriting is useful in proofs. However the conditions to be satisfied by the graphs and the graphmorphisms in order to get well-formed graphs as results of transformations can be also given more operational as is done by the “gluing condition”[Ehrig91]. 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 \( l(K)\equiv \text{GLUING} \). The gluing items of \( D \) are the elements of \( k(K) \) (see the diagram below). 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 because they also have to belong to \( D \) in order to have \( D \) to be well-formed. Another special kind of elements of \( L \) are those which are together with a different element of \( L \) mapped, by \( g \), onto the same element of \( G \). They can be said to be identified. Where the interface graph was defined to specify the identification of elements (primarily the elements of \( L \) and \( D \) to be glued) the identified elements of \( L \) has to be images of elements of \( K \) by \( l \), gluing elements. The gluing condition specifies necessary and sufficient conditions for \( G \) to be a well-formed graph that results from the gluing of \( L \) and \( D \) along \( K \).

**Definition: gluing condition**[Ehrig91]

Let \( L, K, D, G \) be graphs and \( l: L \rightarrow K, k: K \rightarrow D, d:D \rightarrow G \) and \( g: L \rightarrow G \) be graphmorphisms and let:

\[
\text{GLUING} \equiv l(K)
\]

\[
\text{DANGLING} \equiv \{ n \in N_c \mid \exists a(G-g(L)): (g(n)=s_c(a)) \lor (g(n)=t_c(a)) \}
\]

\[
\text{IDENTIFICATION} \equiv \{ x \in G \mid \exists y \in G: (x \neq y) \land g(x)=g(y) \}
\]

Then the diagram represented by these graphs and morphisms is a gluing diagram iff:

\[
\text{BOUNDARY} = \text{DANGLING} \cup \text{IDENTIFICATION} \subseteq \text{GLUING}
\]

---

2 \( G \) is the disjoint union of \( D \) and \( L \) together with an equivalence \( \equiv \) relation based on \( K: G=D \oplus L/\equiv \)

3 If \( G \) is a graph a reference to its elements is given by the use of the index \( c \). For example \( N_c \) is the set of nodes of \( G \)

4 The used notation is a bit sloppy. For a graph \( G \) the meaning of \( x \in G \) is \( (x \in N_c) \lor (x \in E_c) \)
A few examples will illustrate the gluing condition and the included definitions of boundary, dangling and identification. Again labels are used to represent which nodes are mapped on which other nodes by the graph morphisms $g_i$. GLUING = {nodes of $L$ labelled by: $a$, $b$, $c$ and $d$} 

In $G_1$, $b$ and $c$ are identified but they also belong to DANGLING as well as to GLUING. The graph morphism $g_1$ satisfies the gluing condition while $g_2$ and $g_3$ doesn’t satisfy it. Because of the specification of $g_3$, the node of $L$ labelled by $e$ belongs to DANGLING, but not to GLUING. By the specification of $g_3$, the nodes labelled $c$ and $e$ are identified, and therefore both belong to IDENTIFICATION while $e$ doesn’t belong to GLUING.

Figure 7: examples of graph morphisms $g : L \to G$

5. Transformational design based on graph rewriting

It is shown that graph rewriting theory defines preconditions, the gluing condition, for syntactical correctness of transformations. In transformational design more is needed: behaviour preserving transformations. The behaviour preserving property corresponds with the semantic correctness of transformations. In this section the behavioural aspects are integrated in the above discussed model of graphs by the definition of attributed multi-ported graphs. Attributes of a graph are used to define its semantics in a denotational way. Moreover a definition of behaviour preserving transformations will be given together with a sufficient condition for obtaining behaviour preserving transformations. An informal discussion precedes the definitions.

Graphs used as design representations specify behaviour. By behaviour here, similar to [Huijs93, Peng94] a combination of a partial order relation and a (functional) data-relation between the inputs and outputs, external ports, is meant. For example in the graph of figure 6 the nodes labelled with $x$, $y$ are wanted to be distinguished as inputs and the node labelled $z$ as output. External ports need to be defined. For this purpose the definition of multi-ported graphs[Habel87][Habel uses the name multi pointed instead of multi-ported.], also called graphs with distinguished nodes can be used.

Definition:
A multi-ported directed hypergraph is a 4-tuple $G = < N, E, s, t, P_{\text{in}}, P_{\text{out}} >$ in which
- $N$ is the set of nodes and $E$ the set of hyperedges
- $s : E \to N^*$ gives the sources of the hyperedges
- $t : E \to N^*$ gives the targets of the edges
- $P_{\text{in}} \subseteq N$ the set of input ports
- $P_{\text{out}} \subseteq N$ the set of output ports

Intuitively it is clear what the graph of figure 8 specifies. The data relation and the order relation on $\{x, y, z\}$ specified by this graph are composed out of the data- and order-relation of the edges. It can be said that each edge represents the instantiation of an abstract data- and an abstract order-relation.

Abstract relations become concrete by instantiating them as hyperedges in a graph, they become relations between given nodes. Lets the relation NOT be the NOT relation for Boolean values: $\{<0,1>, <1,0>\}$. This relation is often represented by a truth table:

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>in</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>out</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>
Each row in this truth table can be represented by a function on \{in, out\} and so the truth table, the relation, can be described by a set of functions. A set of functions with a common domain is defined to be a table [Brock84]. As is shown in [Huijs 93] this table definition generalises the truth table concept. A function which maps each hyperedge of a graph onto a table representing the data-relation it specifies can be seen as an attribute function. It may be clear that for the graph of figure 9 the attribute function has to map not1 and not2 to respectively:

This means that these tables are found by renaming, \(\infty\), of the column headings of the earlier given truth table. A table algebra [Brock84, Ullman82] can be used to derive data-relations of graphs out of data-relations of their edges. The data-relation for the whole graph of figure 8 is a composition of the hyperedge attributes, tables. The composition operator is the natural join, \(\times\), which combines those rows from the operand tables that result in a function: the rows from the operand tables needs to share the same value in columns with equal heading. This natural join is illustrated for the tables attributed to the subgraphs in figure 9 respectively 10:

The data-relation of the complete graph of figure 8 is given by:

The data-relation restricted to the external ports is derived by the use of the restriction, projection, operator \(\uparrow\), which just projects a table to a subset of its columns:

The hyperedges not1 and not2 of the graph in figure 8 can be seen as common subexpressions. A transformation rule to reduce the common subexpression to only one is \(p\):

Application of this rule on the graph in figure 6 gives the transformation \(G \xrightarrow{p} H\) as shown in figure 11. In this notation for transformations the context graph is left implicit and only the transformation rule \(p\) and the graphomorism \(g\) are given.
can also easily been shown to be equal. Therefore the transformation \( G \xrightarrow{\phi, p} H \) is said to be behaviour preserving. The behaviour of the graphs \( G \) and \( H \) are the combination of their data-relations, tables, and order relations both restricted to their external ports. The transformation is behaviour preserving notwithstanding that the tables of \( L \) and \( R \) are clearly unequal (different numbers of columns). Although a sufficient condition for transformations to be behaviour preserving can be based on the graphs and graphmorphisms of the transformation rule: \( \text{Table}(L) \xrightarrow{l} \text{Table}(R) \xrightarrow{r} \) which means that both tables are equal after renaming to the interface graph. A similar condition can be given for the order relations of \( L \) and \( R \): \( \text{Order}(L) \xrightarrow{l} \text{Order}(R) \xrightarrow{r} \). Therefore a transformation rule is defined to be behaviour preserving if the behaviour of its left hand side renamed to the interface graph is equal to the behaviour of its right hand side renamed to the interface graph.

In Table 12 the definitions are given of the operations informally introduced in the above discussion.

### Definitions of Table Algebra:

1. A **function** \( f \) is a set of tuples such that: \( \forall a, x, y: (<a, x> \in f) \land (<a, y> \in f) \Rightarrow (x = y) \)
2. For a set \( A \) a function \( f \) is said to be a function on \( A \) iff \( \forall a, x, y \in f ) : [\text{Function}(f) \land \forall a: \exists x: <a, x> \in f] \Rightarrow \text{Function}(f, A) \)
3. The restriction of a function \( f \) to a set \( X, f \upharpoonleft X \), is defined by: \( f \upharpoonleft X = \{ f | X \land (f \in X) \} \)
4. A **Table on \( A \)** is a set \( T \) of functions on \( A. A \) is called the set of **attributes** of \( T, \text{Att}(T) \)
5. The **natural join**, \( T_1 \bowtie T_2 \), of two tables, \( T_1 \) and \( T_2 \), is a table on the union of the sets of attributes of \( T_1 \) and \( T_2 \)
6. For a set \( B \) and a table \( T \) the **restriction of \( T \) to \( B \)**, \( T \upharpoonleft B \), is defined by:

\[
T \upharpoonleft B = \{ f | \exists (t \in T) \land (f = t \upharpoonleft B) \}
\]

7. For a function \( r \) and a table \( T \) the **renaming of \( T \) by \( r \)**, \( T \circ \circ r \), is defined by:

\[
T \circ \circ r = \{ f | \exists (t \in T) \land (f = t \circ \circ r) \}
\]

### Definitions of behaviour specified by graphs:

1. An **attributed graph** is a graph \( G \) together with one or more (attribute)functions of type \( A_i \rightarrow \Sigma \), or type \( N_i \rightarrow \Sigma \), for some algebra \( \Sigma \)
2. The **data relation** \( \varepsilon(G) \) specified by a directed hypergraph \( G \) which is attributed by \( T: A_i \rightarrow \text{TABLES} \) is defined by:

\[
\varepsilon(G) = \bigotimes \varepsilon(a) \in A_i \]

3. The **order relation** \( \sigma(G) \) specified by a directed hypergraph \( G \) which is attributed by \( O: A_i \rightarrow \text{ORDERS} \) is defined by:

\[
\sigma(G) = \text{TCL}(\bigotimes \sigma(a) \in A_i)
\]

where TCL is the well known transitive closure for order relations

4. \( \text{TABLES} = \langle \varepsilon, \bigotimes, \bigcup, \bigcap \rangle \) and \( \text{ORDERS} = \langle \sigma, \bigcup, \bigcap, \text{TCL} \rangle \), are algebra’s, sets together with operations.
5. The **behaviour** of a directed hypergraph \( G \) which is attributed by \( T: A_i \rightarrow \text{TABLES} \) and \( O: A_i \rightarrow \text{ORDERS} \) is the tuple \( \beta(G) = \langle \varepsilon(G), \sigma(G) \rangle \) in which \( \varepsilon(G) \) is the data relation specified by \( G \) and \( \sigma(G) \) is the order relation specified by \( G \).
6. The **external behaviour** of a multi-ported directed hypergraph \( G \) which is attributed by \( T: A_i \rightarrow \text{TABLES} \) and \( O: A_i \rightarrow \text{ORDERS} \) and of which the set of external ports is \( P_e = P_{\text{in}} \cup P_{\text{out}} \) is the tuple \( \beta^e(G) = \langle \varepsilon(G), \sigma(G) \rangle \) in which \( \varepsilon(G) = \varepsilon(G) \upharpoonleft P_{\text{in}} \) is the **external data relation** specified by \( G \) and \( \sigma(G) = \sigma(G) \upharpoonleft P_{\text{in}} \) the **external order relation** specified by \( G \).
7. A **transformation** of graph \( G \) into graph \( H: G \Rightarrow H \) is said to be the application of transformation rule \( p: L \xleftarrow{-} K \xrightarrow{r} R \) iff \( p \) is a graphmorphism of \( K \rightarrow G \) can be given which satisfies the gluing condition. For given transformation rule \( p \) and graphmorphism \( g \) this transformation is referenced to by: \( G \Rightarrow g \)
8. A transformation \( G \Rightarrow H \) of graph \( G \) into graph \( H \) is **behaviour preserving** if the external behaviour of both graphs is equivalent: \( \beta^e(G) = \beta^e(H) \)
9. A transformation rule \( L \xleftarrow{-} K \xrightarrow{r} R \) is behaviour preserving if \( \beta(L) \xrightarrow{l} \beta(R) \xrightarrow{r} \)

---

**Figure 12:** Definitions

Based on table algebra and operations on order relations it can now be proven that:

**Theorem:** A transformation \( G \xrightarrow{\phi, p} H \) which is the application of a behaviour preserving transformation rule \( p \) is behaviour preserving
6. Conclusions

A formal framework for transformational design on graph representations is based on graph rewriting theory. An important advantage of this formal framework is the embedding of expression rewriting, term rewriting, on which several of the transformations, which have appeared to be useful, are based. From graph rewriting theory the gluing condition can be taken over and describes the applicability of transformation rules. These transformation rules have a central function because, according to graph rewriting theory, transformations are defined as applications of transformation rules. Interface graphs are useful in the definition of transformation rules and their behaviour preserving characteristic. Behaviour can be derived from a graph by the use of attribute functions which map hyperedges onto the data- and order-relation they specify. Tables are a useful representation of the data-relation and a denotational semantics of hypergraphs can be partly based on table algebra. The general graph concept and the theory of graph rewriting both used as basis for the here presented formal framework of transformational design unifies the transformational design approaches based on different graph representations.

References


[Jones90] G. Jones and M.Sheeran, Circuit design in Ruby, in J. Staunstrup(Eds), Formal Methods for VLSI design, North-Holland, pp13-70


Available at http://wwwspa.cs.utwente.nl/aid/trades/trades.html


