Mechanizing in Higher-Order Logic Proofs of Correctness and Completeness for a Set of RTL Transformations

Elena Teica and Ranga Vemuri

University of Cincinnati, Cincinnati, OH 45221-0030
Email: {eteica,ranga}@ececs.uc.edu

Abstract

This work presents a formal treatment of correctness and completeness for a set of seven uninterpreted Register Transfer Level (RTL) transformations. The completeness property ensures that a transformational derivation system based on this set is able to explore the entire design space, for a well-defined class of designs. The formalization for behavior specifications, RTL implementations and RTL transformations, as well as the mechanized proofs for correctness and completeness are conducted within the higher-order logic of the Prototype Verification System (PVS).

1 Introduction

In spite of the increased confidence one has in automatic synthesis when compared with manual design, the use of High Level Synthesis (HLS) tools does not considerably ease the burden of hardware verification. HLS tools employ a set of complex optimization algorithms which are usually implemented as large pieces of code, hence more error prone. Possible software errors in these implementations may lead to incorrect functionality of the synthesized designs.

When contemplating formal verification of *synthesized* designs, one can resort to a general verification approach, like model checking or theorem proving. As an alternative, many research efforts are focused on ensuring the correctness of the synthesis process itself, thus trying to eliminate the need of a difficult post-facto verification. A survey and classification of approaches to correct synthesis can be found in [1]. One of these approaches is *transformational derivation*; it refers to a class of synthesis techniques wherein an RTL design is derived by applying a sequence of behavior-preserving transformations to an initial design representation. A variety of transformational derivation systems have been proposed; DDD [2], T-Ruby [3], Veritas [4], TRADES [5] and HASH [6] to name a few.

The work presented here establishes the formal foundation on which a transformational derivation system using a core set of *uninterpreted RTL transformations* can be based. Starting from a minimal model for RTL designs we formally specified in in the Prototype Verification System (PVS)[7–9] a set of 7 RTL transformations. We mechanically proved that each transformation of our set is correct (it preserves the computational behavior of the design). The *completeness property* for a set of transformations

is defined as the capability to derive any possible implementation of a given behavior specification using a finite sequence of transformations contained in this set. In proving completeness we use a constructive approach. The practical uses of our work are two-fold:

1. A transformational derivation system based on the core set of RTL transformations presented here will yield (if correctly implemented) correct design implementations. The completeness property of this set ensures a virtually exhaustive search of the design space, for a certain class of designs.
2. Having identified a complete finite set of transformations, then it can be asserted that any correct application of a synthesis algorithm should be assimilable with a sequence of transformations of the complete set. One can check the correctness of an existing synthesis tool by attempting to identify such sequences.

Vemuri [10, 11] reported a completeness argument for a similar set of transformations. However, his argument was informal, written in English without using any formal logic or notation. Paper and pencil proofs usually use ad-hoc notions and need to be examined by others to validate them. In contrast, we present a proof which is fully mechanized in the higher-order logic of PVS and can be run by a simple command in the proof environment. PVS is a higher-order logic specification and proving environment which has an expressive language and a high degree of automation[7].

In Section 2 we present the formal model for RTL designs and behavior descriptions. Section 3 summarizes the set of transformations and the specification and proof methodology. A proof of completeness for this set is outlined in Section 4.

A collection of PVS specifications included in the Appendix illustrates the specification methodology. For the complete specifications and proofs we refer the reader to *http://www.ececs.uc.edu/∼eteica/pvs*.

2 Formal Models for RTL Designs and Behavior Descriptions

We present in this section abstract formal models for RTL designs corresponding to behavior specifications which consist of basic blocks of straight-line code with operations and operators assumed to be binary.

An RTL design consists of a data-path and a controller which defines the sequence of register transfers to be executed in the data-path. The data path of an RTL design consists of a set of operators, a set of registers and interconnections between them. Operators are hardware units that perform arithmetic operations and other data transformations. Registers are used only for storing values (no shift or increment operations are considered). In what follows we will use the term *components* when refering to operators and registers collectively. We denote by *E* the set of all components in the data path, *OP* the set of all operators in *E*, and *REG* - the set of all registers in *E*. With these notations, we call a *data path structure* a tuple of the form: *(E, OP, REG)*.

A register transfer *rt* of a data path structure *(E, OP, REG)* defines the interconnections between a subset of components from *E*, according to those computations scheduled to be performed in the data path at the control step defined by *rt*.

Definition 1. A **register transfer** *rt* associated with a data path structure *(E, OP, REG)* is a tuple of the form:

$$(EXPR, REG_{out}, f_{op} : OP \to (E \times E), f_{reg} : REG_{out} \to E)$$

where $EXPR \subseteq E$, and $REG_{out} \subseteq REG$. We call REG_{out} the set of output registers of *rt*. f_{op} and f_{reg} define the interconnections between components of the data path at the control step corresponding to *rt* as follows: f_{op} is a function mapping an operator to a pair of components (its sources), and f_{reg} is a function that maps an output register to a component (its input).

Although we started with a very simple model, throughout the specification exercise we carefully enforce well-formedness conditions in order to keep the specification consistent with the physical reality it was intended to model. For example, we will not allow a register transfer which contains combinational cycles or floating inputs for operators and registers. To formally define such requirements we will first introduce the definition of the *ancestors* set for a component *e* as the set of all components which are connected to *e* through a direct path.

Definition 2. The **ancestors** set *A* of a component *e* of a data path structure *(E,OP,REG)*, with respect to a mapping function f_{op} *: OP* $\to$ *(E* $\times$ *E)* is recursively defined as:

$$A(e) = \begin{cases} \emptyset & \text{e : register} \\ A(f_{op}(e)`1) \cup A(f_{op}(e)`2) \cup \{f_{op}(e)`1, f_{op}(e)`2\} & \text{e : operator} \end{cases}$$

where $f_{op}(e)`1$ and $f_{op}(e)`2$ represent the first, respectively second projection of the f_{op} function applied to the operator *e*.

Well-formed register transfers. A register transfer *rt* is said to be *well-formed* if: 1) each operator in the set *EXPR* of *rt* has its *ancestors* included in *EXPR* (there are no floating inputs for operators); 2) there are no combinational cycles within an *rt*; 3) each output register in REG_{out} has its source in *EXPR*; 4) concurrent operations are performed on different hardware resources. Using a more formal notation:

Definition 3. A **well-formed register transfer** of a data-path structure (E, OP, REG) is a register transfer $(EXPR, REG_{out}, f_{op}, f_{reg})$ for which the following properties stand:
1) $\forall(e \in EXPR) : ancestors(e) \subseteq EXPR$;
2) $\forall(e \in EXPR) : not(e \in ancestors(e)$;
3) $image(f_{reg}) \subseteq EXPR$;
4) $\forall(e1, e2 \in (EXPR \cap OP)) : (e1 = e2) \Rightarrow (f_{op}(e1)`1 = f_{op}(e2)`1 \wedge f_{op}(e1)`2 = f_{op}(e2)`2)$.

In the absence of conditional and loop constructs, the only control information that needs to be supplied in order to completely define the functionality of the design is the ordered sequence of interconnection configurations. The order is defined with respect to the timing sequencing of operations to be performed in the data path. We define a *control graph* associated with a data path structure *(E,OP,REG)* as a *list* of *well-formed* register transfers: $control_graph(E, OP, REG)$ = $\langle\, rt_1 \,\rangle$, $\langle\, rt_2 \,\rangle$ ... $\langle\, rt_n \,\rangle$, where rt_1, rt_2 .... rt_n are register transfers associated with the data path structure (E, OP, REG).

Definitions for Behavior Descriptions. Although restricted in comparison with the variety of behavior descriptions currently implemented by the synthesis tools, the straight-line code model is sufficient for describing many complex DSP applications. Also, many of the optimization algorithms used by the existing HLS tools operate only within the boundaries of loops and conditional constructs, that is, on the straight-line code portions of a behavior specification.

Informally, a straight-line code behavior description assigns to each output variable the result of a computation performed on a set of input variables. We call this computation a *behavior expression*, and define it recursively as:

```
<beh> ::= <variable> | (<beh> <operation> <beh>)
```

where `operation` is any binary arithmetic or logic function.

A trivial RTL implementation of a given behavior description can be obtained by assigning to each operation an operator instance which performs that operation. Conversely, each design implementation can be associated a behavior description obtained by traversing the operator trees in successive register transfers, and converting them into behavior expressions. The extracted behavior expression associated to a component of an implementation is defined bellow.

Definition 4. The **extracted behavior** of a component *e* in an RTL implementation represented as a control graph list *cg* is defined using two mutually recursive functions:

$$eb1(e, cg) = \begin{cases} eb2(e, cdr(cg)) & e : reg \\ (eb1(car(cg)`f_{op}(e)`1), opf(e), eb1(car(cg)`f_{op}(e)`2)) & e : op \end{cases}$$

$$eb2(r, cg) = \begin{cases} r & cg = null \\ eb1(car(cg)`f_{reg}(r), cg) & r \in car(cg)`REG_{out} \\ eb2(r, cdr(cg)) & r \notin car(cg)`REGout \end{cases}$$

where *opf(e)* denotes the operation performed by an operator *e*, *r* is a register, *car* and *cdr* are the LISP notations for functions used to create and manipulate lists. When refering to the computational behavior of an RTL implementation, we actually refer to the application of the *eb2* function to the output registers of this implementation.

In the PVS specification for the model described so far we associate with each RTL and behavior description component a type whose values best identify with the structures they represent:

- *registers*, *operators* and *operations* are uninterpreted nonempty types;
- functionality of an operator is defined by a function type from an *operator* to an *operation*;
- writes/assignments to output registers are function types;
- *register transfers* and *behavior descriptions* are record types;
- *well-formed register transfers* are of a predicate subtype of *register transfers*;
- *control graphs* are lists of *well-formed register transfers*.

The interconnections between operators, as well as between operators and input registers (corresponding to f_{op} in **Definition 1**) are implicitly defined using a binary-tree structure specified as a parameterized PVS recursive abstract DATATYPE [12] (see

Appendix 1). A similar approach is proposed in [13] to model tree-shaped combinational structures using recursive binary trees defined in HOL. PVS supports only total functions such that any recursive definition is required to terminate. As a corollary, no operator can be its own ancestor; in our model this translates to operator expressions with no combinational cycles, which suites well the second well-formedness condition in **Definition 3**.

A behavior expression has an associated parse tree that can also be modeled as a binary tree datatype in PVS. Unlike the operator expressions, the behavior expressions do not interpret the physical instance of an operator, but are concerned only with the functions performed by these operators. The use of PVS's DATATYPE mechanism for defining the operator trees and behavior expressions facilitated both specifications and proofs. Upon type-checking a datatype definition, PVS automatically generates basic declarations and axioms that formalize an abstract datatype, including extensionality axioms, induction schemes and recursion combinators.

In order to establish the correctness of a transformation as a behavior preserving correctness property, we need to check if two different RTL designs implement the same behavior description. For this we defined an *extracted behavior* function according to **Definition 5** (see Appendix 3) which extracts the computational behavior of an output register in a control graph. This function will be used in the next section to express the correctness theorems.

3 The set of RTL Transformations

It what follows we will succinctly describe each transformation in our set. For each transformation we specify its *inputs*,*function* and *preconditions* that ensure the behavior-preserving correctness. The *Correctness Theorem* is expressed as:

$$T_Preconditions(rtl_impl, set_of_comp) \Rightarrow$$
$$c_behavior(T(rtl_impl, set_of_comp)) = c_behavior(rtl_impl)$$

where *T_Preconditions* is a predicate function associated with a generic transformation *T*. It returns true if certain properties of a set of components (*set_of_comp*) of the design *rtl_impl* are satisfied. The function *c_behavior* extracts the behavior of the output registers (corresponding to the output variables in the behavior description).

For the conciseness and readability of the transformations' descriptions we introduce several notations :

1. $e \in rt$ if the component *e* appears in the *EXPR* field of the register transfer *rt*.
2. $operator?(e)$ is a predicate which defines the set of all *operator* instances.
3. $opfn(e)$ returns the function of operator *e*.
4. $in_regs(rt)$ denotes the set of input registers of a register transfer *rt*.
5. $operators(rt)$ define the set of all operators that appear in a register transfer *rt*.

The transformations discussed here are *uninterpreted* in the sense that we do not consider the semantics of operations within an interpretation domain, but only the structural aspects of the design representation to which the transformations are applied. For example, logic optimization transformations consider the functions implemented by gates (AND,OR,etc), hence they are *interpreted*. ALU folding is concerned only with the

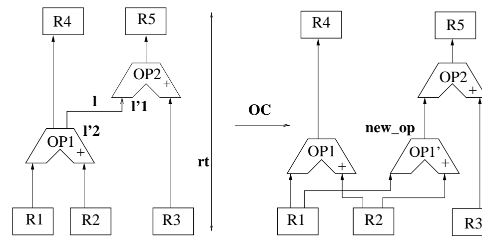

Fig. 1. The OC Transformation

components' interconnections and check that the folded ALUs have the same functionality (without interpreting this functionality), so they are *uninterpreted*.
In what follows we give an intuitive description for each transformation in our set.

The **Operator Copy (OC)** transformation eliminates sharing of operators as illustrated in Figure 1. In this figure, the result of *OP1* is used by both *OP2* and *R4*. OC inserts a new operator *OP1'* which performs the same function as *OP1*. Both *OP1* and *OP1'* have the same inputs but have only one target component each.
Inputs of OC: *1) a register transfer* **rt** *of a control graph* **cg** *to which OC is locally applied, 2) a link* **l** *defined as a tuple of components which are connected within* **rt** *such that the first projection is the source instance and the second projection is the driven instance, 3) a new operator* **new_op**.
Function of OC: *inserts* **new_op** *in* **rt** *as a copy of the second projection of* **l (l'2***), deletes* **l** *and creates a new connection.*

$$OC_Precondition(rt, l, new_op) \stackrel{\Delta}{=}$$
$$\exists (e \in rt) : (operator?(e) \bigwedge (l`2 = e) \bigwedge$$
$$(opfn(new_op) = opfn(l`2)) \bigwedge (not(new_op \in rt)))$$

Operator Instance Substitution (OIS) allows the reuse of an operator instance which is available during a certain register transfer. In Figure 2 operators *OP1* and *OP2* are used to perform addition in two different register transfers. OIS substitutes the appearance of *OP2* by *OP1*.
Inputs of OIS: *1) a register transfer* **rt** *of a control graph* **cg** *to which OIS is locally applied, 2) an operator* **subst** *to be substituted, 3) An operator* **opr** *that will substitute operator* **subst**.
Function of OIS: *Replace the appearance of* **subst** *in* **rt** *by* **opr**.

$$OIS_Precondition(rt, subst, opr) \stackrel{\Delta}{=}$$
$$(opfn(subst) = opfn(opr)) \bigwedge$$
$$(opr \in rt \Rightarrow (f_{op}(opr)'1 = f_{op}(subst)'1 \wedge f_{op}(opr)'2 = f_{op}(subst)'2))$$

Register Transfer Split (RTS) transforms a register transfer into a sequence of two transfers by saving intermediate results into a set of new temporary registers.
Inputs of RTS: *1) a register transfer* **rt** *of a control graph* **cg** *to which RTS is locally*

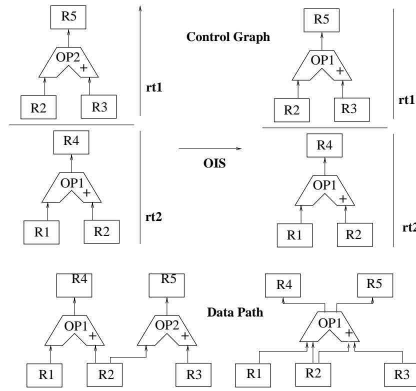

Fig. 2. The OIS Transformation

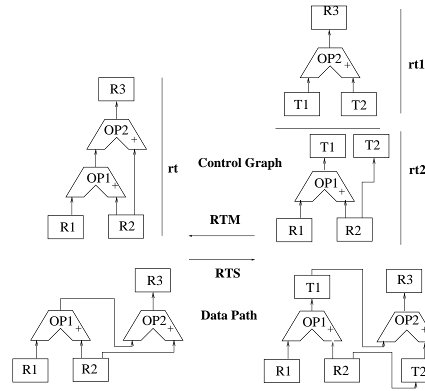

Fig. 3. The RTS and RTM Transformations

applied, 2) a set **split_set** *of operators and input registers to be scheduled in the first register transfer, 3) a set* **temp_set** *of temporary registers.*
Function of RTS: *RTS splits the initial transfer into two successive register transfers: the first register transfer contains the operators and input registers in* **split_set** *and saves intermediate values in temporary registers from* **temp_set***; the rest of the components appear in the second register transfer with updated interconnections, such that intermediate values saved in the temporary registers are correspondingly input to components.*
$RTS_Precondition(rt, split_set) \stackrel{\Delta}{=}$
$$\forall(e \in split_set) : operator?(e) \Rightarrow (f_{op}(e)'1 \in split_set \wedge f_{op}(e)'2 \in split_set)$$
In Figure 3, the RTS transformation is applied to the register transfer *rt*, **split_set** consists of operator *OP1* and input registers *R1* and *R2*, and the **temp_set** is formed of temporary registers *T1* and *T2*.

Register Transfer Merge (RTM) is the inverse of RTS: it merges to successive register transfers when the values computed in one transfer are passed to the next one through a set of intermediate registers (see Figure 3).
Inputs of RTM: *1. Two successive register transfers* **rt1** *and* **rt2** *of a control graph* **cg** *to which RTM is locally applied.*
Function of RTM:*it substitutes the sequence* $\rangle$**rt1,rt2**$\langle$ *by one register transfer obtained by eliminating the intermediate registers and replacing them with simple connections in an orderly manner.*
$RTM_Precondition(rt1, rt2) \stackrel{\Delta}{=}$
$$(in_regs(rt2) \subseteq rt1`REG_{out}) \bigwedge (operators(rt1) \cap operators(rt2) = \emptyset)$$

Register Transfer Decompose (RTD) can be viewed as sequencing two tasks which are initially executed in parallel. In Figure 4, OP2 is deferred to the next control step.
Inputs of RTD : *1) a register transfer* **rt** *of a control graph to which RTS is locally applied, 2) an output register* **R** *of* **rt**.
Function of RTD: *decompose* **rt** *in two successive register transfers such that in the*

second register transfer only the value to be read by **R** *is computed and all other operations are performed in the first register transfer.*
$RTD_Precondition(rt) \stackrel{\Delta}{=} in_regs(rt) \cap rt`REG_{out} = \emptyset$.
Figure 4 shows one register transfer (**rt**) and its decomposition in *rt1* and *rt2* corresponding to an RTD transformation where register **R** is *R5*.

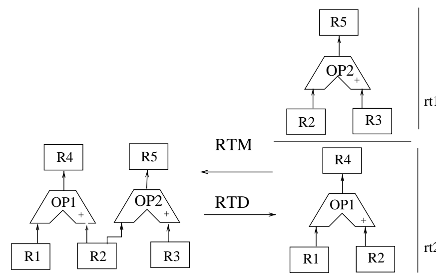

Fig. 4. The RTD and RTC Transformations

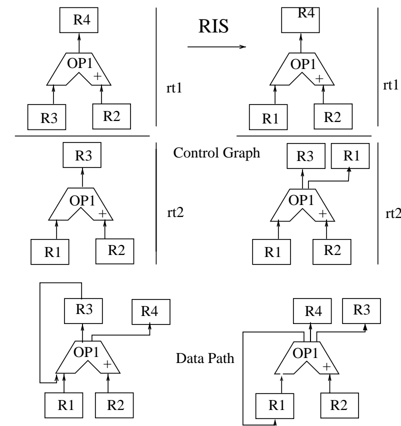

Fig. 5. The RIS Transformation

Register Transfer Decompose (RTC) is the inverse of RTD, it results in combing two successive register transfers which can be executed in parallel (in Figure 4, the operations performed in *rt1* and *rt2* can be executed in the same control step).
Inputs of RTC: *1) two successive register transfers* **rt1** *and* **rt2** *of a control graph to which RTC is locally applied.*
Function of RTC : *combines* **rt1** *and* **rt2** *if their respective computations can performed in parallel.*
$RTC_Precondition(rt1, rt2) \stackrel{\Delta}{=}$
$(rt1`REG_{out} \cap rt2`REG_{out} = \emptyset) \bigwedge (rt1`REG_{out} \cap in_regs(rt2) = \emptyset)$
$\bigwedge (operators(rt1) \cap operators(rt2) = \emptyset)$.

Register Instance Substitution (RIS) is illustrated in Figure 5. The appearance of *R3* as an input register in the register transfer *rt1* is substituted by *R1*. To preserve the behavior of the design, the substituting register *R1* is also added as an output register in that register transfer where *R3* was last written (*rt2*). The connections are updated such that *R1* will read the same value as *R3*.
Inputs of RIS: *1) a register transfer* **rt** *of a control graph to which RIS is locally applied, 2) an input register* **Subst** *of* **rt**, *3) a register* **Wth** *that will substitute* **Subst**.
Function performed by RIS: *The appearance of* **Subst** *as an input register in* **rt** *is substituted by the* **Wth** *register;* **Wth** *is also added as an output register where* **Subst** *was last written. The value written in* **Wth** *in this transfer is the same as the value*

written is **Subst**.

$RIS_Precondition(cg, rt, Subst, Wth) \stackrel{\Delta}{=}$
$\forall(rt \in cg) : not_alive?(Subst, rt) \vee not_alive?(Wth, rt).$

Let $<$ be an order on register transfers of a control graph **cg** such that for any two register transfers **rt1** and **rt2** of **cg**, **rt1** $<$ **rt2** if and only if **rt1** occurs after **rt2** in **cg**, and **rt1** $\leq$ **rt2** if and only if **rt1** occurs after **rt2** or is equal to **rt2**. Then we define register **R** not to be alive in the register transfer **rt** as follows:

$$not_alive?(R, rt) \stackrel{\Delta}{=} \forall(rt1 \leq rt) : (R \in in_regs(rt1)) \Rightarrow (\exists rt2 : rt2 \leq rt \wedge rt1 < rt2 \wedge R \in rt2`REG_{out}) \bigwedge \forall(rt1) : rt < rt1 \wedge (R \in rt1`REG_{out}) \Rightarrow (\exists(rt2 < rt1) : rt < rt2 \wedge R \in in_regs(rt2))$$

In the above definition, a register *R* is said to be *not_alive?* in a register transfer *rt* if: a) every time *R* is read after the control step corresponding to *rt*, than the read value was also written after *rt*, b) every value written to *R* before *rt* is also read before *rt*. In other words, *R* is not alive during *rt* if it does not need to preserve its value during the control step corresponding to *rt*.

In specifying the transformations we used a definitional approach, such that only a couple of axioms were introduced in the specification. We followed a specification methodology which resulted in a uniform definition of transformations; the uniform definition allowed a similar treatment of the well-formedness and correctness proofs for all transformations. As an example, the Appendix 4 shows a complete specification for OIS: a recursive function updates the operators' interconnections according to the transformation applied; next, each field of the affected register transfer is updated to accommodate the transformation, and finally - bring the locally defined transformation in the context of the entire control graph.

A generic *transformation* was defined as a function which assigns control graphs to control graphs (i.e., only the well-formedness property is considered). A *correct transformation* was defined as a predicate subtype of *transformation* induced by the behavior preserving condition (see Appendix 5). A set of transformations for which we will prove completeness was defined as a predicate subtype of *correct transformations* that can be expressed using one of the seven transformations described above. Along the specification exercise we have not explicitly stated the well-formedness or the correctness invariants, nor have we included sublemmas which assert that a certain transformation belongs to the complete set. Instead, the invariants were implicitly enforced by the predicate subtype definitions, and were automatically expressed by the PVS's type-checker as proof obligations. This allowed a much shorter and readable specification for the witness sequences of transformations used to constructively prove completeness, as it will be described in the next section.

4 Proving Completeness for a Set of Transformations

A set of correct (behavior-preserving) RTL transformations is said to be *complete* if for any two RTL designs which implement the same behavioral description, one design can be derived from the other by applying a finite sequence of transformations contained in this set. In what follows we will call transformations belonging to our set *operational*

transformations (OT), and RTL designs implementing the same behavior - *equivalent designs*.

Definition 5. Let *s* be a set of RTL transformations. *s* is *complete* iff:

$$\forall(rtl1, rtl2 : designs, r : output_register) :$$
$$(c_behavior(r, rtl1) = c_behavior(r, rtl2) \Rightarrow$$
$$\exists(seq : (s)) : apply(seq)(rtl1) = rtl2)$$

where: *c_behavior* is a function that extracts the computational behavior of an output register *r* in a *design* implementation, *seq* is a sequence of transformations in *s*, and *apply* is a function which recursively applies transformations in the order defined by *seq*.

Figure 6 illustrates the approach we used in proving completeness. We formulated the following three subgoals which together imply completeness:

1. For every RTL design *rtl1* there exists a sequence of OTs which transforms it into an equivalent design consisting of a single register transfer (all operations are performed in the same control step) *rtls*.
2. For any two equivalent implementations consisting of only one register transfer there exists a sequence of OTs which derives one design from the other.
3. For any RTL design *rtl2* there exists an equivalent design consisting of only one register transfer from which *rtl2* can be derived through a sequence of OTs.

The first two subgoals were constructively proved by instantiating the sequent formulas with algorithmically created witness sequences of OTs. A proof for the Completeness Theorem was derived from the above subgoals using appropriate instantiation.

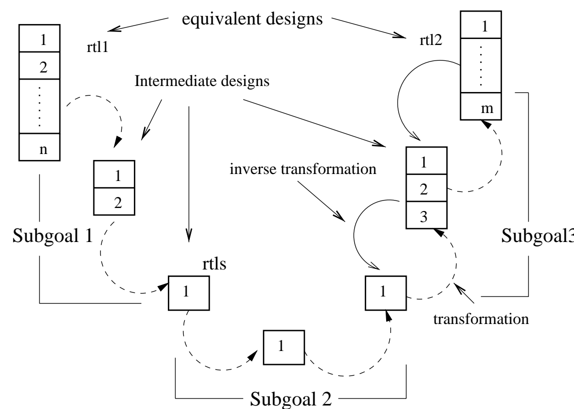

Fig. 6. Approach for the Completeness Proof

A Constructive Proof for the First Subgoal. The first subgoal states that any control graph associated with an RTL design can be transformed into a control graph of length

1 after repeatedly applying a sequence of OTs:

$$\forall(rtl : c_graph) : (\exists(s : ot_seq) : length(apply(s)(rtl)) = 1$$

The only OTs which result in decreasing the control graph length are RTM and RTC. But the simple merging or composing is not always possible since a set of preconditions has to be satisfied. However, it is possible to define sequences of OTs which create a scenario where preconditions for merging or composing are satisfied.

Definition 6. We define *preconditioning for a transformation* `t` the application of a sequence of OTs which results in an equivalent control graph having the same length with the initial one, but where the preconditions for `t` are satisfied.

In order to create the witness for instantiating the sequent of the first subgoal, we defined such sequences of preconditionings followed by the preconditioned transformations that result in repeatedly combining the first two register transfers of a control graph, until the length of the list becomes equal to 1. The algorithm which defines this witness sequence consists of the following steps (see also Figure 7):

1. A **Preconditioning for RTD** step is applied to the second transfer in the control graph in order to create valid preconditions for decomposition. This step proves that a sequence of RIS transformations applied to input registers of the second transfer, followed by RTC transformations, results in disjoint input and output register sets for this transfer (which is the precondition for RTD), while preserving the length of the graph. (see Figure 7.b).
2. **Reduce the cardinality of the output registers set of the second transfer** by deferring the computation for one output to the subsequent transfer. This is done using a sequence of: (a) decomposition (RTD) for one output (*Ri* in Figure 7.c), (b) preconditioning for merging or composing the newly created transfer with the first register transfer, (c) merge or compose the new transfer *(2a)*, with the first transfer (see Figure 7.d)
3. **Reduce the length of the control graph** by recursively applying the previous step until all outputs of the second transfer are combined (through RTM or RTC) with the first transfer. After executing this step, the second transfer becomes *empty* so it can be deleted from the control graph (see Figure 7.e).
4. **Reduce the length of the control graph to 2** by recursively applying the third step.
5. **Reduce the length of the graph from 2 to 1**. This is treated as a separate step because it needs a sequence of transformations different from the one executed in step 3, but using the same OTs.

Proof Strategy for the Second Subgoal. The second subgoal states that any two equivalent RTL designs consisting of only one register transfer, can be derived one from the other through a sequence of OTs:

$$\begin{aligned}&\forall(rtl1, rtl2 : c_graph) :\\&\quad(length(rtl1) = length(rtl2) = 1 \wedge c_behavior(rtl1) = c_behavior(rtl2))\\&\qquad\Rightarrow (\exists(s : ot_seq) : rtl2 = apply(s)(rtl1))\end{aligned}$$

What distinguishes equivalent designs consisting of one register transfer (the associated control graph has length 1) is only a possibly different sharing of operators and a different assignment of operators to physical instances.

The approach of [10, 11] in proving completeness makes use of a Normal Form Structure definition, which is a unique implementation of a given behavior description. The

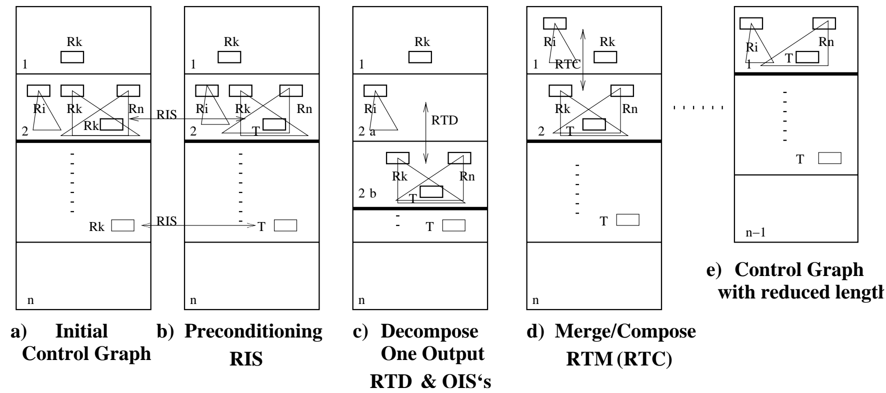

Fig. 7. Reducing the Length of a Control Graph

Normal Form Structure is in the *Maximum Operators Form*, that is all operator sharings are eliminated (see Figure 8.a). There is an important annotation to this definition: the Normal Form is unique *"within naming of the hardware components"*. When specifying in a strict formal system, such a normal form is not a unique design, but a class of designs which implement the same behavior in a *particular form* that defines a set of *isomorphic structures*. The particular form of implementation used in our proofs is a *Minimum Operators Form*:

Definition 7. *A register transfer* `rt` *is said to be in the* `Minimum Operators Form (MinOF)` *if for any two expressions* `e1` *and* `e2` *in* `rt`*, if* `e1` *and* `e2` *have the same functionality and the same right and respectively left sources, then* `e1 = e2`.

In other words, the sharing of operators in `rt` is maximized, and the number of operator instances is minimized (see Figure 8.b). The use of MinOF as the particular form of implementation is motivated by the fact that is easier to define a bijective mapping between elements of two equivalent register transfers which are in the *MinOF* form. This mapping was used in defining the isomorphism between two implementations.

We conjecture that: *Two register transfers having isomorphic structures can be transformed one into the other by a finite sequence of OIS transformations.* This conjecture implies the *uniqueness within naming of the hardware components*. Indeed, the correct application of OIS when the *substituting* operator instance is not already in use in the register transfer has the effect of "renaming" the *substituted* operator.

A proof for the second Subgoal was then derived from the following three subgoals:

1. Any design consisting of only one register transfer can be brought to the *MinOF* form by applying a finite sequence of OIS transformations.
2. Any two equivalent designs which implement the same behavior in the *MinOF* form are structurally isomorphic.
3. The OIS transformation admits as inverse a sequence of OTs.

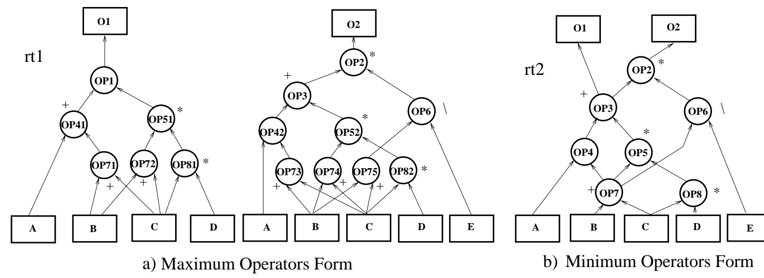

Fig. 8. Equivalent Transfers in Particular Forms of Implementation

Proof Strategy for the Third Subgoal. The third subgoal states that for any RTL design *cg* there exists an equivalent design consisting of only one register transfer from which *cg* can be derived through a sequence of OTs:

$\forall(rtl : c_graph) : (\exists(rs : c_graph, s : ot_seq) :$
$length(rs) = 1 \wedge apply(s)(rs) = rtl)$ A proof for this subgoal was derived from the first subgoal (constructively proved as described above), and a sublemma stating that *each OT admits as inverse a sequence of OTs*. The mechanization of this sublemma is part of the remaining work. Informally, each transformation admits as inverse a sequence of transformations as follows:

- The inverse for OIS is a sequence of OCs (which are defined for only one link at a time), or another OIS. The inverse of OC is OIS.
- The inverse for RTS is RTM and vice-versa.
- The inverse for RTD is RTC. Since RTD is defined only for one output of a register transfer, the inverse of RTC is a sequence of RTD and RTC transformations.
- The inverse for RIS is also an RIS, or (when the substituted register is a primary output) a sequence of RIS and RTM.

5 Conclusions and Future Work

There are several limitations of this work that we will address in the future:

- We considered only implementations for behavior descriptions consisting of straight-line code blocks. The model and the set of transformations can be enhanced to deal with conditional constructs such that the completeness property will still hold.
- We do not guarantee that the preconditions used in proving correctness of the core set of transformations are *weakest* preconditions. This would affect the effectiveness of a verification methodology based on the completeness of our set.
- In the formal models presented here we implicitly considered that all variables are correctly mapped to operators and registers input and output ports.
- The RTL models and transformations presented here are *uninterpreted* , that is, we do not interpret the function performed by an operator in a domain of values. The

correctness of a transformation is thus defined with respect to a relatively strict structural similarity between two RTL designs.

References

1. R. Kumar. Formal Synthesis in Circuit Design - A Classification and Survey. In *Formal Methods in Computer-Aided Design*, pages 294–309, 1996.
2. S. Johnson and B. Bose. DDD: A system for mechanized digital design derivation. In *Proceeding International Workshop Formal Methods in VLSI Design*, 1991.
3. R. Sharp and O. Rasmussen. The T-Ruby design system. In *Computer Hardware Description Languages and their Applications*, pages 587–596, 1995.
4. F. Hanna, M. Longley, and N. Daeche. Formal Synthesis of Digital Systems. In L. Claesen, editor, *Proceedings IMEC-IFIP International Workshop on Applied Formal Methods in Correct VLSI Design*, pages 532–548, 1989.
5. P. Middelhoek and S. Rajan. From VHDL to Efficient and First-Time Right Designs: A Formal Approach. *ACM Transactions on Design Automation of Electronic Systems*, 1(2):205–250, April 1996.
6. D. Eisenbiegler, C. Blumenrohr, and R. Kumar. Implementation Issues About the Embedding of Existing High Level Synthesis Algorithms in HOL. In *International Conference on Theorem Proving in Higher Order Logics*, 1996.
7. S. Owre, J. M. Rushby, and N. Shankar. PVS: A prototype verification system. In Deepak Kapur, editor, *11th International Conference on Automated Deduction (CADE)*, volume 607 of *Lecture Notes in Artificial Intelligence*, pages 748–752, Saratoga, NY, June 1992. Springer-Verlag.
8. S. Owre, N. Shankar, J. M. Rushby, and D. W. J. Stringer-Calvert. *PVS Language Reference*. Computer Science Laboratory, SRI International, Menlo Park, CA, September 1999.
9. N. Shankar, S. Owre, J. M. Rushby, and D. W. J. Stringer-Calvert. *PVS Prover Guide*. Computer Science Laboratory, SRI International, Menlo Park, CA, September 1999.
10. R. Vemuri. *A Transformational Approach to Register-Transfer Level Design Space Exploration*. PhD thesis, Case Western Reserve University, 1989.
11. R. Vemuri. How to Prove the Completeness of Register Level Design Transformations. In *Proceedings of the 27th ACM/IEEE Design Automation Conference*, pages 207–212, June 1990.
12. Sam Owre and Natarajan Shankar. Abstract Datatypes in PVS. Technical Report SRI-CSL-93-9R, Computer Science Laboratory, SRI International, Menlo Park, CA, December 1993. Extensively revised June 1997; Also available as NASA Contractor Report CR-97-206264.
13. T. Melham. *Higher Order Logic and Hardware Verification*. Cambridge University Press, 1993.

6 Appendix

1. Definitions for Operator Trees and Register Transfers:

```
expression[R: TYPE, O: TYPE] : DATATYPE
BEGIN
reg(reg: R): reg?
op(op: O, source1:expression, source2:expression): op?
END expression
```

```
rt: THEORY
BEGIN
register: TYPE+
operator: TYPE+
operation: TYPE+
opfn: [operator -> operation]
IMPORTING expression_adt[register, operator]
transfer : TYPE = [# exp_set:finite_set[expression],
                     outregs:finite_set[(reg?)],
                     regassign:[(reg?)->expression] #]
END rt
```

2. Definitions for Well-formed Register Transfers and Control Graphs:

```
wellformed_set?(e_set:finite_set[expression]):bool=
 (forall(e:(e_set)):subset?(ancestors(e), e_set)) AND
 (forall (e1,e2: (e_set)) :
   (op?(e1) AND op?(e2) AND op(e1)=op(e2)) => e1=e2)

wellformed_out?(rt:transfer) : bool =
 subset?(image(rt`regassign)(rt`outregs), rt`exp_set)

wellformed_rt?(rt:transfer): bool =
     wellformed_set?(rt`exp_set) and wellformed_out?(rt)
wellformed_rt:TYPE = {rt:transfer | wellformed_rt?(rt)}
c_graph: TYPE  = list[wellformed_rt]
```

3. Definitions for Extracted Behavior:

```
 extracted_behavior(cg: cons_graph, e:(car(cg)`exp_set)) :
  RECURSIVE beh =
   if op?(e) then exp(opfn(op(e)),
                      extracted_behavior(cg, source1(e)),
                      extracted_behavior(cg, source2(e)))
    else let m_cg: c_graph = match(e,cdr(cg)) in
     if null?(m_cg) then leaf(e)
        else extracted_behavior(m_cg, car(m_cg)`regassign(e))
     endif
   endif
 MEASURE size(e) + size(cdr(cg))
```

4. Definition for the OIS Transformations:

```
new_exp(e:expression, subst: (op?), opr: operator) :
  RECURSIVE expression=
    if reg?(e) then e
     else if e = subst then op(opr, source1(e), source2(e))
       else op(op(e), new_exp(source1(e), subst, opr),
                      new_exp(source2(e), subst, opr))
      endif endif
 MEASURE size(e)
```

```
ois(rt: transfer, subst: (op?), opr:operator) : transfer =
 (# exp_set:= new_exp_set(rt`exp_set, subst, opr),
    outregs:= rt`outregs,
    regassign:=
      LAMBDA (x:(reg?)):new_exp(rt`regassign(x),subst,opr) #)

OIS(cg:list[transfer],n:below[length(cg)],subst:(op?),
    opr:operator) :
  list[transfer] = replace(cg,n,ois(nth(cg,n),subst,opr))

precondition_ois?(rt:transfer,subst:(op?),opr:operator):bool=
   opfn(opr) = opfn(op(subst)) AND
   (operators(rt)(opr) =>
     (forall(e:(filter_op(rt`exp_set))) :
              op(e)=opr => (source1(e)=source1(subst) and
                            source2(e)=source2(subst))) )
OIS_PB: LEMMA
 forall(cg:c_graph,n:below[(length(cg))],subst:(op?),opr:operator,
        r:(output_registers(cg)) :
  precondition_ois?(nth(cg,n),subst,opr) =>
   extracted_behavior(r,cg) =
     extracted_behavior(r,OIS(cg,n,subst,opr))
```

5. Definitions for Generic Transformations, Correct Transformations, Operational Transformations, and the Completeness Theorem:

```
transform:TYPE = [c_graph->c_graph]
correct_tr:TYPE = {t:transform |
                    forall(cg:c_graph,r:(output_registers(cg))):
                       extracted_behavior(reg(r),cg) =
                        extracted_behavior(reg(r),t(cg))}
oper_tr : TYPE =
    {ot:correct_tr | forall(cg: c_graph) :
       (ot(cg) = iden_tr(cg) % identity transformation
                  OR
       (exists (n:below[length(cg)],r:(nth(cg,n)`outregs)) :
            ot(cg) = RTD(cg,n,r))
                  OR
       (cons?(cg) => exists (n:below[length(cg)-1]) :
            ot(cg) = RTC(cg,n))
                  OR
       (exists (n:below[length(cg)],ss:finite_set[expression]) :
            ot(cg) = RTS(cg,n,ss))
               ............

 COMPLETENESS :THEOREM
 forall(cg1,cg2 :cons_graph) :
  cg_behavior(cg1) = cg_behavior(cg2) =>
   exists(s:oper_tr_sequence): apply_sequence(s)(cg1) = cg2
```