UNIVERSITY OF CINCINNATI

September 5, 2001

I, Elena Teica

hereby submit this as part of the requirements for the degree of:

Ph.D

in computer engineering

It is entitled Formal Correctness and Completeness for a Set of Uninterpreted RTL Transformations

Approved by:

Dr. Ranga Vemuri

Dr. Hal Carter

Dr. Karen Davis

Dr. John Schlipf

Dr. Stephan Pelikan
Formal Correctness and Completeness for a Set of Uninterpreted RTL Transformations

A dissertation submitted to the
Division of Research and Advanced Studies
of the University of Cincinnati
in partial fulfillment of the
requirements for the degree of

DOCTOR OF PHILOSOPHY

in the Department of Electrical & Computer Engineering and
Computer Science of the College of Engineering

September 18, 2001

by

Elena Teica
eteica@ececs.uc.edu

M.S. (Electronics and Telecommunications),
University Politehnica, Bucharest, Romania, 1992

Thesis Advisor and Committee Chair: Dr. Ranga Vemuri
Abstract

The work presented in this thesis is concerned with the correctness of the high-level synthesis process. In particular, it addresses transformational derivation (TD) systems. TD denominates a class of synthesis techniques wherein a register transfer level (RTL) implementation is derived by applying a sequence of behavior-preserving transformations to an initial behavior representation.

We present in this thesis a formal treatment of correctness and completeness for a set of ten uninterpreted register transfer level transformations on which a TD system can be based.

The formal definitions for RTL transformations are based on an uninterpreted model for RTL designs corresponding to straight-line code behavior descriptions. Our model specifies an abstract RTL design as a set of properties asserted about abstract collections (sets) of components (operators and registers). RTL transformations are functions operating on elements of the set defined by these properties, and the completeness property is a higher-order predicate that defines a complete set of functions.

The specification and proof exercise are conducted in the Prototype Verification System (PVS). The use of PVS for the formal treatment presented in this thesis is justified not only by the typed higher-order logic that underlies its specification and proving environment (required by the very nature of the problem we are dealing with), but also by its rich language constructs that allow expressive and efficient specifications. As an alternative to an error prone axiomatic approach, each transformation is specified in a definitional manner. The correctness of each transformation is defined as the behavior preserving property: a transformation is correct if the extracted computational behavior of the output registers is the same in the initial and in the transformed design. Each transformation is mechanically proved correct in PVS based on the assumption that a set of preconditions is satisfied.

A finite set of correct RTL transformations is said to be complete if and only if for any two RTL designs that implement the same behavior description, one design can be derived from the other by applying a finite sequence of transformations contained in this set. In other words, any possible implementation of a given behavior can be synthesized by applying only certain transformations to an initial implementation. We formalized and mechanically proved in PVS the completeness property using a constructive approach. The proof is constructive in the sense that a finite sequence of transformations (belonging to the complete set), that transforms a certain design implementation into another one having the same behavior, is algorithmically defined.

Some of the practical uses of the theoretical results formally established are:

1. A transformational derivation synthesis system based on the core set of RTL transformations presented here will yield correct-by-construction designs (if correctly implemented). The completeness property of this set ensures a virtually exhaustive search of the design space.
2. Any high-level synthesis algorithm that performs only structural optimizations can be viewed as a sequential application of RTL transformations. 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 the synthesis process by attempting to identify such sequences.
Acknowledgments:

To my thesis advisor, Dr. Ranga Vemuri, for the constant support and guidance, for letting me be one of the many young researchers he is fostering in the excellent working environment the DDEL lab is.

To Dr. Carter, Dr. Davis, Dr. Schlipf and Dr. Pelikan for participating in my thesis committee, for reading drafts of this work and for the valuable comments that significantly improved the contents and presentation of this dissertation.

To my many friends and colleagues in the DDEL lab. Nazanine, thank you so much for the warm friendship and kind advise that helped me so much in the harder moments. Rajesh, thank you for being in the same boat with me and putting up with me like a good friend would. Many thanks to Meenakshi, Adrian, Sree, Karam and Iyad for being such good friends and company. Special thanks to Mrak, Viresh, Naren, Alex and Kat for being good friends and support while at DDEL and after they left. Thank you all for the nice atmosphere you created in our lab: Sriram, Jeff, Shankar, Preetham, Priya, Avartika, Vinoo, Satish, Amit, Srini, Jawad, Vijay, Shiva, Madhu, Manish, Vina, Anuradha, Hemanth, Mukesh, Raoul, Srinivas, Sunder.

To all my Romanian friends in Cincinnati for being my second family here. Special thanks to Ioana and George who guided my first steps in the United States, to Elena and Maria for being some of the best friends one could find, to Radu, Catalina & Daniel, Alex & Simona, Adrian & Monica, Romanita & Dragos, Luminita, Pompi, Dana, Sofia.

To all my friends at home for the best memories one can keep.

To my teachers at University “Politehnica” in Bucharest for doing such a good job in training so many generations of engineers.

To my parents Paula and Tianu, my sister Diana and my grandfather Nicu Demetrescu for their love. They are the reason for my work. I dedicate it to them.
# Contents

1 Introduction

1.1 Background and Related Work ................................................. 3

1.1.1 Approaches to Correct Synthesis ...................................... 4

1.1.2 Transformational Derivation (TD) Systems .............................. 6

1.2 Motivation ............................................................................. 9

1.3 Overview of the Research ..................................................... 12

1.4 Organization ........................................................................ 13

2 Formal Models for Behavior Descriptions and RTL Designs 14

2.1 Formal Definitions for an Abstract Model for RTL Designs .......... 15

2.2 Definitions for Behavior Descriptions ..................................... 18

2.3 PVS Specifications for the Abstract Model ................................. 19

2.3.1 PVS Specifications for Register Transfers ............................. 20

2.3.2 PVS Specification for Control Graphs ................................. 23

2.3.3 PVS Specifications for Behavior Descriptions ...................... 24

2.3.4 PVS Definitions for Extracted Behaviors ............................. 24

2.4 Summary .............................................................................. 28

3 Informal Description of RTL Transformations 29

3.1 Operator Copy (OC) .............................................................. 31
3.2 Operator Instance Substitution (OIS) ........................................ 32
3.3 Register Transfer Split (RTS) .................................................. 33
3.4 Register Transfer Merge (RTM) .............................................. 34
3.5 Register Transfer Decompose (RTD) ......................................... 35
3.6 Register Transfer Compose (RTC) ........................................... 36
3.7 Register Instance Substitution (RIS) .......................................... 37
3.8 Clean Outputs (CO) ............................................................. 39
3.9 Add Outputs (AO) ............................................................... 40
3.10 Delete Register Transfer (DRT) ............................................. 42
3.11 Summary ................................................................. 43

4 Formal Specifications in PVS for RTL Transformations 44
4.1 Specification Methodology ....................................................... 44
4.2 Operator Copy (OC) ............................................................ 49
4.3 Operator Instance Substitution (OIS) ....................................... 53
4.4 Register Transfer Split (RTS) ............................................... 54
4.5 Register Transfer Merge (RTM) .............................................. 58
4.6 Register Transfer Decompose (RTD) ....................................... 60
4.7 Register Transfer Compose (RTC) .......................................... 61
4.8 Register Instance Substitution (RIS) ....................................... 62
4.9 Clean Outputs (CO) and Add Outputs (AO) ................................ 65
4.10 Delete Register Transfer (DRT) ........................................... 67
4.11 PVS Specifications for Generic Transformations ......................... 68
4.12 Summary ................................................................. 70

5 A Mechanized Proof of Completeness for the Set of RTL Transformations 71
5.1 Formal Definition and PVS Specification of Completeness ............ 72
C Proofs for the Wellformedness and Behavior Preserving Correctness 144
  C.1 Example for the OC Transformation . . . . . . . . . . . . . . . . . . . . . . . . . . . 144
  C.2 Proof for the Wellformedness Property . . . . . . . . . . . . . . . . . . . . . . . . . . . 144
  C.3 Proof for the Correctness Property . . . . . . . . . . . . . . . . . . . . . . . . . . . . 146
    C.3.1 Proof for the Equivalence of Control Graphs Before and After the Transfor-
    mation - The Correctness Theorem . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 147
  C.4 Proof for Correctness for the RIS Transformation . . . . . . . . . . . . . . . . . . . . 148
  C.5 Proof for Correctness for the CO Transformation . . . . . . . . . . . . . . . . . . . . 151

D Proof Script for the Completeness Theorem 154
  D.1 Preconditioning for RTD - Proof Strategy . . . . . . . . . . . . . . . . . . . . . . . . . . . 155
  D.2 Proof Strategy for Discharging Type-Checking Conditions (TCCs) . . . . . . . . . . . 158

E Proof Strategy for the \texttt{to\_minform\_sequence} Lemma 161

F Strategy for Proving the Existence of an Inverse Sequence of OIS 165

G Formalization and Proofs of RTS Completeness for Scheduling 170

Bibliography 174
# List of Figures

1.1 Example of a Design Flow and Points of Verification ........................................ 2
1.2 Classification of Elementary Transformations ..................................................... 7
1.3 Witness Generator Method Approach ................................................................. 11

2.1 Example of Controller-Datapath Implementation .................................................. 16
2.2 The Fields of a Register Transfer ........................................................................... 21
2.3 Extracted Behavior of an Expression in a Control Graph ....................................... 25

3.1 The OC Transformation ......................................................................................... 31
3.2 The OIS Transformation ....................................................................................... 32
3.3 The RTS and RTM Transformations ....................................................................... 33
3.4 The RTD and RTC Transformations ....................................................................... 35
3.5 The RIS Transformation ....................................................................................... 37
3.6 A Sequence of RIS Transformations ...................................................................... 38
3.7 The CO Transformation ....................................................................................... 40
3.8 The AO Transformation ....................................................................................... 41
3.9 The DRT Transformation ..................................................................................... 42

4.1 The Effect of a Generic Local Transformation on a Control Graph ....................... 46
4.2 Extracted Behavior ............................................................................................... 47
4.3 The Effect of the OC Transformation on the Expression Trees ............................. 50
4.4 The Effect of the OIS Transformation on the Expression Trees . . . . . . . . . . . . 53
4.5 The Effect of the RTS Transformation on the Expression Trees . . . . . . . . . . . . 56
4.6 The Effect of the RTM Transformation on the Expression Trees . . . . . . . . . . . . 59
4.7 The Effect of the RTD Transformation on the Expression Trees . . . . . . . . . . . . 61
4.8 Definition of RTC . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 62
4.9 The Effect of the RIS Transformation . . . . . . . . . . . . . . . . . . . . . . . . . . 63
4.10 The Effect of CO and AO Transformations . . . . . . . . . . . . . . . . . . . . . . . 66
4.11 Definition for Operational Transformations . . . . . . . . . . . . . . . . . . . . . . . 69

5.1 Combining Two Register Transfers . . . . . . . . . . . . . . . . . . . . . . . . . . . . 72
5.2 Approach for the Completeness Proof . . . . . . . . . . . . . . . . . . . . . . . . . . 74
5.3 Reducing the Length of a Control Graph . . . . . . . . . . . . . . . . . . . . . . . . . 77
5.4 The Initial Design . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 78
5.5 The Preconditioning for RTD (Step 1) . . . . . . . . . . . . . . . . . . . . . . . . . . 78
5.6 The Decomposition for Output Register B (Step 2) . . . . . . . . . . . . . . . . . . . 79
5.7 The Preconditioning for RTM (Step 2) . . . . . . . . . . . . . . . . . . . . . . . . . . 79
5.8 Merging the First Two Transfers (Step 2) . . . . . . . . . . . . . . . . . . . . . . . . . 80
5.9 Decomposing for Output Register O2 (Iteration of Step 3) . . . . . . . . . . . . . . . 80
5.10 Preconditioning for RTC (Iteration of Step 3) . . . . . . . . . . . . . . . . . . . . . . 81
5.11 Composing the First Two Transfers (Iteration of Step 2) . . . . . . . . . . . . . . . . 81
5.12 Reduce the Length of the Control Graph (Ending an Iteration of Step 3) . . . . . . 82
5.13 Reduce the Length of the Graph From 2 to 1 (Step 4) . . . . . . . . . . . . . . . . . 82
5.14 The Application of the 2to1 function . . . . . . . . . . . . . . . . . . . . . . . . . . . 87
5.15 Two Equivalent Register Transfers . . . . . . . . . . . . . . . . . . . . . . . . . . . . 92
5.16 Equivalent Transfers in Particular Forms of Implementation . . . . . . . . . . . . . 93
5.17 Two Isomorphic Register Transfers . . . . . . . . . . . . . . . . . . . . . . . . . . . . 96
Chapter 1

Introduction

With the enormous complexity of ASIC and systems applications, design engineers confront several critical issues. First, the faster time to market requirements versus the increasing complexity of the designs requires an efficient exploration of architectural trade-offs, and design reuse. Second, while the advances in semiconductor technology allow the implementation of many features into a very complex design, the verification task has to ensure that the functional correctness is not compromised. Design verification is notoriously expensive, consuming as much as 70% of the design team’s manpower, schedule and budget [1].

Integration of High Level Synthesis (HLS) into the design flow is favorable with respect to these requirements. HLS tools generate register transfer level (RTL) architectures from behavior descriptions by assigning functions in the behavior description to operators, variables to registers, data dependencies to interconnection configurations and operations to control steps. In addition, HLS tools use a collection of optimization algorithms that automatically explore the design space and present the user with a set of RTL architectures that implement the desired functionality and satisfy a set of performance constraints, such as area, latency, throughput and power dissipation. The input into an HLS system is a behavior description, which is more intuitive than a structural description, hence more suitable for design reuse. Moreover, the simulation times at the behavior level could be as much as 20 times shorter than simulation times at the RT level [2].

Figure 1.1 illustrates a design flow starting at the behavior level. The HLS tool reads the behavior description into an internal design representation on which the architectural optimization algorithms operate. The optimal RTL design generated by the HLS tool undergoes further gate level and physical design optimizations. Each step in the design cycle is followed by functional verification. In Figure 1.1 we show the points where verification is performed as shaded squares, and indicate possible verification methods that can be applied for each level of abstraction. Tradi-
tionally, verification is done by simulation, which is much faster at behavioral level. Unfortunately, the use of HLS tools does not eliminate the need to verify the synthesized RTL designs.

![Figure 1.1: Example of a Design Flow and Points of Verification](image)

In spite of the increased confidence one has in automatic synthesis when compared with manual design, the use of HLS tools does not considerably ease the burden of verification. HLS tools employ a set of complex optimization algorithms which are usually implemented as large pieces of code, hence error prone. Possible software errors in these implementations may lead to incorrect functionality of the synthesized designs. Another source of errors is the translation from the input specifications, usually written in a certain Hardware Description Language (HDL), to the internal representations on which the synthesis algorithms actually operate; this translation might alter the intended semantics of the input description. An ideal scenario would be a synthesis system free of
software errors and working on the same hardware description format as the input specification. Both aims are very difficult to achieve: synthesis software implementations are complex and their verification extremely tedious; hardware specification languages are diverse, intuitive and high-level, while synthesis algorithms are usually implemented in imperative languages such as C, and need to operate on well-established data structures that allow efficient implementations.

In order to bring more confidence in the HLS process, researchers have focused on the application of formal methods to synthesis systems. The relatively recent success in applying formal verification techniques to industrial size designs demonstrates the viability of the mathematically rigorous approaches to design verification. In the following section we summarize some of the formal approaches to design verification and correct synthesized designs.

1.1 Background and Related Work

The success of a verification method is determined by its efficiency and ease of use. Simulation is the traditional verification approach; it is highly automated and reasonably fast for medium size designs, but the multi-million transistor complexity of today’s designs renders it impractical. Even worse, simulation is an incomplete process that is known to leave room for errors (for example, the very expensive Pentium II Processor floating point unit bug [3]). This explains the ever increasing role of formal verification in industrial design. We briefly summarize here the main trends in formal verification [4]:

1. **Theorem Proving** is one of the earliest approaches to formal hardware verification [5]. A theorem proving environment consists of a formal language for specifying the behavior and its implementation, and a formal deductive system that allows proving any true property expressable using the formal specification language. The choice of the underlying logic for a theorem proving system is a tradeoff between efficiency and expressiveness. Systems based on first-order logic (such as ACL2 [6]) are more suited for proof automation, hence more efficient. In comparison, specification languages based on higher-order logic are more expressive, but proof systems based on it are harder to automate (for example HOL [5]). The Prototype Verification System (PVS) [7] is an example of a theorem proving environment based on a typed higher-order logic that also enjoys a high degree of automation. We describe PVS in more detail further in this thesis, as most of the work discussed here is done using PVS.
2. **Model Checking** is a verification technique that ascertains the correctness of designs modeled as Finite State Machines (FSMs). It relies on algorithms and data structures that allow an effective search of the FSM state space in order to check for possible violations of properties (specifications of behavior). There are two main approaches to model checking [8]:

- **Property Model Checking**, where the specifications are expressed as temporal logic properties (written in temporal logic languages such as CTL [9]).
- **Automata Based Approaches**, where the correctness of the implementation is checked either by containment of languages recognized by automata on infinite words [10, 11], or according to certain conformance criteria based on the equivalence of observable states [12].

Model checking is completely automatic but suffers from the state explosion problem, especially for datapath oriented designs. The use of BDDs [13] to encode state transition systems [14], and more recently - the use of SAT-based techniques [15, 16], alleviate the state explosion problem to that point where model checkers can be successfully used to verify some industrial size applications (with \( \approx 10^{120} \) reachable states) [8].

When contemplating formal verification of *synthesized* designs, one can resort to one of these two general approaches, or to a combination of them. Both approaches have strengths and weaknesses which are continuously addressed by the research community [4, 8]. Model checkers are preferred for their automatic use, but they are quite restricted in the size of the designs they can handle, due to the state explosion problem. Theorem provers allow a parametric interpretation of datapaths, thus eliminating state explosion, but are less automated and usually require considerable knowledge from the user. As an alternative, many research efforts are focus on ensuring the correctness of the synthesis process itself, thus trying to eliminate the need of a difficult post-facto verification.

1.1.1 Approaches to Correct Synthesis

There are three main approaches to formal correctness of synthesis [17]: *pre-synthesis verification*, *formal synthesis* and *post-synthesis verification*:

- **Pre-synthesis verification** refers to formally checking the correctness of the synthesis tools software implementation; correctness of the implementation ensures correct-by-construction designs. As noted before, synthesis algorithms are very complex and difficult to verify. Moreover, this effort has to be repeated for every new implementation or enhancement of an existing one. One of the few applications of this approach is the verification of a weak division algorithm for Boolean simplification, using Nuprl proof assistant [18].
• **Post-synthesis verification** is oblivious to *how* the synthesis is performed. The synthesized design is checked using general formal verification methods (such as theorem proving or model checking) without considering any knowledge about the synthesis process.

• **Formal Synthesis** defines a synthesis methodology where the implementation is derived from the specification through a sequence of refinement steps which are proved to preserve the specified behavior. The synthesized designs are correct-by-construction since only those changes to the initial specification are allowed that are not altering the input specifications behavior. In what follows we refer to HLS systems based on the application of such behavior preserving transformations as *transformational derivation systems*. Later in this chapter we present an overview of existing transformational derivation systems.

Several other contributions to synthesized designs verification are briefly summarized here. They do not fall precisely into any of these three main trends, but specifically address verification of the synthesis result by taking into account knowledge about the steps performed during the HLS process.

Narasimhan [19, 20] develops an assertion-based methodology to verify high-level synthesis algorithms. Several HLS algorithms are formalized and proved correct in PVS. A set of assertions are systematically identified during the formal proof of correctness for these algorithms as subgoals written in the PVS language. They are further expressed as C assertions and correspondingly inserted into the software implementation. If no assertion is fired during synthesis, then the resulting implementation is correct. If an assertion is fired, then the corresponding subgoal gives information about the error encountered. This information is used in debugging the synthesis software implementation.

Mansouri [21] proposed a verification methodology where both input specification and the synthesized implementation are specified in PVS. Based on knowledge supplied by the synthesis tool, a correctness condition generator extracts a set of lemmas that assert the equivalence between specification and implementation at observable critical states. The correctness lemmas as well as proof scripts are automatically generated and run without further human interaction. The method has been implemented in a verification tool (the correctness condition generator, CCG) and is integrated with a high-level synthesis system.

Eveking et al. [22] developed a method for verifying equivalence of designs before and after the scheduling task. Their approach is independent of the particular scheduling algorithm used and works by assimilating an initial design into a scheduled design through transformational steps.
Lock et al. [23] proposed a method for proving the synthesized designs correctness by checking a set of correctness properties for an intermediate result that captures the binding information.

A verification and debugging methodology using witness generators (WG) is presented by Radhakrishnan et al [24, 25]. The method is based on the observation that it is possible to generate a sequence of elementary transformations that perform the same tasks as a traditional high-level synthesis tool, but in a transformational manner. Identifying such sequences of transformations can be done by examining the binding data structures supplied by the synthesis tool, without any knowledge about the synthesis algorithm used to perform the task. For example, a sequence of register instance substitutions can be identified by looking at the register binding table without knowing whether a clique partitioning algorithm or a left-edge algorithm was used for register allocation. In Sections 1.2 and 6.3 we describe this method in more detail and show how the work discussed in this thesis supports verification of synthesized designs using witness generators.

This WG method has similar goals as previous research [19, 20], that is, to verify a synthesized design and, if incorrect, provide help in determining the location of the software error in the synthesis tool. In addition, WG is readily applicable to a wide variety of high-level synthesis algorithms. The same generality enjoys the method of Eveking [22], but it refers only to scheduling algorithms.

1.1.2 Transformational Derivation (TD) Systems

In a Transformational Derivation (TD) System the implementation is derived from the specification through a sequence of behavior preserving transformations. The transformations used for synthesis can be classified into two categories [17]: general purpose transformations, which are logical transformations applied to an input behavior description, and hardware-specific transformations, which are circuit transformations applied to a hardware representation. Hardware-specific transformations can be applied at different levels of abstraction: gate-level (such as logic optimization transformations), RT Level (such as ALU folding, register instance substitution) or system level.

The above classification concerns the representations to which transformations are applied. Another classification refers to how the transformations interpret these representations. In this respect we can distinguish two categories: interpreted transformations, that consider the semantics of functions and constructs within an interpretation domain, and uninterpreted transformations, that consider only the structural aspects of a representation. Among general purpose transformations, commutative rewriting, associative rewriting and fold/unfold transformations [26] are interpreted, code motion is sometimes uninterpreted. As examples of hardware-specific transformations, logic optimization transformations consider the functions implemented by gates (e.g., AND, OR), hence
they are interpreted. ALU folding is concerned only with the components’ interconnections and checks that the folded ALUs have the same functionality (without interpreting this functionality), so they are uninterpreted. Figure 1.2 illustrates these classifications.

<table>
<thead>
<tr>
<th></th>
<th>General Purpose</th>
<th>Hardware Specific</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Interpreted</strong></td>
<td>e.g. dead-code elimination</td>
<td>gate level</td>
</tr>
<tr>
<td></td>
<td></td>
<td>e.g logic optimization</td>
</tr>
<tr>
<td><strong>Uninterpreted</strong></td>
<td>e.g. code motion</td>
<td>e.g. ALU folding</td>
</tr>
</tbody>
</table>

Figure 1.2: Classification of Elementary Transformations

The transformations presented in this thesis are uninterpreted RTL transformations.

Kumar et al.[17] formulated two criteria for comparing TD systems: efficiency and security. General purpose transformations are applied at higher levels of abstraction and they result in more powerful optimizations. Hardware-specific transformations are better suited for synthesis and their software implementations are small and less error prone. However, strategies in applying these transformations may become as complex as classical synthesis algorithms making them less secure. Another advantage of RTL transformations is that they can be based on more accurate performance estimates.

In the remaining of this section we present an overview of the existing TD Systems. They are presented according to the first criterion for classification, as TD systems based on hardware-specific transformations, and TD systems based on general purpose transformations.

**TD Systems Based on Hardware-Specific Transformations:**

- **RLEXT - Register Level EXploration Tool** [27] transforms a register-level datapath specification into a possible optimized implementation by applying a user-guided sequence of hardware-specific transformations: addition/deletion of operators, path elements and connections, and rescheduling of transfers. A checking procedure can be invoked by the user at any time in order to locate and repair possible inconsistencies introduced by these transformations. Though hardware descriptions are formally defined, RLEXT does not provide formal
proofs of correctness.

- **T-Ruby** [28] is a hardware description language based on abstract relations as means to specify circuit behavior. These relations and their combining rules generate an algebra that defines equivalences between relational expressions. Relations are models for circuit elements, as well as a graphical interpretation corresponding to an abstract floorplan for the circuits they describe. The language forms the basis of a TD system where transformations take the form of rewrite rules. A user input rewrite rule is passed to a theorem prover along with a corresponding precondition in order to check its correctness.

- **DDD - Digital Design Derivation** [29] is based on the algebraic manipulation of functional specifications. Both specification and implementation are written as executable programs in a dialect of LISP. The behavior is expressed as an iterative system of tail-recursive functions. Initial transformations derive a *structure* in the form of a controller operating on an abstract architecture. Next, *factorization* transformations are applied (such as assigning common subexpressions to the same device, creating a counter from an incrementer). Factorizations are followed by a set of transformations that introduce more concrete representations for constants and operations. The final representation is integrated with existing logic synthesis tools, such as boolean equation minimizers and VLSI layout generators. Some of the DDD transformations are sophisticated but apparently they have not been mechanically verified for correctness.

**TD Systems Based on General Purpose Transformations:**

- **LAMBDA/DIALOG** [30] represents the current state of a design as a locally-valid rule that is altered by user input refinement steps. The approach of LAMBDA to formal synthesis can be expressed as a rule of logic:

  \[
  \begin{align*}
  \text{IF} & \quad \text{present-design} + \text{additions} \quad \text{ACHIEVES} \quad \text{task1} \\
  \text{AND} & \quad \text{........} \\
  \text{AND} & \quad \text{present-design} + \text{additions} \quad \text{ACHIEVES} \quad \text{taskn} \\
  \text{THEN} & \quad \text{present-design} + \text{additions} \quad \text{ACHIEVES} \quad \text{specification}
  \end{align*}
  \]

Each refinement step changes the current rule (design state) and reformulates the *tasks* such that after each refinement (transformation) the design is closer to the final implementation and the *tasks* are simpler. The logic of LAMBDA uses higher-order polymorphic predicate calculus with partial terms. The user interaction is facilitated by the DIALOG graphical interface.
• **HASH** [31] uses data-flow graphs to represent input behaviors formally specified in HOL. Each step in the synthesis process represents a transformation in HASH and is implemented in HOL. As a result, heuristics for scheduling, resource binding and optimization can be performed outside the logic, and then input back into the HOL proving environment as a (meta) transformation whose validity has to be checked.

• **VERITAS** [32] is a higher-order logic based on type-theory used for both specification and formal proof of correctness. Circuits are represented as consisting of a set of nodes between which components (gates, ALUs) are strung. Possible transformations to be applied to this representation are organized as a set of 8 techniques; a technique is a pair of functions: a subgoaling function, that expresses the transformation, and a validating function, that creates a validating theorem of the form $\vdash Impl \Rightarrow Spec$. The set of techniques is not complete, but remains open such that a user can enhance it.

### 1.2 Motivation

The work presented in this thesis establishes the formal foundation on which a transformational derivation system using a core set of uninterpreted RTL transformations can be based. The set of transformations discussed here affect the assignment of operations to control steps, and the sharing of resources, i.e., those optimizations performed by the scheduling and allocation steps of the HLS process. This close association with traditional synthesis tasks hints at the automation of a TD system similar to that of conventional synthesis algorithms: a derivational synthesis algorithm is a performance-guided strategy of applying sequences of elementary changes.

The use of RTL transformations for synthesis does not exclude a pre-synthesis optimization step, where general purpose transformations could be applied in order to obtain an optimized behavior description. For the purpose of this thesis we consider our set of transformations to be used independently of any pre-synthesis transformational step, although a pre-optimized behavioral description is hard to define apart from structural changes. This separation on the scope of the core set of transformations to be used by a certain derivational tool will possibly diminish its optimization capability, but has clear advantages that address the efficiency and soundness of a derivational synthesis tool based on this set.

In what follows we enumerate three important issues that have to be considered in designing the core set of transformations for a derivational tool, and state the contributions of our work relative to each issue:
• **Soundness of RTL transformations.** *A core set should contain a small number of elementary transformations; the correctness of these transformations should be unambiguously established.* Based on a formal model for RTL designs we formally specify in PVS a set of ten RTL transformations. We then mechanically prove that each of the ten transformations of our set is correct.\(^1\) A transformation is correct if it preserves the computational behavior of the design.

• **Suitability for automation.** *The automation of a TD synthesis system assumes the design of analytic or heuristic synthesis algorithms which are guided by performance measures/costs (such as area or latency).* More obvious than in the case of general purpose transformations, hardware changes can be evaluated in terms of their impact on the design performance. Sharing of an operator results in a concrete change in area and change of the routing configuration; splitting of a register transfer increases the number of control steps and affects the area. These more accurate performance estimates are important to a synthesis algorithm. In this work we are concerned only with the functional aspects of transformations.

• **Capability to explore the design space.** *A TD synthesis tool should be capable of exploring the design space and present the user with a set of RTL designs that implement the desired functionality and satisfy a set of performance constraints.* For real size designs the exhaustive enumeration of each possible implementation is desirable but impractical. The use of heuristic approaches to find a local optimum solution seems to be a good compromise and is currently preferred in industrial synthesis systems. In the case of TD systems a new problem arises: *Are the core transformations elementary enough such that possibly better implementations are not overlooked?* In the case of a core set consisting of RTL transformations this question can be positively answered: *There exists a small and complete set of RTL transformations by sequences of which any possible implementation in the design space can be reached.*

---

\(^1\)In this thesis we do not make a distinction between the terms **correctness** and **soundness**.
others to validate them. In contrast, we present a proof that is fully mechanized in the higher-order logic of PVS and can be run by a simple command in the proof environment.

The above considerations address only TD synthesis, but there are also verification applications supported by a complete set of correct transformations. Any HLS algorithm that performs only structural optimizations can be viewed as a sequential application of RTL transformations. Having identified a complete set of transformations, then we can assert 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 the synthesis process by attempting to identify such sequences. This idea is used [24, 25] as an approach to synthesis software debugging. In Figure 1.3, each entry in the binding data structures generated by the synthesis tool is automatically identified with a transformation and recorded as such in a witness sequence. At the same time, the transformation is applied to an initial design representation where one can check if its preconditions are satisfied, and if not, record the failed precondition; this failed precondition gives information about an error in the synthesis software.

![Figure 1.3: Witness Generator Method Approach](image-url)
1.3 Overview of the Research

The contributions of the work presented here are the following:

- **A formal model for RTL designs corresponding to behavior descriptions consisting of straight-line code.** Our formal model is abstract and uninterpreted; the model is abstract because it is not tied to any syntactic description for RTL designs; the model is uninterpreted because we do not assign any computational semantics to its components.

- **Formal models and mechanized proofs of correctness for a set of ten RTL transformations.** Based on formal model we formally specify a set of ten uninterpreted RTL transformations. The correctness of a transformation is defined as the behavior preserving property: a transformation is correct if the computational behavior of the output registers is not modified. The correctness of every transformation is subject to a set of preconditions. This is expressed by the following Correctness Theorem:

\[
T_{Preconditions}(\text{rtl_impl}, \text{set_of_modules}) \Rightarrow \\
\text{computational_behavior}(T(\text{rtl_impl}, \text{set_of_modules})) = \text{computational_behavior}(\text{rtl_impl})
\]

where \(T_{Preconditions}\) is a predicate function associated to a generic transformation \(T\). It returns true if certain properties of a \(\text{set_of_modules}\) (operators or registers) of the design \(\text{rtl_impl}\) are satisfied. The function \(\text{computational_behavior}\) extracts the behavior of the output registers (corresponding to the output variables in the behavior description).

- **A mechanized proof of completeness for the set of RTL transformations.** A finite set of transformations is said to be complete for a certain class of designs if any correct implementation of a given behavior can be reached using a finite sequence of transformations from this set. We mechanically proved in the PVS environment that our set of RTL transformations is complete.

Although the proof for completeness presented in this thesis follows the main general idea of Vemuri [36], it differs essentially in the proof strategies employed. The mechanized proof of completeness presented here may validate (in principle) the existing manual proof but can not claim to discover any eventual inaccuracies. That is because, although similar in spirit, the models and the transformations presented here differ from those of Vemuri [33, 34, 35, 36]. However, our specifications clarify some complex definitions for temporary registers and normal form structure (which is “unique (within naming of the hardware components)” [34]).

The mechanized proving approach revealed some particular cases in transformations’ application that were overlooked in a paper-and-pencil reasoning style. The precise definitional specification
approach, as well as the rigorous mechanized proof, are able to account exhaustively for each particular case, while in the paper-and-pencil style we tend to subjectively consider only some cases.

The specification and proof exercise are conducted in PVS [7, 37, 38, 39]. PVS is a theorem proving environment based on a typed higher order logic enhanced with dependent and predicate subtypes. In specifying the models we used a definitional approach such that only few axioms are introduced. The definitional specification is facilitated by a variety of constructs built in the PVS language [38] (predicate subtypes, record structures, abstract datatypes [40]).

“It is through this specification process that developers uncover design flaws, inconsistencies, ambiguity and incompleteness” [8]. On several occasions, inconsistent definitions for register-transfer models were discovered by the PVS’s type-checker, and an incomplete specification for one transformation’s precondition was discovered during the proof exercise. After this, we actually attempted to find weaker preconditions by iteratively inserting into the antecedent partial preconditions until a proof was derived. Finding weaker preconditions is important for two reasons:

1. A synthesis system that implements these transformations is more time-efficient if it has to check for fewer preconditions, and

2. A software implementation debugging tool based on witness generation is more accurate if the witness sequence records the violation of a weaker precondition than an unnecessarily stronger one.

1.4 Organization

In Chapter 2 we present the formal model for RTL designs corresponding to specifications consisting of basic blocks of straight line code with the operations and operators assumed to be binary. Chapter 3 informally summarizes each of the ten RTL transformations, and is followed by the formal definition and proofs of correctness in Chapter 4. A proof of completeness for this set of transformations is detailed in Chapter 5. Each of the formal definitions introduced in these chapters are accompanied by their formalization in PVS. Insights on the practical applicability of the formal result of this work are presented in Chapter 6. We conclude and state the limitations and future work in Chapter 7. The Appendix contains descriptions of the proof strategies used and some of the lengthy specifications we preferred not to include in the previous chapters for clarity reasons.
Chapter 2

Formal Models for Behavior Descriptions and RTL Designs

A formal specification (model) is an abstract description of a system written using a rigorous mathematical notation which has a precise semantics within an interpretation domain. The validity of a property of the model can be checked using a sound and complete set of rules (a logic calculus). As compared with a hardware description language (for example in VHDL), a formal specification of an RTL design is a provable, abstract and usually non-executable model. However, an executable formal model can be obtained, for example, by embedding the semantics of an existing hardware description language into a formal reasoning system (like the embedding of ELLA, SILAGE and VHDL languages in HOL [41]), or by extending the language of an existing proving environment in order to allow the specification of executable hardware models (e.g., ACL2 [6]) as an extension of Nqthm).

In this work we are not concerned with the executability of our models since we do not intend to check a set of various properties for each individual design (as simulation would do), but rather check if a certain property stands for an entire class of designs. For example, we do not investigate the effect of a transformation on a certain implementation, but on an entire class of implementations. As opposed to a netlist description, where interconnections between labeled components are defined, our model specifies an abstract RTL design as a set of properties asserted about abstract collections (sets) of components (operators and registers). Structural RTL transformations are functions operating on elements of the set defined by these properties, and the completeness property is a higher-order predicate that defines a complete set of functions.

Casting the model according to the scope of the proof is typical to transformations verification. Most of the derivational systems presented in Chapter 1 use specialized formal description languages
to specify the input at that level of abstraction where transformations are applied. Models for both
designs and transformations are oriented toward a clear formulation of the verification goal: in
the T-Ruby system designs are represented in terms of abstract relations and transformations are
applied as rewrite rules that result in relations which are necessarily equivalent with the original
ones; the LAMBDA system views designs as rules: a new (transformed) design is a new rule
obtained by unifying the old rule with the premises of the new one, etc.

In comparison with the above formalisms, our models are abstract, in the sense that they are not
tied to any syntactical description. A similar goal is followed by Rajan [42] but for specifying pre-
synthesis transformations applied to dependency graphs. Here, powerful abstractions are achieved
by defining the models in a property-oriented approach, that is, the model is characterized by a set
of axioms.

Axiomatic definitions are allowed in the PVS environment, but they are susceptible to introducing
inconsistencies in the specification. As an alternative, a model-oriented definitional approach is
facilitated by a variety of constructs built in the system’s language [38] (predicate subtypes, record
structures, abstract datatypes [40]).

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

Similar models were discussed by Blumenrohr [43] and Vemuri [35]. Formal definitions for data
paths, control graphs and register transfers are associated with the semantics of these structures,
and are formalized in the language of the PVS system. The use of PVS for the formal treatment
presented in this thesis is justified not only by the typed higher-order logic that underlies its
specification and proving environment (required by the very nature of the problem we are dealing
with), but also by its rich language constructs that allow expressive and efficient specifications. A
brief introduction into the PVS language and proof environment can be found in Appendix A

2.1 Formal Definitions for an Abstract Model for RTL Designs

An RTL design consists of data-path and a controller that defines the sequence of register transfers
to be executed in the data-path. Figure 2.1 shows a datapath-controller implementation for the
straight line code behavioral input represented as a flow graph in Figure 2.1.b.

The data path of an RTL design consists of a set of operators, a set of registers and interconnections
between them. In what follows we 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 \textit{data path structure} a tuple of the form: \((E, OP, REG)\). A register transfer can be viewed as a ‘snapshot’ of a portion of the datapath at a certain control step. The entire functionality of the design is ‘animated’ by a controlled succession of register transfers defined as a \textit{control graph}. A register transfer 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 \( rt \).

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

\[
(EXPR; REG_{\text{out}}; f_{\text{op}} : OP \rightarrow (E \times E), f_{\text{reg}} : REG_{\text{out}} \rightarrow E)
\]

where \( EXPR \subseteq E \), and \( REG_{\text{out}} \subseteq REG \). We call \( REG_{\text{out}} \) the set of output registers of \( rt \). \( f_{\text{op}} \) and \( f_{\text{reg}} \) define the interconnections between components of the data path at the control step corresponding to \( rt \) as follows: \( f_{\text{op}} \) is a function mapping a component to a pair of components (its sources), and \( f_{\text{reg}} \) is a function that maps an output register to a component (its input).
For example, in Figure 2.1 the components of the register transfer \(rt3\) are:

\[
EXPR = \{ \text{OP1, OP2, T1, T2, C} \} \quad f_{op}(\text{OP1}) = (\text{OP2, C}) \quad f_{reg}(D) = \text{OP1}
\]

\[
REG_{out} = \{ D \} \quad f_{op}(\text{OP2}) = (\text{T1, T2})
\]

In PVS it was more convenient to represent the operators, input registers and their interconnections as binary tree structures where leaf nodes are the input registers. Thus, the \(f_{op}\) function will be defined implicitly in the PVS specification.

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 do not allow a register transfer that contains combinational cycles, floating inputs for operators and registers or concurrent operations to be performed on the same hardware resource. To formally define such requirements we first introduce the definition of the \(ancestors\) set for a component \(e\) (operator or register) as the set of all components connected to \(e\) through a direct path.

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

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

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 \(EXPR\), and 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) \(\forall (e \in EXPR) : (e \notin ancestors(e));\)
3) \(\text{image}(f_{reg}) \subseteq EXPR;\)
4) \(\forall (e1, e2 \in (EXPR \cap OP)) : (e1 = e2) \Rightarrow (f_{op}(e1)^1 = f_{op}(e2)^1 \land f_{op}(e1)^2 = f_{op}(e2)^2).\)
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:

\[
\text{control}_\text{graph}(E, OP, REG) = \langle rt_1 \rangle, \langle rt_2 \rangle \ldots \langle rt_n \rangle.
\]

where \(rt_1, rt_2 \ldots \) \(rt_n\) are well-formed register transfers associated with the data path structure \((E, OP, REG)\).

Control graphs specified as ordered sequences of well-formed register transfers completely define RTL designs functionality: interconnections between data-path components are determined from the constituent register transfers; the control information supplied to the data-path is defined by the ordering of transfers. With this interpretation, when we refer to a property of a design, we actually refer to a property of the control graph that defines that design.

### 2.2 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. 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: \(<\text{beh}> ::= \text{<variable>} | (<\text{beh}> \text{<operation>} <\text{beh}>),\) where \text{operation} is any binary arithmetic or logic function. Most specification languages use the above form to describe the syntax of computations. In our models we identify a variable with a register instance that holds its value. With this simplification, we can define a behavior description as:

**Definition 4.** A behavior description is a tuple of the form \((REGS, f_{assign} : REGS \rightarrow \text{beh})\), where \text{REGS} is a set of registers and \(f_{assign}\) is a function that assigns to each variable in \text{REGS} a behavior expression \(\text{beh}\).

A trivial RTL implementation of a given behavior description can be obtained by assigning to each operation in the behavior description an operator instance that performs that operation.
Conversely, each register transfer can be associated with a behavior description whose \( \text{REGS} \) set corresponds to the \( \text{REG}_{\text{out}} \) set, and the \( \text{f}_{\text{assign}} \) function corresponds to \( \text{f}_{\text{reg}} \) and assigns to each output register a behavior expression obtained by traversing the operator trees.

The recursive function that extracts a behavior expression associated to a component of an implementation is defined by two mutually recursive functions:

**Definition 5.** The extracted behavior of a component \( e \) in a control graph list \( cg \), with respect to a mapping function \( f_{\text{op}} : \text{OP} \rightarrow (\text{EXPR} \times \text{EXPR}) \), is defined using two mutually recursive functions:

\[
eb1(e, cg) = \begin{cases} 
\eb2(e, \text{cdr}(cg)) & e : \text{reg} \\
(eb1(\text{car}(cg) \cdot f_{\text{op}}(e)\cdot1), \text{opf}(e), eb1(\text{car}(cg) \cdot f_{\text{op}}(e)\cdot2)) & e : \text{op}
\end{cases}
\]

\[
eb2(r, cg) = \begin{cases} 
\eb1(\text{car}(cg) \cdot f_{\text{reg}}(r), cg) & r : \text{reg} \\
\eb2(r, \text{cdr}(cg)) & r \notin \text{car}(cg) \cdot \text{REG}_{\text{out}}
\end{cases}
\]

where \( \text{opf}(e) \) denotes the operation performed by an operator \( e \), \( r \) is a register, \( \text{car} \) and \( \text{cdr} \) are the LISP notations for functions used to create and manipulate lists. The function \( \eb1 \) defines the recursive traversal of operator trees, and terminates when a leaf node (a register) is encountered. The function \( \eb2 \) defines the recursive traversal of the control graph list and terminates when the list becomes \( \text{null} \). When referring 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.

### 2.3 PVS Specifications for the Abstract Model

We present in this section the PVS specifications for the models defined so far (a brief introduction into the PVS language and prover is in Appendix A). The main idea followed is to 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**;
- interconnections between operators, and between operators and input registers are binary tree
behavior expressions are binary tree structure datatypes;

- 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.

Each of these interpretations are supported by the PVS language and some of them, as it will be pointed out later, by the type-checking and proving environment.

2.3.1 PVS Specifications for Register Transfers

The interconnections between operators, and between operators and input registers (corresponding to \( f_{op} \) in Definition 1) are implicitly defined using the binary-tree structure expression, specified as a parameterized PVS recursive abstract DATATYPE [40]. An expression can be a register expression (the leaf element in the binary-tree structure), or an operator expression (the node element).

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

A similar approach is proposed by Melham [5] to model tree-shaped combinational structures using recursive binary trees defined in HOL. Like HOL, PVS supports only total functions such that any recursive definition is required to terminate. When PVS type-checks the expression theory, it automatically asserts a well-foundedness axiom to ensure the termination of inductive definitions. As a corollary, no expression can be its own ancestor; in our model this translates to operator expressions with no combinational cycles, which suits the second well-formedness condition in Definition 3.

A register transfer is viewed as an assignment of expressions (from exp-set) to a set of target registers (in outregs). According to this definition, the field exp-set corresponds to \( EXPR \) and
The concept of operator expressions as binary-tree structures requires a slight reformulation of the fourth well-formedness condition stated in Definition 3: any two expressions rooted at the same operator must be equal. This is similar to enforcing the existence of a unique physical instance of an operator within a register transfer. As shown before, combinational cycles are prohibited by the very logic of PVS. What remains to be enforced by the predicate subtype definition of a well-formed register transfer is that there are no floating inputs, that is, every operator or output register always reads a value within the transfer that contains

\[ f_{op}, \text{outregs} \text{ to } REG_{out} \text{ and } \text{regassign} \text{ to } f_{out} \text{ in Definition 1.} \]

Figure 2.2 illustrates the fields of a register transfer. All operators and input registers are nodes of an expression tree contained in the \text{exp_set} field; output registers (\text{R7, R8 and R9}) are elements of the \text{outregs} set; the \text{regassign} function defines the interconnections between output registers and an operator or register instance.

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

Figure 2.2: The Fields of a Register Transfer

Well-Formed Register Transfers. The definition of operator expressions as binary-tree structures requires a slight reformulation of the fourth well-formedness condition stated in Definition 3: any two expressions rooted at the same operator must be equal. This is similar to enforcing the existence of a unique physical instance of an operator within a register transfer. As shown before, combinational cycles are prohibited by the very logic of PVS. What remains to be enforced by the predicate subtype definition of a well-formed register transfer is that there are no floating inputs, that is, every operator or output register always reads a value within the transfer that contains
them (conditions 1 and 3 in Definition 3):

wellformed_set?(e_set: finite_set[expression]): bool = 
(forall (e:(e_set)) : subset?(ancestors(e), e_set)) % well-formed condition 1 
AND 
(forall (e1,e2: (e_set)) : 
(op?(e1) AND op?(e2) AND op(e1) = op(e2)) => e1 = e2) % well-formed condition 4

wellformed_out?(rt:transfer) : bool = 
subset?(image(rt`regassign)(rt`outregs), rt`exp_set) % well-formed condition

wellformed_rt?(rt:transfer): bool = wellformed_set?(rt`exp_set) and wellformed_out?(rt)

wellformed_rt : TYPE = {rt:transfer | wellformed_rt?(rt)}

Equivalent Register Transfers. The well-formedness of a register transfer is defined by a minimal set of conditions which exclude configurations that should not be allowed in practice, but include those that are sound with respect to our proving goal. We did not enforce that an operator or register expression belonging to the exp_set must have a target, thus allowing expressions which are not used in any computations. There is no significance associated with these spurious expressions, so they were simply abstracted out from the model. Likewise, the regassign function assigns an expression to a register expression that may or may not belong to the set of output registers (outregs). Disallowing spurious expressions and restricting the range of the regassign function would lead to numerous and unnecessary type-checking conditions being carried along each further specification and proof.

However, there is an instance when this model underspecifies register transfers. Two register transfers having the same output registers reading values from the same expression trees are, for all practical purposes, equivalent. But they may not be strictly equal because of the spurious expressions in the exp_set, and the extended function definition for regassign. To deal with this, we define an equivalence relation that accounts only for the useful aspects of a register transfer. In the following PVS specification, the eq?(rt1,rt2:transfer) predicate is true if register transfers rt1 and rt2 have equal outregs fields, and their regassign fields assign each register expression in the corresponding outregs sets to equal expressions. The eq? relation is reflexive, symmetric and transitive, hence it is an equivalence relation. The eq? relation is extended to lists of register transfers by the eq?(cg1,cg2:list[transfer]) overloading definition.
eq?(rt1,rt2:transfer) : bool = 
    rt1`outregs = rt2`outregs and 
    forall(r:(rt1`outregs)) : rt1`regassign(r) = rt2`regassign(r)

eq_rt_transitivity: LEMMA 
forall(rt1,rt2,rt3:transfer) : (eq?(rt1,rt2) and eq?(rt2,rt3)) => eq?(rt1,rt3)

eq_iden: LEMMA forall(rt:transfer) : eq?(cg,cg)

eq_comm: LEMMA forall(rt1,rt2:transfer) : eq?(rt1,rt2) => eq?(rt2,rt1)

eq?(cg1,cg2:list[transfer]) : bool = 
    (length(cg1) = length(cg2) and 
    forall(n:below[length(cg1)]) : eq?(nth(cg1,n),nth(cg2,n)))

The class representative element for this equivalence relation is defined by the clean rt?(rt) predicate shown below, which is true if register transfer rt has no spurious expressions. Some results presented later in this thesis are conditioned upon the satisfaction of this predicate. In the view of considerations presented here, this should not be viewed as a restriction on these results, but as a legitimate assumption about the model.

clean_rt?(rt:transfer) : bool = 
forall(e:(rt`exp_set)) : 
    ((exists(e1:(filter_op(rt`exp_set))) : source1(e1) = e or source2(e1) = e) 
    OR 
    (exists(r:(rt`outregs)) : rt`regassign(r) = e))

2.3.2 PVS Specification for Control Graphs

An RTL implementation of a straight-line code description is completely defined by a control graph viewed as a sequence of register transfers. This suggests a list model for control graphs. In PVS a list is defined as a built-in recursive abstract datatype and is supported by an extensive library of functions. It was convenient then to model control graphs as lists of well-formed register transfers:

c_graph: TYPE = list[wellformed_rt]
2.3.3 PVS Specifications for Behavior Descriptions

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 expression and behavior types facilitated both specifications and proofs for our models. PVS automatically generates basic declarations and axioms that formalize an abstract datatype, including extensionality axioms, induction schemes and recursion combinators. The application of induction and extensionality is done automatically in the proving environment.

A behavior description is defined as a function type beh_description which assigns each register a behavior expression.

```
behavior[fn: TYPE, instance: TYPE]: DATATYPE BEGIN
  leaf(v: instance): leaf?
  exp(f: fn, left: behavior, right: behavior): exp?
END behavior

beh: TYPE = behavior[operation, expression]
beh_description : TYPE = [(reg?)->beh]
```

2.3.4 PVS Definitions for Extracted Behaviors

The function extracted_behavior below corresponds to the eb1 function in Definition 5. It assigns to each expression e (operator or register) a unique behavior expression:

```
extracted_behavior(e: expression) : RECURSIVE beh =
  if reg?(e) then leaf(e)
  else exp(opfn(op(e)), extracted_behavior(source1(e)), extracted_behavior(source2(e)))
endif
MEASURE reduce_nat((LAMBDA(r:register):0),
  (LAMBDA (op:operator),(n1:nat),(n2:nat):n1+n2+1))
```
The behavior expression returned by extracted behavior for an operator expression \( e \) can be interpreted as \textit{the computation performed by} \( e \). And all different expressions which have the same extracted behavior are interpreted as different implementations of that behavior.

\textbf{The Behavior of an Expression in a Control Graph} can be extracted by traversing the operator trees in successive register transfers, and converting them into behavior trees. Figure 2.3 illustrates the first two successive register transfers of a control graph \( cg \). \( cgr \) is the sub-list of \( cg \) obtained by removing the first transfer. Register transfer \( rt1 \) is executed after register transfer \( rt2 \). The computational behavior implemented in \( cg \) by the operator expression rooted at the operator instance \( OP3 \) in the register transfer \( rt1 \) is, in an informal notation:

\[
e_b(OP3, cg) = (e_b(OP2, cg) \circ fn(OP3) e_b(D, cg)) = (e_b(C, cg) \circ fn(OP2) e_b(D, cg)) \circ fn(OP3) e_b(D, cgr) (C \circ fn(OP2) e_b(D, rest(cg))) \circ fn(OP3) e_b(OP2, cgr) \ldots
\]

where \( e_b(E, CG) \) denotes the extracted behavior of the component \( E \) in the control graph \( CG \).

![Figure 2.3: Extracted Behavior of an Expression in a Control Graph](image)

The most obvious definition for \textit{extracted behaviors} makes use of mutual recursive functions (like in \textbf{Definition 5}), where the recursive traversal of the control graph list calls the recursive traversal of the operator trees which in turn calls the recursive traversal of the list when it reaches a leaf (register) node, etc. Unfortunately, PVS does not support mutual recursive functions. To avoid this we defined a \texttt{match} function that locates an output register in a control graph; the recursive extraction traverses the operator trees, and calls the \texttt{match} function when traversal reaches a leaf node. This definition for the extracted behavior has two drawbacks that considerably affected the proof’s length: 1) we had to introduce one more conditional in the function definition to deal with the leaf case for operator trees, 2) we had to define the \texttt{exp\_set} field as a \textit{finite} set of expressions,
in order to define a size measure for register transfers; this measure combined with the length of the control graph lists was used to define the recursion measure for the extracted behavior function.

In the following PVS definition of extracted behavior, cons_graph is the type of control graphs which contain at least one register transfer. The function instances defines the set of output registers and operator expressions of a register transfer, i.e., those elements that represent starting points for extracting the behavior within a register transfer. In case expression e is an input register, the function match identifies a sublist of cg whose first element contains expression e as an output register (it locates the register transfer where register e was last written), and the extraction resumes at this point. The recursive traversal terminates when all input registers are matched to the null list.

```pvs
match(e: (reg?), cg:c_graph) : RECURSIVE c_graph =
    if null?(cg) then null
    else if instances(car(cg))(e) then cg else match(e, cdr(cg)) endif
    endif
    MEASURE length(cg)

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))
```

Note. Throughout this thesis we will count the register transfers of a control graph in the extracted behavior direction (i.e, we will traverse the list from the last control step towards the first control step), and opposite the data flow direction. Thus, in the definitions above, car(cg) (which denotes the first element of a list) corresponds to the register transfer which is executed at the last control step.
The Extracted Behavior of a Register Instance in a Control Graph (as an element of the uninterpreted type `register`, not as a register expression (`reg?`)) is defined as follows:

\[
\text{extracted_behavior}(r: \text{register}, \text{cg}: \text{c_graph}) : \text{beh} = \\
\begin{cases} \\
\text{let } \text{m_cg}: \text{c_graph} = \text{match}(\text{reg}(r), \text{cg}) \text{ in} \\
\quad \text{if } \text{null?}(\text{m_cg}) \text{ then } \text{leaf}(\text{reg}(r)) \\
\quad \text{else } \text{extracted_behavior}(\text{m_cg}, \text{car}(\text{m_cg}) \text{'regassign}(\text{reg}(r))) \\
\end{cases}
\]

In the definition above the extracted behavior of a register `r` equals the leaf behavior expression (when `r` does not appear as a written register in `cg`), or the extracted behavior of that expression from which `r` reads its last written value (when `r` appears as a written register in `cg`). (Note the overloading of the `extracted_behavior` function).

The Behavior Implemented By a Control Graph is defined as a function `cg_behavior` which assigns each control graph `cg` a behavior description:

\[
\text{cg_behavior}(\text{cg}: \text{c_graph}): \text{beh_description} = \lambda(x: (\text{reg}?)): \text{extracted_behavior}(\text{reg}(x), \text{cg})
\]

Two control graphs are defined to be `equivalent` if they have the same set of output registers and the same extracted behavior. The set of output registers is the set of registers belonging to the `outreg` field of the register transfer executed at the last control step. In the specification bellow, the equivalence of two control graphs is defined by the `equiv_cg?` predicate:

\[
\text{output_register}(\text{cg}: \text{c_graph}) : \text{set}[(\text{reg}?)] = \text{outregs}(\text{car}(\text{cg})) \\
\text{equiv_cg?}(\text{cg1}, \text{cg2}: \text{c_graph}) : \text{bool} = \\
\quad \text{output_registers}(\text{cg1}) = \text{output_registers}(\text{cg2}) \\
\quad \text{and} \\
\quad \text{cg_behavior}(\text{cg1}) = \text{cg_behavior}(\text{cg2})
\]

27
2.4 Summary

In this chapter we presented definitions and PVS specifications for an abstract model for RTL designs and behavior descriptions. The goal was to find a minimal specification that captures all the important features of a design while abstracting out the irrelevant details. An over-specified model would considerably complicate the proofs for the more complex tasks of this work: defining and proving correctness and completeness for a set of transformations. On the other hand, an under-specified model might admit RTL descriptions that should not be allowed in practice. The definition of a well-formed register transfer presented in this chapter is an example of avoiding illegal constructs by enforcing additional properties on an initially under-specified model. The price paid for using a minimal model is the need to define an equivalence relation as means to check if two designs are structurally equivalent. This impacts the complexity of the proofs since introducing user-defined equivalences prevents the user from automatically applying extensionality rules. However, the alternative (enforcing more restrictions on well-formed register transfers) would have been much more expensive: each property added to a predicate subtype definition results in more type-checking conditions, which in turn lead to more complicated definitions for functions operating on such types.

In order to establish the correctness of a transformation as a behavior preserving correctness property, we need to be able to check if two different RTL designs implement the same behavior description. The `extracted_behavior` function defined in this chapter assigns to each design the behavior description implemented by it. This function will be used in the next chapters to express the correctness theorems for uninterpreted RTL transformations.

We concentrated on an accurate representation of RTL designs and behavior descriptions using those PVS types that best describe them and, at the same time, take advantage of those features of PVS which facilitate descriptions and proofs.
Chapter 3

Informal Description of RTL Transformations

As shown in Section 1.2, there are three important issues that have to be considered in designing the core set of transformations for a transformational derivation (TD) tool:

- **Soundness of RTL transformations.** A core set should contain a small number of elementary transformations such that their correctness can be easily and unambiguously established.

- **Suitability for automation.** The automation of a transformational synthesis system assumes the design of analytic or heuristic synthesis algorithms which are guided by performance measures/costs (such as area or latency).

- **Capability to explore the design space.** A TD tool should be capable of exploring the design space and present the user with a set of RTL designs that implement the desired functionality and satisfy a set of performance constraints.

In Section 1.1.2 we presented an overview of transformational derivation systems based on hardware-specific transformations. With the exception of T-Ruby [28], all these systems are limited to the register transfer (RT) level and provide at most an interface with tools that perform further optimizations at lower levels of abstraction [29]. The soundness of transformations is addressed in all the above mentioned systems, but only DDD [29] is concerned with the automation of the synthesis process. Another algorithmic approach to transformational derivation is proposed by Camposano [44] and applied to improve the scheduling of RT level designs in the Yorktown Silicon Compiler [45].
The only complete set of correct RTL transformations is due to Vemuri [36, 34, 35]. Here, the design is defined as consisting of two components: a data-path graph and a control graph, where the data path denotes the structural composition of the hardware units, and the control graph defines the sequencing of register transfers conducted in the data path. A transformation makes changes only within a register transfer of the control graph and the corresponding portions of the data path.

Our complete set of RTL transformations follows the main ideas of Vemuri and are applied to similar structures. Several simplifications on the design representation were considered:

- RTL designs correspond to specifications which consist of basic blocks of straight line code (more suited for DSP applications).
- We allow only simple connection elements (like in designs targeted to FPGAs).

Our set of transformations can be extended to deal with if-then-else control blocks and buses such that the additional transformations would still preserve the completeness of the set.

It what follows we informally describe each transformation in the complete set. For each transformation we specify its inputs, function, preconditions that ensure the behavior-preserving correctness, and formulate the correctness criteria. We first introduce several notations we used for the conciseness and readability of the transformations’ descriptions:

1. $e \in rt$ if the instance $e$ (operator or register) appears in the register transfer $rt$. The formally correct notation is $e \in rt'EXPR$, where $EXPR$ is defined in Chapter 2 as a field of $rt$.
2. $\text{operator}'?(e)$ is a predicate which defines the set of all operator instances.
3. $\text{opfn}(e)$ is a function that defines the functionality of an operator instance.
4. $\text{source}1(e)$ and $\text{source}2(e)$ define the inputs of an operator instance.
5. $\text{comp\_behavior}(cg)$ defines the extracted behavior of all output registers of the control graph $cg$.
6. $\text{in\_regs}(rt)$ and $\text{out\_regs}(rt)$ define the set of input, respectively output registers of a register transfer $rt$.
7. $\text{operators}(rt)$ define the set of all operator instances that appear in a register transfer $rt$.  

30
3.1 Operator Copy (OC)

The OC transformation eliminates sharing of operator instances as illustrated in Figure 3.1. 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 operation each.

![Figure 3.1: The OC Transformation](image)

**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 instances (operators or registers) 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 first projection of $l$ ($l^1$), deletes $l$ and creates a new connection.

**Preconditions for OC:**
1. well-foundedness of $l$: a) $l$ should exist as a connection in $rt$, b) $l^1$ should be an operator,
2. $new_{op}$ and $l^1$ should have the same functionality,
3. $new_{op}$ should not be already in use in $rt$.

**Correctness Criteria:**

$OC\_Precondition(rt,l,new_{op}) \triangleq$

$(\exists(e \in rt) : (operator?(e) \land l^1 = e \land (source1(e) = l^2 \lor source2(e) = l^2))) \lor$

$(\exists(r \in in\_regs(rt)) : r = l^1 \land (f_{reg}(rt))(r) = l^2) \land$

$opfn(new_{op}) = opfn(l^1) \land \neg (new_{op} \in rt)$

$OC\_Precondition(rt,l,new_{op}) \Rightarrow comp\_behavior(cg) = comp\_behavior(OC(cg,rt,l,new_{op}))$

Figure 3.1 shows a register transfer (corresponding to $rt$) and the effect of the OC transformation applied to the marked link ($l$) and the new operator $OP1'$.
3.2 Operator Instance Substitution (OIS)

The OIS transformation allows the reuse of an operator instance which is available during a certain register transfer. In Figure 3.2 operators $OP_1$ and $OP_2$ are used to perform addition in two different register transfers. OIS substitutes the appearance of $OP_2$ by $OP_1$.

![Control Graph and Data Path for OIS Transformation](image)

**Figure 3.2: The OIS Transformation**

**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 $opr$.

**Preconditions for OIS:**
1. $subst$ and $opr$ should have the same functionality.
2. If $opr$ is already used in $rt$, than $subst$ and $opr$ should have the same functionality, same “left” sources and same “right” sources. In this case, OIS performs the inverse of OC.
Correctness Criteria:

\[
OIS\_Precondition(rt, subst, opr) \triangleq \\
(opf_n(subst) = opf_n(opr)) \land \\
(opr \in rt \Rightarrow (source1(opr) = source1(subst) \land source2(opr) = source2(subst)))
\]

\[
OIS\_Precondition(rt, subst, opr) \Rightarrow \text{comp}\_behavior(cg) = \text{comp}\_behavior(OIS(eg, rt, subst, opr))
\]

In Figure 3.2, \( rt \) is the first register transfer, \( subst \) is \( OP2 \) and \( opr \) is \( OP1 \). Note that the resource optimization algorithms of HLS tools can be viewed as sequences of OIS transformations.

### 3.3 Register Transfer Split (RTS)

RTS splits a register transfer into a sequence of two smaller register transfers by saving intermediate results into a set of new temporary registers. In Figure 3.3 the output of \( OP1 \) is initially input to \( OP2 \) in the same register transfer. RTS inserts the temporary register \( T1 \) and \( T2 \) to save the intermediate result of \( OP1 \) and transfers it to \( OP2 \) in the next register transfer. The inverse transformation of RTS, Register Transfer Merge (RTM) is presented in the next section.

**Figure 3.3: The RTS and RTM Transformations**

**Inputs of RTS:**

1. A register transfer \( rt \) of a control graph \( cg \) to which RTS is locally applied,
2. A set \( \text{split}\_set \) of operators and input registers to be scheduled in the first register transfer,
3. A set \( \text{reg}\_set \) of temporary registers.
Function of RTS: RTS creates 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 registers in `temp_set` should be “new” in the sense that they are not used anywhere else in the control graph.

Preconditions for RTS:
1. Every operator in `split_set` should have its sources in the same set, or as input registers of `rt` (data dependencies are preserved).

Correctness Criteria:

\[
RTS\_\text{Precondition}(rt; \text{split} \_\text{set}) \equiv \\
\forall (e \in \text{split} \_\text{set}) : (\text{split} \_\text{set} \subseteq \text{operators}(rt) \land \\
\text{operator}(e)) \Rightarrow (\text{source}1(e) \in \text{split} \_\text{set} \land \text{source}1(e) \in \text{split} \_\text{set})
\]

\[
RTS\_\text{Precondition}(rt; \text{split} \_\text{set}) \Rightarrow \text{comp} \_\text{behavior}(cg) = \text{comp} \_\text{behavior}(RTS(cg; rt; \text{split} \_\text{set}))
\]

In Figure 3.3, 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`. Note that the scheduling algorithms of HLS tools can be viewed as sequences of RTS transformations.

### 3.4 Register Transfer Merge (RTM)

RTM is the inverse of RTS: it merges two successive register transfers when the values are passed between these two transfers through a set of intermediate registers (see Figure 3.3).

**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 `rt1/rt2` by one register transfer obtained by eliminating the intermediate registers and replacing them with simple connections in an orderly manner.

**Preconditions for RTM:**
1. The set of output registers of the first register transfer (`rt1`) should be a subset of the input set of the second register transfer (`rt2`), and
2. The sets of operators of the two register transfers should be disjoint.
Correctness Criteria:

\[ \text{RTM-Precondition}(rt_1, rt_2) \triangleq \]
\[ (\text{in_regs}(rt_2) \subseteq \text{out_regs}(rt_1)) \land (\text{operators}(rt_1) \cap \text{operators}(rt_2) = \emptyset) \]
\[ \text{RTM-Precondition}(rt_1, rt_2) \Rightarrow \text{comp_behavior}(cg) = \text{comp_behavior}(\text{RTM}(cg, rt_1, rt_2)) \]

In Figure 3.3, the set of output registers of \( rt_2 \) is \( \{T_1, T_2\} \), the set of input registers of \( rt_1 \) is also \( \{T_1, T_2\} \), the operators set of \( rt_1 \) is \( \text{OP2} \) and the operators set of \( rt_2 \) is \( \text{OP1} \); the above preconditions for RTM are satisfied.

### 3.5 Register Transfer Decompose (RTD)

RTD can be viewed as sequencing two tasks which are initially executed in parallel. In Figure 3.4 OP2 is deferred to the next control step.

**Figure 3.4: The RTD and RTC Transformations**

**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:** It decomposes \( rt \) into 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.
Preconditions for RTD:
1. The sets of output and input registers of \( rt \) are disjoint.

Correctness Criteria:
\[
RTD\_Precondition(rt) \triangleq in\_regs(rt) \cap out\_regs(rt) = \emptyset.
\]
\[
RTD\_Precondition(rt) \Rightarrow \text{comp}\_behavior(cg) = \text{comp}\_behavior(RTD(cg, rt)).
\]

Figure 3.4 shows one register transfer (\( rt \)) and its decomposition in \( rt1 \) and \( rt2 \) corresponding to an RTD transformation where register \( R \) is \( R5 \).

### 3.6 Register Transfer Compose (RTC)

RTC is the inverse of RTD, it results in combing two successive register transfers which can be executed in parallel (in Figure 3.4, 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:** combine \( rt1 \) and \( rt2 \) into one register transfer \( rt \) such that all computations performed initially in \( rt1 \) and \( rt2 \) are now performed in one register transfer, \( rt \).

**Preconditions for RTC:**
1. The sets of output registers of \( rt1 \) and \( rt2 \) are disjoint,
2. The set of input registers of \( rt2 \) and the set of output registers of \( rt1 \) are disjoint.
3. The sets of operators of \( rt1 \) and \( rt2 \) are disjoint.

**Correctness Criteria:**
\[
RTC\_Precondition(rt1, rt2) \triangleq
(out\_regs(rt1) \cap out\_regs(rt2) = \emptyset) \land (out\_regs(rt1) \cap in\_regs(rt2) = \emptyset) \\
\land (operators(rt1) \cap operators(rt2) = \emptyset).
\]
\[
RTC\_Precondition(rt1, rt2) \Rightarrow \text{comp}\_behavior(cg) = \text{comp}\_behavior(RTC(cg, rt1, rt2)).
\]
3.7 Register Instance Substitution (RIS)

RIS is illustrated in Figure 3.5. 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 $R3$.

![Figure 3.5: The RIS Transformation](image)

**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 $Subst$.

**Preconditions for RIS:**
1. The life spans of $Subst$ and $Wth$ do not overlap.

**Correctness Criteria:**
\[ RIS_{-Precondition}(cg, rt, Subst, Wth) \triangleq \forall (rt \in cg) : not\_alive?(Subst, rt) \lor not\_alive?(Wth, rt). \]
\[ RIS_{-Precondition}(cg, rt, Subst, Wth) \Rightarrow \text{comp\_behavior}(cg) = \text{comp\_behavior}(RIS(cg, rt, Subst, Wth)) \]
Let $<$ be an order on register transfers of a control graph $cg$ such that for any two register transfers $rt_1$ and $rt_2$ of $cg$, $rt_1 < rt_2$ if and only if $rt_1$ occurs after $rt_2$ in $cg$, and $rt_1 \leq rt_2$ if and only if $rt_1$ occurs after $rt_2$ or is equal to $rt_2$. Then we define register $R$ not to be alive in the register transfer $rt$ as follows:

$$\text{not\_alive}(R, rt) \triangleq$$

\hspace{1cm} a)\forall(rt_1 \leq rt): (R \in \text{in}\_\text{regs}(rt_1)) \Rightarrow (\exists rt_2: rt_2 \leq rt \land rt_1 < rt_2 \land R \in \text{out}\_\text{regs}(rt_2)) \land

\hspace{1cm} b)\forall(rt_1): rt < rt_1 \land (R \in \text{out}\_\text{regs}(rt_1)) \Rightarrow (\exists rt_2 < rt_1: rt < rt_2 \land R \in \text{in}\_\text{regs}(rt_2))

In the above definition, a register $R$ is said to be $\text{not\_alive}$? 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 in $R$ 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 $rt$.

Figure 3.6: A Sequence of RIS Transformations

Figure 3.6 shows the repeated application of RIS transformations that results in a register instance substitution as defined by Vemuri [36]: an output register $R$ is substituted, so all the register instances that read a value from $R$ in the following control steps (before the register is written again), are also substituted. In our version of RIS, an input register is substituted and a sequence of such substitutions has the same effect as the register substitution defined by Vemuri [36]. In
Figure 3.6, $R3$ is substituted first as an input in the register transfer $rt2$, than in $rt2$.

In the transformed control graph, $R3$ is first written in the register transfer $rt4$ and then again in $rt1$, without being previously read. This useless writes are forbidden in the models used by [36] since they unnecessarily complicate the definitions for life-spans and some of the transformations’ preconditions. But in real designs, this case is nor illegal, nor unusual. Because of our step-wise definition of RIS, the useless writes may appear in the designs as shown in Figure 3.6. To overcome this drawback, we introduced a new transformation called *clean outputs* that deletes the unused output registers.

### 3.8 Clean Outputs (CO)

The CO transformation removes the appearance of registers from a set $Rset$ as outputs of a register transfer $rt$. It preserves the computational behavior of the design if every register $R_i$ in $Rset$ is subsequently written before being read, or if $R_i$ reads a value from itself at the control step defined by $rt$.

**Inputs of CO:**
1. A register transfer $rt$ of a control graph $cg$ to which CO is locally applied.
2. A set $Rset$ of registers that appear as outputs in $rt$.

**Function of CO:** Removes every register in $Rset$ from the set of output registers of $rt$.

**Preconditions for CO:** For every register $R_i$ in $Rset$
1. if $R_i$ appears as an input in a register transfer $rt1$ which is executed after $rt$, then $R_i$ also appears as an output in a register $rt2$ which is executed after $rt$, but before $rt2$, or
2. at the control step defined by $rt$, $R_i$ reads a value from itself.

**Correctness Criteria:**

\[
CO_{\text{Precondition}}(rt, Rset) \triangleq \\
\forall (r \in Rset) : (\forall (rt1 < rt) : r \in \text{in_regs}(rt1) \Rightarrow \exists (rt2 < rt) : rt2 > rt1 \land (r \in \text{out_regs}(rt2))) \lor \\
(f_{\text{reg}}(rt))(r) = r
\]

where $f_{\text{reg}}(rt)$ is a function returning the component connected to output register $r$ in $rt$.

\[
CO_{\text{Precondition}}(rt, Rset) \Rightarrow \text{comp}_\text{behavior}(cg) = \text{comp}_\text{behavior}(CO(cg, rt, Rset))
\]

In Figure 3.7, $Rset = \{R5\}$. The application of CO to register transfer $rt2$ and $Rset$ is correct since $R5$ is written again in $rt1$ before being read.
3.9 Add Outputs (AO)

The AO transformation adds a set of registers $R_{set}$ to the set of output registers of a transfer $rt$, and defines their interconnection with components of $rt$. The computational behavior of the design is preserved if every register $R_i$ in $R_{set}$ is subsequently written before being read, or if $R_i$ is added such that it reads a value from itself at the control step defined by $rt$. In addition, $R_{set}$ and the set of output registers of $rt$ should be disjoint.

**Inputs of AO:**
1. A register transfer $rt$ of a control graph $cg$ to which AO is locally applied.
2. A set $R_{set}$ of registers that appear as outputs in $rt$.
3. A function $F : R_{set} \rightarrow \text{comp}(rt)$ that assigns to each register in $R_{set}$ a component in $rt$, where $\text{comp}(rt)$ denotes the set of all expression trees contained in $rt$).

**Function of AO:** Add registers in $R_{set}$ to the set of output registers of $rt$, and defines their interconnection according to the function $F$.

**Preconditions for AO:**
For every register $R_i$ in $R_{set}$
1. $R_{set}$ and the set of output registers of $rt$ are disjoint, and 2. if $R_i$ appears as an input in a
register transfer \texttt{rt1} which is executed after \texttt{rt}, then \(R_i\) also appears as an output in a register \texttt{rt2} which is executed after \texttt{rt}, but before \texttt{rt2}, or

3. at the control step defined by \texttt{rt}, \(R_i\) reads a value from itself.

**Correctness Criteria:**

\[
\text{AO Precondition}(rt, Rset) := \\
\quad (Rset \cap \text{out regs} = \emptyset) \land \\
\quad (\forall (r \in Rset) : (\forall (rt1 < rt) : r \in \text{in regs}(rt1) \Rightarrow \exists (rt2 < rt) : rt2 > rt1 \land (r \in \text{out regs}(rt2))) \lor F(r) = r) \\
\]

\[
\text{AO Precondition}(rt, Rset, F) \Rightarrow \text{comp behavior}(cg) = \text{comp behavior}(\text{AO}(cg, rt, Rset, F))
\]

In Figure 3.8, \(Rset = \{R5,R2\}\) and \(F(R5) = \text{OP1}, F(R2) = R2\). The application of AO to register transfer \texttt{rt2} and \(Rset\) is correct since \(R5\) is written again in \texttt{rt1} before being read, and \(R2\) reads a value from itself.

![Figure 3.8: The AO Transformation](image_url)
3.10 Delete Register Transfer (DRT)

DRT removes a delay node from a control graph. A delay node corresponds to a control step at which no computation is performed. In our model we interpret a delay node as an *empty* register transfer, that is a register transfer with no output registers.

**Inputs of DRT:** 1. A register transfer rt of a control graph cg to which DRT is locally applied.

**Function of DRT:** Removes rt from the control graph cg.

**Preconditions for DRT:** rt is an empty register transfer.

**Correctness Criteria:**

\[ DRT_{\text{Precondition}}(rt) \triangleq \text{outregs}(rt) = \emptyset \]

\[ DRT_{\text{Precondition}}(rt) \implies \text{comp behavior}(cg) = \text{comp behavior}(DRT(cg, rt)) \]

![Figure 3.9: The DRT Transformation](image)

The OC, RTD and RIS transformations presented here are more detailed than those of Vemuri [36] or Camposano [44]. For example, the OC transformation of Vemuri creates multiple copies of an operator at the same time, while our OC transformation performs only one copy at a time. The RTD transformation in [36] and the cutting transformation in [44] consider a set of output registers at a time, while the RTD transformation presented here is executed for each output register. The more detailed definitions we used are due to the definitional approach adopted as an alternative to an error-prone axiomatic definition. If a set of operators or registers was to be considered, than the definitions for each of the above transformations should have been expressed as functions applied recursively on the elements of this set (in a definitional approach), or their outcome could have been globally stated through an axiom (in an axiomatic approach).
3.11  Summary

In this Chapter we informally presented each RTL transformation. We described their inputs, function, preconditions and the correctness criteria. Similarities and differences between our set of RTL transformations and other transformations found in literature were presented. As before, in the formal specifications of these transformations we assign to each informal notion a PVS type that best describes its meaning: inputs are elements of a defined type, functions are PVS interpreted function types, preconditions are predicate functions and the correctness criteria are expressed as theorems and were proved for each transformation. In the following chapter we present the specification methodology, the formal specifications, and proofs of correctness for each transformation.
Chapter 4

Formal Specifications in PVS for RTL Transformations

4.1 Specification Methodology

In specifying each transformation we considered three issues meant to improve the exactness and readability of the specifications, and ease the proofs:

1. the unambiguous definition of each transformation according to the informal descriptions presented in Chapter 3,

2. a uniform specification approach to make proofs easier and reusable,

3. make use of the interplay between specification and proofs to weaken preconditions.

Unambiguous Definitions. There are two approaches to define a transformation: a) a definitional approach, where a function that recursively updates each expression in the affected register transfer(s) is defined, and b) an axiomatic approach where the updating function is not explicitly defined, but its outcome is stated by an axiom. The former is usually tedious as it involves recursive definitions, the latter can introduce ambiguities since there is no way to check for an axiomatic definition more than the integrity of the types used. In our case, the definitional approach is preferable because it is not only safer but it also allows shorter definitions. Consider an axiomatic definition for the Operator Copy (OC) transformation; it should state that:

1) the set of operators of the affected register transfer contains one more operator, namely the copy operator;

2) the copy operator has the functionality and same sources as the copied operator;
3) the operator that reads in the result of the copied operator (corresponding to the specified link),
will read the result from the copy operator;
4) if the result of the copied operator is initially read in by an output register (through the specified
link), then that output register will read the result from the copy operator;
5) if some operator’s inputs are both connected to the copied operator then decide which input will
read from the copy;
6) the rest of the operators’ connectivity remains the same.

As it will be seen in the next section, by using a definitional approach all the above six properties
are expressed using one recursive definition with only three nested conditionals, and a lambda
expression (to define outputs’ connectivity). Thus, the above axiom becomes provable instead of
being simply stated. Note that in defining transformations that affect a limited portion of a struc-
ture (like RTL transformations), a mechanism that automatically implies the conservation of the
unaffected parts would be useful. Such mechanism is available in other specification languages, like
the Larch Language [46] with its modifies clause. In some sense, the definitional approach is closer
to this goal since the conservation of the unaffected parts of the design is implicit in the functions’
definition while in an axiomatic approach it should be stated explicitly.

**Uniform Specification.** The PVS specification of each transformation follows a common approach
summarized in the following steps:

1. Define a function that recursively updates the expression trees of the modified register transfer(s),
according to the effect of the transformation applied;
2. Update each field of the register transfer(s) involved in the transformation;
3. Bring the locally defined transformation in the context of the entire control graph.

Using this uniform specification we were able to separate the recursive behavior extraction along
the operator trees, which is typical to each transformation, from the recursive traversal of the con-
trol graph list, which is done similarly for all transformations. A set of general lemmas that assert
the global correctness of a transformation *when the extracted behavior is locally preserved* was for-
mulated and proved once and for all. Then, for each transformation the local “perturbation” was
proved to *locally* preserve the computational behavior, provided a specific set of preconditions is
satisfied. In the next sections we will present in more detail the proof methodology used for individ-
ual transformations and give here an outline of the general lemmas dealing with the incorporation
of a generic local transformation in the control graph list.
An RTL transformation replaces a portion of the graph list with a modified sublist and leaves the rest of the graph list unchanged. In Figure 4.1, \( cg1 \) and \( cg2 \) can be considered to represent the affected portion of the control graph before and after the transformation, respectively.

The \( cgs \) portion of a control graph is the sequence of register transfers executed after \( cg1 \) or \( cg2 \), and \( cge \) is the sequence of register transfers executed before \( cg1 \) or \( cg2 \). The general lemma states that: if all registers that appear in \( cgs \) have the same extracted behavior in \( cg1 \) and \( cg2 \), then all registers of \( cgs \) have the same extracted behavior in control graphs \( CG1 \) and \( CG2 \) (see Figure 4.1).

This is expressed by the following general lemma:
The \texttt{eb_cg CG cg_reg} lemma was proved from the two subgoals corresponding to Figures 4.1.a and 4.1.b using appropriate instantiations. We will state here only some considerations which guided our proofs, and refer the reader to Appendix B for a more detailed description of the proof strategy used.

The proofs involve recursion on both operator trees and control graph lists and are considerably longer and more difficult than the proofs of the local behavior correctness generated for each transformation. In proving the two sub-lemmas we used \textit{strong induction} on the length of the control graphs. That is because the behavioral extraction along control graphs is not a smooth traversal of successive register transfers. The extraction starts at the root of an operator tree and, when it reaches a register leaf, it \textit{jumps} at that register transfer where that register expression was last written. In Figure 4.2, the behavior extraction of expression $e$ starts at the operator tree rooted at $OP1$. Upon reaching the leaf register expressions $A$ and $B$, the \texttt{extracted_behavior} function (presented in Chapter 2) searches the control graph for the last register transfers where these registers appear as outputs, and resumes the extraction from their operator sources. The sublists have now lengths 1 and 2, respectively.

It was then necessary to use strong induction which states that if a property holds for all control graphs with length less than $n$, then it also holds for graphs of length $n$. The strong induction theorem is available in a PVS library called \textit{Prelude} and was applied automatically by instantiating...
it with the control graph list type and the well-founded subterm relation for lists, also predefined in PVS. The induction prover command in PVS, \texttt{INDUCT}, is applied quasi-automatically, with minimum interaction from the user. It selects an induction scheme according to the type of the induction variable and uses the formula specified in the command line to formulate an induction predicate, then simplifies yielding base and induction cases. The induction scheme can be explicitly supplied as an optional argument; for example, the simple induction on lists or operator trees is applied automatically, while the strong induction scheme should be specified and appropriately instantiated by the user.

The uniform specification of transformations allowed us to reduce the proof effort by reusing for several transformations some proofs interactively performed in the PVS’s proof checker for one transformation. In the PVS environment it is possible to automatically run a proof script; also, the proofs performed interactively in the proof checker environment are saved in a readable script file that can be edited.

For each individual transformation we proved two lemmas and the correctness theorem. The two lemmas refer to the well-formedness of the transformed register transfer(s) (as defined in Section 2.3.1), and to the local correctness of the transformation. The local correctness refers to a set of properties about the affected register transfer that have to hold in order to ensure the behavior preserving correctness of the control graph’s output registers. The strategy used in proving these lemmas and the Correctness Theorem is explained in Appendix C for one transformation, (OC). For all other transformations, except RIS, the proof strategy is similar. The details of proving correctness for RIS are explained in Appendix C.4.

\textbf{The interplay between specification and proofs.} The behavior preserving correctness of each transformation was proved provided a set of preconditions is satisfied. This is expressed by the following \textit{Correctness Theorem}:

\begin{align*}
T_{Preconditions}(cg, rt, set[expression]) \Rightarrow \\
& \text{computational behavior}(T(cg, rt, set[expression])) = \text{computational behavior}(cg)
\end{align*}

Finding weaker preconditions was important for two reasons: a) a synthesis system that implements these transformations is more time-efficient if it has to check for fewer preconditions, and b) a software implementation debugging tool based on \textit{Witness Generators} [24] is more accurate if the witness sequence records the violation of a weaker precondition than an unnecessarily stronger one. Finding the \textit{weakest} precondition is not possible; consider for example, the commutativity property of an operation, which can be sometimes allowed and sometimes not. It can not be decided just by analyzing the structural information if commutativity preserves behavior or not, so enforcing
as a precondition that commutativity should not be allowed (as it is implicitly in our models) is sometimes a stronger precondition than needed. However, it was possible to look for weaker preconditions by iteratively inserting preconditions in the antecedent until the behavior preserving correctness proof was derived. PVS’s logical system is based on sequent calculus, consisting of an antecedent, as a conjunction of formulas, and a consequent, as a disjunction of formulas. The set of preconditions appears as a conjunction of formulas in the antecedent. We started by attempting to prove the behavior correctness with only those preconditions that enforced the well-formedness of the transformed design. Upon reaching a subgoal that could not be proved, we conjunctively added to the antecedent a precondition that is necessary to discharge that subgoal. The proof environment is exited and the specification is enhanced with the new precondition. We run again the proof already performed (but starting from a stronger antecedent) and, if a proof is not derived, insert a new precondition.

In the following sections we present and explain the PVS specifications for each elementary RTL transformation and discuss on the proofs of correctness.

### 4.2 Operator Copy (OC)

The OC transformation is locally applied to a register transfer of a control graph. The new register transfer contains one more operator having the same sources as the copied operator. Since we conveniently model the operators, input registers and their interconnection as binary tree structures, the insertion of the new operator is in fact the insertion of an *operator tree expression* defined by the following function:

```plaintext
op_copy_exp(copied: (op?), new_op: operator) : expression =
  op(new_op, source1(copied), source2(copied))
```

OC receives as inputs a *link*, defined as a tuple of the form \([\text{expression}, \text{side}, \text{expression}]\), where the first and the third projections represent the driven and the source instance respectively. The *side* type is an enumeration type representing the port of the driven instance. The *new_op* input variable of type *operator* is the copy operator instance. The operator to be copied is the root of the expression tree \(1^3\); it is efficient to define as inputs accepted by the OC transformation those links whose third projections are operator expressions \((\text{op}(1^3))\) and not register expressions \((\text{not reg}(1^2))\). The sort of links acceptable as inputs for the OC transformation are defined below by the predicate subtype \(\text{op}_\text{link}\).
The \texttt{new\_exp} function in Figure 4.3 defines, for each expression in the register transfer, the new expression tree which accommodates the OC transformation. The recursive nature of the expression trees dictates the recursive updating to be performed by \texttt{new\_exp}. Three nested if-conditionals check for the base cases: a register expression, or an eventual operator connected to the copied instance through the link \( l \), as the first or second source. Note that the first projection of \( l \) is not necessarily an operator, it could be an output register (like the \([O1, OP2]\) link in Figure 4.1) in which case the effect of OC is completed by updating the \texttt{regassign} function. In the recursive step, \texttt{new\_exp} descends on the two direct ancestors in the operator tree. The termination of the recursive definition is ensured by the measure function \texttt{reduce\_nat} supplied in the abstract DATATYPE theory generated upon type-checking the \texttt{expression} THEORY. A function that assigns a natural number to a binary tree gives a measure of the size of an expression which decreases when descending the tree.

To complete the OC transformation definition, each field of the transformed register transfer is updated:

1. The \texttt{exp\_set} field of the new register transfer is the set of the new (updated) expression trees, plus the new expression rooted at the copy operator and defined previously by the \texttt{op\_copy\_exp} function. The \texttt{new\_exp\_set} function defines axiomatically the set of the new expressions; in a purely definitional approach, the new set would be constructed by recursively adding, to an initially empty set, updated expressions of each old ones;

2. The set of output registers (\texttt{outregs}) remains unchanged;

3. The writing of output values (defined by \texttt{regassign}) is updated such that: if an output regis-

\begin{figure}[h]
\centering
\includegraphics[width=0.8\textwidth]{figure4_3.png}
\caption{The Effect of the OC Transformation on the Expression Trees}
\end{figure}
The \texttt{OC} function defines the effect of the locally applied transformation to the entire control graph. It merely replaces the register transfer at position \texttt{n} with the transformed one, \texttt{oc(nth(cg,n),new_op,l)} (We identify a register transfer in a control graph \texttt{cg} by its position).

\begin{verbatim}
OC(cg:list[transfer],n:below[length(cg)],new_op:operator,l:op_link) : list[transfer] =
  replace(cg,n,oc(nth(cg,n),new_op,l))
\end{verbatim}

The OC precondition states that: a) the input link \texttt{l} does exist as a connection within the register transfer \texttt{rt}, that is, it belongs to the set of links of \texttt{rt} defined by the function \texttt{get_rt_links}; b) the copy and the copied operators have the same functionality; c) the copy operator, \texttt{new_op} is not already used within \texttt{rt}.

\begin{verbatim}
oc_precondition(l:op_link, new_op:operator,rt:transfer) : bool =
  get_rt_links(rt)(l) and opfn(new_op) = opfn(op(l'2)) and not operators(rt)(new_op)
\end{verbatim}

In the following PVS specification, the first lemma (\texttt{wellformed_oc}) refers to the well-formedness of the transformed register transfer. The well-formedness is also conditioned by a set of preconditions, usually weaker than those required for behavior preserving correctness. For OC, the copied and
the copy operator might not have the same functionality but the well-formedness is still preserved. The second lemma, \texttt{oc\_preserve\_behavior} asserts the local correctness of the transformation, that is, it states that every register have the same extracted behavior in the initial and in the transformed register transfer. Both lemmas were proved using induction on the expression trees, case analysis, expressions and behavior expressions extensionality axioms and, of course, preconditions. A description of the proof strategy is presented in Appendix C.

In proving the correctness theorem, \texttt{OC\_pb}, we used the above two lemmas along with the general lemma presented in the previous section. The proof was derived only by simple rewritings, since the recursive traversal of control graph lists for extracting registers’ behavior was dealt with by the general lemmas.

The correctness theorem states that every output register has the same extracted behavior in the initial and in the transformed control graph. The set of output registers should be seen as a predefined set, in the sense that, at the end of the last control step, only registers from this set are read.

An initial RTL implementation of a given behavior (as suggested in Section 2.2) can preserve the naming of its registers corresponding to the variable names used in the behavior description. An optimized implementation may contain some temporary registers that have no correspondent in the behavior description. However, from the point of view of the structural representation of a design, there is no distinction between the two kinds of registers. The design representation of Vemuri makes
use of a classification of registers as *architectural* - those coming from the behavior description, or *temporary* - those introduced by a synthesis tool. We argue that such a classification unnecessarily complicates the models; the behavior preserving correctness of a transformation is defined with respect to the RTL design to which it is applied, not to the behavior description implemented by that design. Defining a register as *temporary* would require a reference to the behavior input, or to an initial implementation of that behavior input, from which the set of architectural registers can be determined. Following a similar argument, we prefer not to define the output registers as corresponding to output variables of a behavior description, but as those registers that appear as outputs (belong to the `outregs` field) in the last register transfer. The restriction imposed by this definition on the RTL models is not essential since it does not restrict the *class of designs* representable using these models.

### 4.3 Operator Instance Substitution (OIS)

The OIS transformation locally applied to a register transfer receives as inputs an operator expression (`subst` in the specification below) and an operator instance which will substitute the (eventual) appearance of `subst` in that register transfer. The specification for OIS is similar to that of OC: the recursive function `new_exp` defines the new connectivity of operators and input registers according to the effect of OIS, the `ois` function specifies each field of the affected register transfer, and the `OIS` function defines the OIS transformation as `ois` applied to the n'th register transfer.

**Figure 4.4: The Effect of the OIS Transformation on the Expression Trees**

```
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 reduce_nat ((LAMBDA (r:register) :0) ,
    (LAMBDA (op:operator), (n1:nat), (n2:nat) : n1+n2+1))(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))
```
In Figure 4.4 subst corresponds to the operator expression rooted at OP3 and opr to the operator instance OP2. In the transformed register transfer, OP3 and the links represented with a thin line will disappear, and a new link (with dotted line in the Figure) is created. According to the above definition of new_exp, the new expression corresponding to that rooted at OP1 will be rooted at the same operator instance, its right source remains the same (the base case when reg?(e) is true), and its left source is the operator expression rooted at OP2 (the base case when e = subst).

We proved the behavior preserving correctness when the OIS precondition holds. The precondition_ois? predicate below is true when opr and the root operator of subst have the same functionality, and, if opr appears in the register transfer rt, then the expression tree rooted at opr has the same sources as the root of subst. When opr does not appear in rt, then the identical functionalities suffices.

The proofs for well-formedness and correctness are similar to those described in Appendix C for the OC transformation. The difference consists in a different case analysis for the particular cases of the new_exp function.

```plaintext
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))
```

4.4 Register Transfer Split (RTS)

RTS is the only transformation for which an axiomatic definition would have been shorter and clearer. The more cumbersome definitional specification is due to the difficult definition of the temporary registers that save intermediate results (as described in Section 3.3). Informally, the temporary registers have to be new (not already used in a certain control graph), and distinct from each other. To formally define such properties about a register expression we introduced the following two functions:
• A **new_reg** function that returns for each finite set of registers, \( rset \), a register which does not belong to it (\( \text{new_reg}(rset) \) is **“new to \( rset \)”**).

\[
\text{new_reg}(\text{finite_set: set}[(\text{reg}?)]) : (\text{reg}?) = \text{choose}\{(r: (\text{reg}?)) \mid \text{not } rset(r)\}
\]

• An **insert_new_reg** function which assigns to each expression belonging to a finite set \( eset \) a new register expression.

\[
\text{insert_reg}(eset: \text{finite_set}[\text{expression}], rset: \text{finite_set}[(\text{reg}?)]) : \text{RECURSIVE} [(eset)\rightarrow(\text{reg}?)] = \\
\text{if empty}(eset) \text{ then LAMBDA } (x:\text{eset}) : \epsilon(\text{fullset}[(\text{reg}?)]) \text{ else let } e: \text{expression} = \text{choose}(eset), \text{new}_r: (\text{reg}?) = \text{new_reg}(rset) \text{ in} \\\n\text{LAMBDA } (x:\text{eset}) : \text{if } x = e \text{ then new}_r \text{ else insert_new_reg(rest(lset),add(new}_r,rset))(x) \text{ endif endif}
\]

\[
\text{MEASURE card}(eset)
\]

The **insert_new_reg** function is used to define the temporary registers with the desired properties: **new** and **distinct from each other**. In the following specification, the **insert_new_reg_lemma** states that for every set of expressions \( eset \) and every set of (existing) registers, \( rset \), every expression of \( eset \) is mapped through **insert_new_reg** to a new register. The function **new_regs** defines the set of new registers as the range of the **insert_new_reg** function applied to \( eset \), and the **map_reg_link** function is **insert_reg** partially defined for the range restricted to the new registers. The **bijective_map_reg_link** lemma states that the **insert_new_reg** function performs a one-to-one mapping from expressions to new registers.

\[
\text{insert_new_reg_lemma : LEMMA} \quad \forall (eset: \text{finite_set}[\text{expression}], rset: \text{finite_set}[(\text{reg}?)], e: (\text{eset})): \quad !rset(\text{insert_reg}(eset, rset)(e))
\]

\[
\text{new_regs}(eset: \text{finite_set}[\text{expression}], rset: \text{finite_set}[(\text{reg}?)]): \text{set}[(\text{reg}?)] = \text{image}(\text{insert_new_reg}(eset, rset))(eset)
\]

\[
\text{map_reg_link}(eset: \text{finite_set}[\text{expression}], rset: \text{finite_set}[(\text{reg}?)]): [[\text{eset}\rightarrow\text{new_regs}(eset, rset)]] = \\
\text{LAMBDA } (x: (\text{eset})): \text{insert_new_reg}(eset, rset)(x)
\]

\[
\text{bijective_map_reg_link : LEMMA} \quad \forall (eset: \text{finite_set}[\text{expression}], rset: \text{finite_set}[(\text{reg}?)]): \text{bijective?}(\text{map_reg_link}(eset, rset))
\]
With this definition, the notion of temporary registers insertion can be modeled as applying the \texttt{insert\_new\_reg} function to the finite set of expression that appear in the affected register transfer \((\text{eset} = \text{rt}'\text{exp}\_\text{set})\), and to the set of all registers that appear in the control graph \((\text{rset} = \text{registers}(\text{cg}))\).

In the following recursive definition of \texttt{new\_exp}, the operator expressions are updated according to the insertion of the new registers: if \(e\) is a register or it belongs to the \texttt{split\_set} then it remains unchanged; if it correspond to a deferred operation and any of its sources is an operator expression \(\text{ope}\) in \texttt{split\_set}, then a temporary register is inserted as the \texttt{insert\_new\_reg} function to \(\text{ope}\) and the set of existing register instances, \texttt{rset}.

```plaintext
new_exp(rt:wellformed_rt, e:(rt'exp_set), split_set: set[expression], rset: finite_set[(reg?)]) : RECURSIVE expression =
let eset : finite_set[expression] = rt'exp in
if (split_set(e) or reg?(e)) then e
else
  if split_set(source1(e)) and split_set(source2(e))
    then op(op(e), insert_new_reg(eset,rset)(e,source1(e)), insert_new_reg(eset,rset)(e,source2(e)))
  else
    if split_set(source1(e))
      then op(op(e), insert_new_reg(eset,rset)(e,source1(e)), new_exp(rt,source2(e), split_set,rset))
    else
      if split_set(source2(e))
        then op(op(e), new_exp(rt,source1(e), split_set,rset), insert_new_reg(eset,rset)(e,source2(e)))
      else
        op(op(e), new_exp(rt,source1(e),split_set,rset), new_exp(rt,source2(e), split_set, rset))
  endif
endif
endif
MEASURE reduce_nat ((LAMBDA (r:register) :0) ,
(LAMBDA (op:operator), (n1:nat), (n2:nat) : n1+n2+1))(e)
```

In Figure 4.5 the set of expressions to be scheduled in the current control step, \texttt{split\_set} in the PVS definitions, contains the operator expression rooted at \texttt{OP5} and its ancestor register expressions: \texttt{I2} and \texttt{I3}. The temporary register \texttt{T1} is defined by evaluating the \texttt{insert\_new\_reg} function at operator expression rooted at \texttt{OP5}.

Two PVS functions specify the two register transfers resulted after split: \texttt{first\_rts} defines the register transfer executed after the current step, and \texttt{second\_rts} the one executed in the current step. (Throughout the specifications we preferred to count the register transfers in the behavior extraction direction, opposite to the data-flow direction. This way we avoided the behavior extraction on a reverse list of transfers, which would have complicated the proofs.) The \texttt{exp\_set} field of the
first_rts contains the updated expressions deferred to the next control step, and the temporary registers inserted (split_regs), as input registers. The set of output registers remains the same, but the regassign function is updated such that an output register that used to read from an expression in the split_set will read now the temporary value stored in a temporary register.

```plaintext
first_rts(rt:wellformed_rt, split_set: finite_set[expression], rset: finite_set[(reg?)]): transfer =
  let eset : finite_set[expression] = rt\'exp_set in
  (# exp_set:= union(union(new_exp_set(rt,difference(rt\'exp_set,split_set),
    split_set,rset),split_regs(rt,split_set,rset)),inputs(rt)),
    outregs:= rt\'outregs,
    regassign:= LAMBDA (x:(reg?):
      if (rt\'outregs(x)) then
        if (split_set(rt\'regassign(x)) and op?(rt\'regassign(x)))
          then insert_new_reg(eset,rset)(rt\'regassign(x))
        else new_exp(rt,(rt\'regassign)(x), split_set,rset)
          endif else rt\'regassign(x) endif #)
```

The expressions set of the second register transfer consists of the split_set. Its output registers set consists only of temporary registers. The regassign function recovers the split links, which is possible since the insert_new_reg function that created the output registers is a bijective function. rts is then defined as the list of the two register transfers.

```plaintext
second_rts(rt:wellformed_rt, split_set: finite_set[expression], rset: set[(reg?)]) : transfer =
  let eset: finite_set[link] = rt\'exp_set in
  (# exp_set:= split_set, outregs:= split_regs(rt, split_set,rset),
    regassign:= LAMBDA (x:(reg?):): if split_regs(rt,split_set,rset)(x) then
      inverse(insert_reg(lset,rset))(x)\'2 else x endif #)

rts(rt:wellformed_rt, split_set: finite_set[expression], rset: set[(reg?)]) : list[transfer] =
  cons(first_rts(rt, split_set, rset), cons(second_rts(rt, split_set, rset), null))
```

The replace function used to define the “global” RTS is similar to that used for OC and OIS, just that it replaces a register transfer with a tuple of register transfers. The overloading capability supported by the PVS language allowed this common specification.
The RTS precondition states that every operator scheduled in the current control step has its sources also scheduled in this step. This is similar to the definition of a wellformed set of expressions: each expression in a well-formed set has its ancestors in the same set. In other words, the set of expressions selected to be scheduled in the current control step should be a well-formed subset of the register transfer’s exp set field. The correctness theorem contains one more precondition: that the split set is not empty; this only ensures that an empty register transfer is not created.

The wellformedness for each register transfer generated by RTS, the local preserving correctness property and the Correctness Theorem for RTS was proved following the same strategy presented in Appendix C for OC.

4.5 Register Transfer Merge (RTM)

The RTM transformation is the inverse of RTS. Figure 4.6 illustrates the effect of RTM on the result of the RTS transformation applied to the register transfer showed in Figure 4.5. The new exp function updates the operator trees in the first register transfer such that those operators whose inputs were temporary registers, after the transformation will be connected to those expression of the second transfer which correspondingly write values in these temporary registers.

The extracted behavior of the first register transfer’s outputs is locally preserved if: a) the set of output registers of the second register transfer is a subset of the set of input registers of the first
new_exp(first, second: transfer, e: expression):
RECURSIVE expression =
if reg?(e) then
  if second\'outregs(e) then second\'regassign(e)
  else e
  endif
else op(op(e), new_exp(first, second, source1(e)),
  new_exp(first, second, source2(e)))
endif
MEASURE size(e)

rtm(first, second: transfer) : transfer =
(# exp_set := union(new_exp_set(first, second, first\'exp_set),
second\'exp_set),
outregs := first\'outregs,
regassign := lambda (x:(reg?)) :
if first\'outregs(x)
  then new_exp(first, second, first\'regassign(x))
  else x endif #)

rtm_precondition(first, second: wellformed_rt) : bool =
subset?(second\'outregs, inputs(first)) and empty?(intersection(operators(first), operators(second)))

rtm_preserves_behavior : LEMMA
forall (first, second: wellformed_rt, r: (first\'outregs)) :
  rtm_precondition(first, second) =>
    extracted_behavior(reg(r), cons(first, cons(second, null))) =
    extracted_behavior(reg(r), cons(rtm(first, second), null))

Note that the extracted behavior of the temporary registers is not preserved; for example, before
the transformation, the extracted behavior of \( T_1 \) is, in an informal notation,

\[ T_1 = I_2 \ (\text{opfn}(\text{OP5})) \ I_3. \]

After the transformation, the extracted behavior of \( T_1 \) in the combined register transfer is simply
the leaf value \( T_1 \). To preserve the extracted behavior of the \( \text{output registers of the design} \) it was
sufficient to constrain the \( \text{temporary} \) registers such that they should not appear in any register
transfer executed after the control step affected by the transformation. Here is, again, an example
of defining a register as \( \text{temporary} \) with respect to a set of registers which are in use in an RTL
representation, and not in the behavior description. In the following definition, \( \text{start} \_\text{cg} \) defines
the sub-list of the control graph \( cg \) consisting of register transfers executed after the control step \( n \). The enhanced precondition was sufficient to derive a proof for the correctness theorem \( \text{RTM}_{PB} \).

\[
\text{RTM}_{\text{precondition}}(cg: \text{cons_graph}, n: \text{below}[(\text{length}(cg)-1)], r: \text{(reg?)}) : \text{bool} = \\
\text{rtm}_{\text{precondition}}(\text{nth}(cg,n),\text{nth}(cg,n+1)) \text{ and} \\
\text{not registers}(\text{start}_{cg}(cg,n))(r)
\]

\( \text{RTM}_{PB} : \text{THEOREM} \)

\[
\forall (cg: \text{cons_graph}, n: \text{below}[(\text{length}(cg)-1)], r: \text{(outregs(car(cg))))} : \\
\text{RTM}_{\text{precondition}}(cg,n,r) \Rightarrow \\
\text{extracted}_{\text{behavior}}(\text{reg}(r),cg) = \text{extracted}_{\text{behavior}}(\text{reg}(r),\text{RTM}(cg,n))
\]

The well-formedness of the resulted transfer and the local correctness required the same precondition, \( \text{rtm}_{\text{precondition}} \). All proofs for RTM followed the same strategy as before.

### 4.6 Register Transfer Decompose (RTD)

The specification of the RTD transformation does not require a recursive definition for the new expressions, since all expressions remain the same, but are distributed in different register transfers. In Figure 4.6 the computation of the value to be written in the \( O2 \) output register is deferred to the next control step, and a new register transfer, \( \text{rtd\_first} \), is created. This implies that all operator expressions used to compute this value are present in the same register transfer. The \( \text{cone} \) function recursively creates this set of expressions. For a clearer specification we allow the decomposition corresponding to only one output register, thus the \( \text{outregs} \) field of \( \text{rtd\_first} \) contains only this register.

The function \( \text{rtd\_second} \) contains the remaining computations and it differs from the initial register transfer only in the set of output registers. With this definition, the operator expression rooted at \( OP2 \) is still present in \( \text{rtd\_second} \), but is not used. This does not affect the well-formedness of the resulted register transfers, nor the behavior preserving correctness of the transformation. As in the case of the unused writes (see Section 3.7), we allow these unused operators and supply a \( \text{clean\_operators} \) function similar to \( \text{clean\_outputs} \), which is applied only when these “extra” operators would cause a false precondition failure.

The only precondition necessary to ensure a correct application of RTD states that the input and output registers sets of the affected register transfer are disjoint:
components deffered to the next control step (rd_first)

\[
\text{rtd}_\text{first}(rt: \text{transfer}, r: (rt'outregs)) : \text{transfer} = \\
\begin{align*}
# \ exp\_set & := \text{cone}(rt'\text{regassign}(r)), \\
\text{outregs} & := \text{singleton}(r), \\
\text{regassign} & := rt'\text{regassign} #
\end{align*}
\]

\[
\text{rtd}_\text{second}(rt: \text{transfer}, r: (rt'outregs)) : \text{transfer} = \\
\begin{align*}
# \ exp\_set & := rt'\text{exp}\_set, \\
\text{outregs} & := \text{remove}(r, rt'\text{outregs}), \\
\text{regassign} & := rt'\text{regassign} #
\end{align*}
\]

\[
\text{rtd}(rt: \text{transfer}, r: (rt'outregs)) : \text{list}[\text{transfer}] = \\
\text{cons}(\text{rtd}_\text{first}(rt, r), \text{cons}(\text{rtd}_\text{second}(rt, r), \text{null}))
\]

Figure 4.7: The Effect of the RTD Transformation on the Expression Trees

Because of the non-inductive definition for RTD, the well-formedness proofs were much simpler, since they did not require induction on expression trees. The correctness proofs were similar to those performed for the previous transformations.

4.7 Register Transfer Compose (RTC)

Figure 4.7 shows the outcome of the RTD transformation applied to the register transfer in Figure 4.6. RTC performs the inverse of RTD: RTC applied to the two register transfers shown in Figure 4.7 bellow results in the combined transfer in Figure 4.6.

The RTC transformation is locally defined by the rtc function. The RTC precondition expressed by the \text{rtc.precondition} predicate is true if: a) the two register transfers to be composed have disjoint output register sets, b) any input register of the first register transfer does not appear as an output in the second register transfer, and c) the set of operator instances of the two transfers are disjoint.
The \textit{replace2} function in the definition of RTC replaces a register transfer by a sequence of two register transfers. The proofs for well-formedness and correctness were similar to those derived for RTD.

\begin{verbatim}
RTC(cg:cons_graph,n:below[length(cg)-1]) : list[transfer] =
    replace2(cg,n,rtc(nth(cg,n),nth(cg,n+1)))

RTC_PB: THEOREM
forall(cg:cons_graph,n:below[length(cg)-1],r:(outregs(car(cg)))) :
    rtc_precondition(nth(cg,n),nth(cg,n+1)) =>
        extracted_behavior(reg(r),cg) = extracted_behavior(reg(r),RTC(cg,n))
\end{verbatim}

\section{4.8 Register Instance Substitution (RIS)}

The RIS transformation substitutes the appearance of an input register in a certain register transfer with another register. In Figure 4.9 \textit{S} is substituted with \textit{W} in the \textit{n}’nth register transfer. To preserve the behavior correctness, \textit{W} is also added as an output register in that register transfer where \textit{S} was last written. \textit{S} also appears as an input in the \textit{n+2} register transfer, so the simple substitution of \textit{S} with \textit{W} in transfer \textit{n+3} would have affected the transformation correctness.
The `new_exp` function recursively updates the expression trees in the \(n\)’th register transfer according to the substitution performed. Function `ris_in` defines the changes performed in the register transfer to which the transformation is applied (corresponding to transfer \(n\) in the Figure), and `ris_out` defines the changes operated on that register transfer where the substituted register was last written (corresponding to transfer \(n+3\) in the Figure).

We deal separately with the case when the substituted register is a primary input. A register `subst` which appears as an input in the \(n\)’th register transfer of the control graph `cg` is a primary input if the following predicate is true: 

\[
\text{null?(match(subst,rest\_cg[transfer](cg,n)))}
\]

In the formula above, `rest\_cg(cg,n)` is the sublist of `cg` which contains transfers executed before the control step \(n\). The `match(r,cg)` function (as explained in Section 2.3.4) returns that sublist of `cg` whose first register transfer contains register `r` as an output (it `locates` that register transfer where `r` was last written).

In the Figure 4.9, the primary input `A` is substituted with a new register `A'`. The RIS transformation introduces a new register transfer, \(n+1\), which simply writes in `A'` the value of `A`. A function called `simple\_rt` takes as inputs the substituting and the substituted registers and returns a register
transfer whose only expression is the register expression corresponding to the substituted register; this expression is assigned to the only one element of the outregs set, the register expression corresponding to the substituting register. The following PVS definition for the RIS transformation captures the two cases: a) when the substituted register is a primary input, and b) when its value was previously written in a control step identified in the control sequence by the match_pos function.

\[
\text{RIS}(cg: \text{cons\_graph}, n: \text{below}\{(\text{length}(cg))\}, \text{subst}:(\text{inputs}(\text{nth}(cg,n))), \text{wth}:(\text{reg?})) : \text{c\_graph} = \\
\text{if null?}(\text{match}(\text{subst}, \text{rest}\_cg[\text{transfer}](cg,n))) \text{ then} \\
\text{replace}(cg,n, (\text{ris\_in}(\text{nth}(cg,n), \text{subst}, \text{wth}), \text{simple\_rt}(\text{wth}, \text{subst}))) \\
\text{else} \\
\text{let } n1 : \text{nat} = \text{match\_pos}(\text{rest}\_cg[\text{transfer}](cg,n), \text{subst}) \text{ in} \\
\text{replace}(\text{replace}(cg,n, (\text{ris\_in}(\text{nth}(cg,n), \text{subst}, \text{wth})), n+n1), \\
\text{ris\_out}(\text{nth}(cg,n+n1), \text{subst}, \text{wth})) \\
\text{endif}
\]

The RIS precondition in 3.7 states that the substituted and the substituting registers’ life spans do not overlap. During the proofs we were able to derive a weaker precondition, inspired by the fact that the life-span of a register in a control graph is a rather global property, along the entire control graph, while the RIS transformation is applied locally. Thus, for the more interesting case where the substituted register is not a primary input, the precondition used to derive the correctness proof was much weaker:

Let \( n1 \) be the position of that register transfer where the substitution is performed, and \( n2 \) the position of the control transfer where the substituted register was last written. The RIS precondition predicate is true if for any \( n \):

- if \( n1 < n \leq n2 \), the substituting register does not appear as an output register in the register transfer positioned at \( n \), and

- if \( n1 \leq n < n2 \), the substituting register does not appear as an input register in the register transfer positioned at \( n \)

When the substituted register is a primary input, then the substituting register should not be alive during the register transfer to which RIS is applied. The \text{not\_alive?} predicate was defined in PVS following its definition in Section 3.7.
The effect of the RIS transformation on a control graph is not as local as the other’s transformations. In proving the correctness of RIS we could not use the general lemmas, so the recursion on both operator trees and graph lists were necessary. A description of the proof strategy used for the RIS correctness is presented in Appendix C.4.

RIS_preserve_behavior: THEOREM
\[
\forall (cg: \text{cons}\_\text{graph}, n: \text{below}\[\text{length}(cg)], subst: (\text{inputs}(\text{nth}(cg,n))), wth: (\text{reg})): \\
\forall (r: (\text{car}(cg)'\text{outregs})) : \\
\text{ris_precondition}(cg, n, subst, wth) \Rightarrow \text{extracted_behavior}(\text{reg}(r), cg) = \\
\text{extracted_behavior}(\text{reg}(r), \text{RIS}(cg, n, subst, wth))
\]

4.9 Clean Outputs (CO) and Add Outputs (AO)

Clean Outputs (OC). The \text{clean\_outputs1}\_sr function in Figure 4.10 defines the OC transformation: the register transfer corresponding to the \(n\)'th control step in the control graph \(cg\) is replaced with a register transfer \text{new\_rt} whose \text{exp\_set} and \text{regassign} fields remain unchanged; any register expressions that appear in \text{rset} are removed from the \text{outregs} field. The \text{precondition clean} predicates is true if all registers that appear in the \text{outregs} field of the \(n\)'th (\text{nth}(cg,n)) register transfer, as well as in \text{rset} are subsequently written before being read, or if, within \text{nth}(cg,n) they read a value from themselves.

Registers \text{O1} and \text{O2} that appear as outputs in \text{rt2} in the figure satisfy the precondition, hence they can be removed by CO transformation: \text{O2} reads a value from itself, and \text{O1} is written again in register transfer \text{rt1} without being previously read.

The well-formedness of the transformed control graph is unconditioned. To prove correctness we used recursion on operator trees as well as on the length of the control graph. That is because
Figure 4.10: The Effect of CO and AO Transformations

the precondition for correctness is checked globally on the control graph. The proof strategy for correctness is outlined in Appendix C.5.

Add Outputs (AO). The AO transformation performs the inverse of OC; it adds a set \( rset \) of output registers and defines their connectivity with components of the affected register transfer. In the following PVS definition, \( \text{add_outputs}_{sr} \) specifies the AO transformation applied to the \( n \)’th register transfer of a control graph \( cg \). A finite set of registers, \( rset \) is added to the \( \text{outregs} \) field of \( \text{nth}(cg,n) \), and \( f: [(rset)->(\text{nth}(cg,n)\text{’exp_set})] \) defines the interconnections between registers in \( rset \) and expressions in \( \text{nth}(cg,n)\text{’exp_set} \).

```plaintext
wellformed_clean_outputs1_sr: LEMMA
forall(cg: c_graph, n: below [length(cg)], rset: set[(reg?)]) :
  wellformed_cg?(clean_outputs1_sr(cg,n,rset))

clean_outputs1_sr_PB: LEMMA
forall(cg: cons_graph, n: below [length(cg)], rset: finite_set[(reg?)]) :
  forall(r: output_registers(cg)) :
    precondition_clean(cg,n,rset) =>
    extracted_behavior(reg(r), cg) =
    extracted_behavior(reg(r), clean_outputs1_sr(cg,n,rset))
```

Add Outputs (AO). The AO transformation performs the inverse of OC: it adds a set \( rset \) of output registers and defines their connectivity with components of the affected register transfer. In the following PVS definition, \( \text{add_outputs}_{sr} \) specifies the AO transformation applied to the \( n \)'th register transfer of a control graph \( cg \). A finite set of registers, \( rset \) is added to the \( \text{outregs} \) field of \( \text{nth}(cg,n) \), and \( f: [(rset)->(\text{nth}(cg,n)\text{’exp_set})] \) defines the interconnections between registers in \( rset \) and expressions in \( \text{nth}(cg,n)\text{’exp_set} \).
The predicate `precondition_add` is true if registers in `rset` are not already used as output registers in the `n`th register transfer, and every register `r` in `rset` is subsequently written before being read or `r` is connected through `f` to itself. The correctness theorem (`add_outputs_sr`) was proved using induction on operator trees and on the length of the graph. The proof strategy is very similar to that used form proving the correctness of the CO transformation (as shown in Appendix C.5).

4.10 Delete Register Transfer (DRT)

The DRT transformation removes a delay node from a control graph. In our model, a delay node is defined as an `empty` register transfer, i.e., a register transfer with no output registers (no value is computed or transferred). In the following PVS definition, `delete_rt` models the DRT
transformation applied to the \( n \)’th element of the control graph list \( cg \) as the concatenation of the sublists containing the register transfers executed after and before the \( n \)’th transfer. The well-formedness is always preserved by the DRT transformation (in the \texttt{wellformed_delete}\_\texttt{rt} lemma).

\[
\text{delete}\_\texttt{rt}(cg:\texttt{list[transfer]}, n:\texttt{below[length(cg)]}) : \texttt{list[transfer]} = \\
\text{append}(\text{start}\_\texttt{cg}(cg,n), \text{rest}\_\texttt{cg}(cg,n))
\]

\[
\texttt{wellformed}\_\texttt{delete}\_\texttt{rt} : \texttt{LEMMA} \\
\forall cg:\texttt{c}\_\texttt{graph}, n:\texttt{below[length(cg)]} : \texttt{wellformed}\_\texttt{cg?}(\text{delete}\_\texttt{rt}(cg,n))
\]

The computational behavior of the designed is preserved by DRT if the deleted node is \textit{empty}. A proof for the correctness theorem was derived using appropriate instantiations of the general lemma presented at the beginning of this chapter, and using a local property which states that the value of any register transfer is not modified across the \( n \)th control step.

\[
\text{delete}\_\texttt{rt}\_\texttt{preserves}\_\texttt{behavior} : \texttt{THEOREM} \\
\forall cg:\texttt{cons}\_\texttt{graph}, n:\texttt{below[length(cg)]}, r:\texttt{register} : \\
\text{empty?}(\text{nth}(cg,n)) \Rightarrow \texttt{extracted}\_\texttt{behavior}(r,cg) = \texttt{extracted}\_\texttt{behavior}(r,\text{delete}\_\texttt{rt}(cg,n))
\]

\section*{4.11 PVS Specifications for Generic Transformations}

The PVS type that best describes RTL transformations is a \textit{function type} which defines the functions whose domain and range are of the control graph type, \texttt{c}\_\texttt{graph}. This simple definition does not contain any information about the computational behavior correctness of transformations, which is the main issue during our proofs and should be specifically stated as an invariant for each transformation. PVS provides a very convenient way to express the correctness invariant without complicating the specification files: define a \textit{correct transformation} as a \textit{predicate subtype} of \texttt{transform} induced by the behavior preserving condition.

\[
\texttt{transform} : \texttt{TYPE} = [\texttt{c}\_\texttt{graph}\rightarrow\texttt{c}\_\texttt{graph}] \\
\texttt{correct}\_\texttt{tr} : \texttt{TYPE} = \{t:\texttt{transform} | \forall cg:\texttt{c}\_\texttt{graph}, r:(\texttt{output}\_\texttt{registers}(cg)) : \\
\texttt{extracted}\_\texttt{behavior}(\text{reg}(r),cg) = \\
\texttt{extracted}\_\texttt{behavior}(\text{reg}(r),t(cg))\}
\]

Using the above PVS specification, a finite sequence of behavior preserving RTL transformations
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)) OR
   (cons?(cg) => exists (n:below[length(cg)-1]): ot(cg) = RTM(cg,n)) OR
   (exists (n:below[length(cg)], subst:(inputs(nth(cg,n))), wth:(reg?)):
     ot(cg) = RIS(cg,n,subst,wth)) OR
   (exists (n:below[length(cg)], subst:(op?),opr:operator):
     ot(cg) = OIS(cg,n,subst,opr)) OR
   (exists (n:below[length(cg)],opr:operator,l:op_link):
     ot(cg) = OC(cg,n,opr,l)) OR
   (exists (n:below[length(cg)]): ot(cg) = clean_outputs(cg,n)) OR
   (exists (n:below[length(cg)]): ot(cg) = delete_rt(cg,n)) OR
   (exists (n:below[length(cg)],r:(reg?): ot(cg,r) = add_outputs(cg,n,r))
\}

Figure 4.11: Definition for Operational Transformations

can be conveniently defined as a list of elements of the correct_tr type. When constructively defining such sequences (as will be presented in Chapter 5), the PVS type-checker will automatically enforce (and attempt to prove) the correctness invariant as a proof obligation. Thus, the use of predicate subtypes brings an even more important advantage (besides the conciseness and clarity of specifications): it avoids an error-prone manual specification of invariants.

We specify the complete set of transformations as a subtype oper_tr of correct_tr, induced by the ability to express a transformation as an Operator Instance Substitution, Operator Copy, Register Transfer Split, Register Transfer Merge, Register Transfer Decompose, Register Transfer Compose, Register Instance Substitution, Clean Outputs, Add Outputs or Delete Register Transfer. The transformation iden_tr below is the identity transformation, it leaves the control graph unchanged. This specification is shown in Figure 4.11.

69
4.12 Summary

We described in this chapter the PVS specifications for ten RTL transformations. In specifying the transformations we used a definitional approach, which is much secure and “descriptive” than an axiomatic approach. 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.

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 generic correct transformation was defined as a transformation which preserves the extracted behavior of the output registers of the control graph. A set of transformations for which we will prove completeness was defined as the type of correct transformations that can be expressed using the seven transformations specified in Sections 4.2-4.8.
Chapter 5

A Mechanized Proof of Completeness for the Set of RTL Transformations

From the point of view of a transformational derivation synthesis system, the existence of a finite and complete set of transformations ensures that only a finite (and preferably small) number of transformations need to be implemented in order to allow a virtually exhaustive search of the design space (for a well-defined class of designs). Also, any HLS algorithm that performs only structural optimizations can be viewed as a sequential application of RTL transformations; having identified a finite and complete set, then we 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 the synthesis process by attempting to identify such sequences [24].

We present here a mechanized constructive proof of completeness for the set of RTL transformations described in the previous chapters. The proof is constructive in the sense that a finite sequence of transformations (belonging to the complete set), that transforms a certain design implementation into another one having the same behavior, is algorithmically created. The algorithm developed and described in detail in Section 5.2 has also important practical advantages. Consider an application example where an initial implementation of a design exists (synthesized or manually generated). The user wants to interactively improve the latency performance of this design, and the most obvious way to attempt this is by combining two successive register transfers. There are two transformations in our set that can achieve this: RTM (merge) and RTC (compose), but both of them have to be applied in the presence of a certain set of preconditions. Figure 5.1 shows two successive register transfers that can be neither merged nor composed.

One of the RTM preconditions states that the set of output registers of the transfer \( rt2 \) (\( \{B, E, C\} \) in Figure 5.1) should be a subset of the set of input registers of transfer \( rt1 \) (\( \{B, F\} \)). To conform
with the RTC preconditions, the set of input registers of \( rt1 \) and the set of output registers of \( rt2 \) should not intersect. In our example, these two conditions are not satisfied, so RTM or RTC can not be used. However, according to a corollary of the completeness property, there exists a sequence of transformations that result in an equivalent implementation where the operations performed initially in two separate register transfers will be, in the transformed design, performed in the same register transfer. In Section 5.2.1 we present a constructive proof based on an algorithm which applies a sequence of correct RTL transformations that result in combining two successive register transfers. The sequences thus constructed are be used to instantiate subsequent formulas in order to prove subgoals of the completeness theorem, but also, as illustrated by the example above, they represent useful results of a proven correct algorithm.

In the next section we define and formalize the property of completeness. Again, we use those PVS types that best describe the abstract notions used in our definitions. In the rest of the chapter we describe the proof strategy for proving completeness of the core set of RTL transformations presented in Chapters 3 and 4.

### 5.1 Formal Definition and PVS Specification of Completeness

A set of correct (behavior-preserving) RTL transformations is said to be complete if for any two RTL designs that 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 call transformations belonging to this complete set operational transformations (OT), and RTL designs implementing the same behavior - equivalent designs.
**Definition.** Let \( s \) be a set of RTL transformations. \( s \) is **complete** if:

\[
\forall (rtl1, rtl2 : designs, r : output\_register) : \\
(e\_behavior(r, rtl1) = e\_behavior(r, rtl2) \Rightarrow \exists (seq : (s)) : apply(seq)(rtl1) = rtl2)
\]

where: \( e\_behavior \) 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 that recursively applies transformations in the order defined by \( seq \).

A sequence of operational transformations is specified in PVS as a list of elements of the \( oper\_tr \) type. Remember that \( oper\_tr \) was defined in the introduction of Chapter 4 as a predicate subtype induced by the ability to express a correct transformation (of type \( correct\_tr \)) as a merge, split, compose, etc. The \( apply\_sequence \) function below recursively defines the application of a sequence of transformations.

```pvs
oper_tr_sequence = TYPE = list[oper_tr]
apply_sequence(s:oper_tr_sequence) = RECURSIVE correct_tr = 
  if null?(s) then iden_tr else LAMBDA (x: c_graph) : apply_sequence(cdr(s))(car(s)(x)) endif 
MEASURE length(s)
```

The Completeness Theorem states that for any two equivalent designs, \( cg1 \) and \( cg2 \), there exists a sequence of operational transformations which when applied to \( cg1 \) results in \( cg2 \). Two designs are equivalent (expressed by the \( equiv\_cg? \) predicate in Section 2.3.4) if they have the same set of output registers and each output register has the same extracted behavior in the two designs.

```pvs
COMPLETENESS: THEOREM
forall(cg1,cg2:cons_graph) :
  equiv_cg?(cg1,cg2) => exists(s:oper_tr_sequence) : eq?(apply_sequence(s)(cg1), cg2)
```

### 5.2 Proving Completeness for a Set of Transformations: A Proof Tree

Figure 5.2 illustrates the approach we used in proving completeness for the set of operational transformations presented in the previous chapters. The proof follows the general idea of Vemuri [36, 34] but avoids resorting to a unique normal form, which would impose over-specifying models for
RTL designs and their constituent register transfers. We formulated the following three subgoals which together imply completeness:

1. For every RTL design $rtl1$ there exists a sequence of operational transformations that 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 operational transformations that 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 operational transformations.

![Diagram](image)

Figure 5.2: Approach for the Completeness Proof

In the following figure, the three lemmas correspond to PVS specifications of these three subgoals. $cg_1$ and $cg_2$ are two equivalent designs. The $\text{single rt cgs}$ function defines the set of all RTL designs containing only one register transfer that implement the behavior description $bd$ for the set of registers in the $\text{output regs}$ set. The $\text{cg behavior}$ function returns the behavior description implemented by the control graph $cg$. (Section 2.3.3 contains the $\text{beh description}$ and $\text{cg behavior}$ definitions.)
single_rt_cgs(bd:beh_description,output_regs:finite_set[(reg?)]) : set[c_graph] =
{cg:c_graph | cons?(cg) and % these two conditions imply that
  length(cg) < 2 and % cg has length 1
  outregs(car(cg)) = output_regs and
  forall(r:(output_regs)) : extracted_behavior(reg(r),cg) = bd(r)}

LEMMA_1: LEMMA
forall(cg:cons_graph) :
  exists(s:oper_tr_sequence) :
    single_rt_cgs(cg_behavior(cg),output_registers(cg))(apply_sequence(s)(cg))

LEMMA_2: LEMMA
forall(bd:beh_description,output_regs:finite_set[(reg?)]) :
  (forall(cg1,cg2:(single_rt_cgs(bd,output_regs))) :
    exists (s:oper_tr_sequence) : eq?(apply_sequence(s)(cg1),cg2))

LEMMA_3: LEMMA
forall(cg:cons_graph) :
  exists (cgs:c_graph,s:oper_tr_sequence) :
    single_rt_cgs(cg_behavior(cg),output_registers(cg))(cgs) and
    eq?(apply_sequence(s)(cgs),cg)

The first two subgoals were constructively proved by instantiating the sequent formulas with algorithmically created witness sequences of operational transformations. The proof strategy for these subgoals is presented in the next sections. A proof for the Completeness Theorem was derived from the above lemmas using appropriate instantiations (The proof script appears in Appendix D).

5.2.1 A Constructive Proof for LEMMA_1

The first subgoal formulated in the previous section states that any RTL design can be transformed into an equivalent design containing only one register transfer, by using only operational transformations. That is, any control graph associated with an RTL design can be transformed into a control graph of length 1 after repeatedly applying a sequence of operational transformations.

\[ \forall (rtl : c_graph) : (\exists (s : ot_eq) : length(apply(s)(rtl)) = 1) \]

Intuitively, this can be achieved through a sequence of transformations that undo the actions taken by the scheduling algorithms of HLS tools; the only two transformations that allow this are RTM and RTC. As illustrated in the introduction of this chapter, 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 operational transformations that create a scenario where preconditions for merging or composing are satisfied.

**Definition.** We define preconditioning for a transformation as the application of a sequence of operational transformations that results in an equivalent control graph having the same length with the initial one, but where the preconditions for transformation 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 one. The algorithm that defines this witness sequence consists of the following steps (as illustrated in Figure 5.3):

1. **Precondition for RTD.** 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. (Figure 5.3.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 (\(R_i\) in Figure 5.3.c), (b) preconditioning for merging or composing the newly created transfer with the first register transfer, and (a sequence of OIS transformations), and (c) merge or compose the new transfer with the first transfer (Figure 5.3.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 (Figure 5.3.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.

**Example.** Figure 5.4 shows a small RTL design example. The control graph \(CG\) represents the sequence of register transfers to be executed in the data path \(DP\). Interconnections between elements of the datapath are shown with thick lines for transfer 1, thin lines for transfer 2 and dotted line for transfer 3.
Figure 5.3: Reducing the Length of a Control Graph

Figure 5.5 illustrates the application of the first preconditioning step: every register that appears as both an input and an output in the second register transfer (register B in this example) is substituted with a temporary register (T). The RIS transform that performs this substitution considers the particular case when the substituted register (B) is a primary input. It then inserts a simple register transfer, as defined in Section 4.8. To preserve the length of the control graph, the simple register transfer inserted is composed with the next transfer (3 in our example). In the next paragraph we define an RISPLUS macro as an RIS followed by RTC. RISPLUS preserves the length of the graph when RIS is applied to a primary input (Figure 5.5.b).

In the second step of our constructive proof, a decomposition is performed on one output. Figure 5.6 shows the result of this decomposition applied to register B as an output of register transfer 2. In order to perform a merge of the first two register transfers (1 and 2 in the figure), a preconditioning for RTM is applied to the new register transfer generated after decomposition (transfer 2).

Every operator in the exp_set field of this transfer is substituted with an operator instance (using OIS) which does not appear in the first register transfer. Here, OP1 is substituted with OP3. Figure 5.7 shows the result of these substitutions. The substituting operator must have the same functionality as the substituted operator.

The RTM transformation can be now correctly applied. Its outcome is shown in Figure 5.8. This concludes the application of the second step, and one iteration of the third step.

The above sequence of transformations, (RTD, OIS and RTM/RTC) is recursively resumed until all
Figure 5.4: The Initial Design

Figure 5.5: The Preconditioning for RTD (Step 1)
Figure 5.6: The Decomposition for Output Register B (Step 2)

Figure 5.7: The Preconditioning for RTM (Step 2)
Figure 5.8: Merging the First Two Transfers (Step 2)

Figure 5.9: Decomposing for Output Register O2 (Iteration of Step 3)
Figure 5.10: Preconditioning for RTC (Iteration of Step 3)

Figure 5.11: Composing the First Two Transfers (Iteration of Step 2)
Figure 5.12: Reduce the Length of the Control Graph (Ending an Iteration of Step 3)

Figure 5.13: Reduce the Length of the Graph From 2 to 1 (Step 4)
outputs of the second transfer are either merged or composed with the first transfer. In our example there is only one remaining output, $O_2$. The decomposition corresponding to this output results in the configuration shown in Figure 5.9. Here, after decomposition, an empty register transfer (a delay node) is generated.

Next, the second transfer is preconditioned for RTC, by substituting the appearance of $O_2$ with the new operator $O_4$ (Figure 5.10). Transfer 2 is composed with transfer 1 (Figure 5.11). A delete rt operation removes this empty transfer. After this step the length of the control graph is reduced to 2 (Figure 5.12). This terminates the application of the third step, and one iteration of the fourth step.

The fourth step of the proof defines the recursive application of the previous steps until the length of the graph is reduced to 2. This is the case in our example.

Figure 5.13 illustrates the sequence of transformations that reduces the length of the graph to one, which is the goal of our constructive proof. The second register transfer is decomposed for $T$ (Figure 5.13.b), and the length is temporary increased to three. An RTM transformation applied to the first two register transfers recover the initial length of the graph (Figure 5.13.c). The RTD transformation applied to register $D$ generates an empty register transfer (Figure 5.13.d) which is subsequently deleted (Figure 5.13.e). After preconditioning for composition, by applying OIS to the $O_2$ operator in the second register transfer (Figure 5.13.e), the two register transfers are composed. This results in the single register transfer implementation shown in Figure 5.13.f.

1. **Preconditioning for RTD.** The RTD precondition requires that the set of input and output registers of the affected register transfer are disjoint. To achieve this we define a break cycles function that recursively substitutes each register that appear as both an input and an output (belongs to a loop set) with a temporary one such that, after completing the application, the loop set is null. The RISPLUS function is a macro of two transformations: RIS and RTM. Remember from Section 4.8 that, when the substituted register is a primary input, the result of the RIS transformation may increase the length of the control graph. RISPLUS is a sequence of two transformations: an RIS, followed by composing the simple register transfer introduced in this particular case with the next transfer, such that the length of the control graph is preserved. RISPLUS always substitutes an existing register transfer with a new (temporary) register, so, after each substitution, one cycle is broken.
According to our definition for preconditioning, the `break_cycles` function applied to the second register transfer of a control graph whose length is greater than two is a preconditioning for RTD: it defines a sequence of operational transformations (RIS and RTC) that preserves the length of the control graph, and creates the preconditions for decomposing the second register transfer.

In the following definition, `preconditioned` is a function applied to the clean version of a control graph: a particular case in the application of CO transformation, clean output, ensures that there are no unused writes, i.e., no output registers in the second register transfer are written in the first register transfer without being previously read. Preconditioning is applied to graph lists with length greater than 1 (nonsimple? defines this type of graphs). The `preconditioned_sequence` lemma states the existence of a sequence of operational transformations that results in a preconditioned graph. Following the recursive definition of `break_cycles`, the lemma was proved by induction on the cardinality of the loop set. A description of the proof strategy used is in Appendix D.1
2. **Reduce the cardinality of the output registers set of the second transfer.** In the PVS specification below, the `reduce_outregs` function results in an equivalent control graph having the same length as the initial one, but where the set of output registers of the second transfer contains one register less. First, the RTD transformation is applied to an arbitrary output register of the second transfer in the preconditioned graph. Following this decomposition, the `NEW_OPS` function preconditions the newly created transfer for merging or composing; it substitutes each operator instance with a new operator which is not used in first transfer. The register transfer resulted after decomposition contains only one output register (Section 4.6). If this register appears as an input in the first transfer of the graph, then RTM can be safely applied; if not, than RTC is applied.

```plaintext
define reduce_outregs(cg:(nonsimple?)) : (nonsimple?) =  
  if empty?(nth(preconditioned(cg),1)) then preconditioned(cg)  
  else let r:(reg?) = choose(nth(preconditioned(cg),1)`outregs),  
    oset : set[operator] = operators(cg),  
    dec: cons_graph = NEW_OPS(RTD(preconditioned(cg),1,r),1,oset) in  
    if inputs(nth(dec,0))(r) then RTM(dec,0) else RTC(dec,0) endif  
  endif  
reduce_outregs_sequence: LEMMA  
forall(cg:(nonsimple?)) :  
  exists (s:oper_tr_sequence) : reduce_outregs(cg) = apply_sequence(s)(cg)```

We proved that for any control graph with length greater than one, the `reduce_outregs` function is equivalent with the application of a sequence of operational transformations, namely: the sequence defined by the `preconditioned` function (obtained from lemma `preconditioned_sequence`), followed by a decomposition, a sequence of operator substitutions (corresponding to `NEW_OPS`), and a merge or compose transformation.

3. **Reduce the length of the control graph.** In this step, the `reduce_outregs` function is recursively applied until all computations corresponding to output registers of the second transfer are combined with the first transfer. The function `eliminate_rt` below results in an equivalent control graph, having the same length as the initial one, but where the second register transfer is `empty`, that is, its set of output registers is null. An empty transfer could be interpreted in an extended model as a delay node. The deletion of a delay node requires defining a `delete_rt` function that simply deletes an empty transfer from the control graph list. The `eliminate_rt` lemma was proved by induction on the cardinality of the `outregs` set of the second transfer, and using the witness sequence resulted after skolemizing the `reduce_outregs_sequence` lemma.
4. Reduce the length of the control graph to 2. The length of the control graph list resulted after applying the eliminate Rt function is decremented by deleting the empty transfer. After deletion, the process described in the previous step is recursively applied such that the graph length can be further reduced. The reduce cg length function below recursively reduces the length until the resulted list contains only two transfers. The length reduction stops at this point because the break cycles function used to precondition for decomposition will not preserve the graph length for lists with length 2. This particular case is dealt with in the next step. The reduces cg length sequence lemma was proved by induction on the length of the graph list, and using the witness sequence defined by the eliminate Rt sequence lemma.

5. Reduce the length of the graph from 2 to 1. After applying the preconditioned function to the second (and last) transfer, the length of the graph might increase. The added register transfer will contain only simple value transfers, as described in Section 4.8. In this case, after eliminating the second register transfer, a preconditioning step is not required any more, so the length of the graph can not further increase. The length2to1 function is applied to a graph with only two transfers (of type (length2?)) and results in graphs of length one (of type (simple?)). Its application is illustrated in Figure 5.14. The proof for the length2to1 sequence lemma makes use of the witness sequence supplied by the eliminate Rt lemma and is constructed by appending delete rt functions to these sequences.
length2to1\(cg:\text{(length2?)}) : \text{(simple?)}\) = 
if length(eliminate\_rt\(cg\)) = 2 then delete\_rt(eliminate\_rt\(cg\),1) 
else delete\_rt(eliminate\_rt(delete\_rt(eliminate\_rt\(cg\),1)),1) 
endif

\text{length2to1\_sequence: LEMMA}
forall\(cg:\text{(length2?)}) : 
exists (s:\text{oper\_tr\_sequence}) : \text{length2to1\(cg\}) = \text{apply\_sequence}(s)(cg)

Figure 5.14: The Application of the 2to1 function

The first subgoal was proved by instantiating the consequent with the sequence of transformations obtained by appending the witness sequences whose existence is stated (and has been constructively proved) by the \text{reduce\_cg\_length\_sequence} and the \text{length2to1\_sequence} lemmas.

5.2.2 The Type-Checking Mechanism as Means to Express Invariants

This section describes a constructive proof for the first subgoal. Sequences of transformations were algorithmically defined and lemmas that assert the existence of sequences of operational transformations that have the same effect as the constructed sequences, were proved. After executing the last step we were able to provide a witness used to instantiate the sequent formula of the first subgoal. Note that along the specification exercise we have not explicitly stated the well-formedness or the correctness invariants, nor have we included sublemmas that assert that a certain transformation belongs to the complete set. Instead, the invariants were implicitly enforced by the predicate subtype definitions for \text{oper\_tr}, and automatically expressed by the PVS type-checker as proof obligations.

In the remaining part of this section we explain how the PVS predicate subtype mechanism creates the proof obligations for these invariants.
The `oper_tr` type was defined as a predicate subtype using a declaration of the form:

```
oper_tr : TYPE = {ot:correct_tr | Po(ot)},
```

where `Po` is a predicate defining those `correct_tr` elements that can be expressed as applications of operational transformations (the complete definition is in Section 4.11).

The `correct_tr` type is defined as:

```
correct_tr : TYPE = {ct:transform | Pc(ct)},
```

where `Pc` is the predicate that returns true for those functions `[c_graph -> c_graph]` that preserve the behavior correctness of the transformed design (the complete definition is in Section 4.11).

The `transform` type contains functions of the form `[c_graph -> c_graph]`, i.e. those functions which when applied to control graphs return also control graphs (lists of `well-formed` register transfers).

PVS makes use of a sound set of constraints that are automatically asserted upon type-checking the subtypes in order to ensure their consistency. The two constraints corresponding to type formulas used in our specification are [47]:

1. \( \Pi(\{y : T | a\}) = \lambda(x : \mu(T)) : (\Pi(T)(x) \land a[x/y]) \)
2. \( \Pi([A \rightarrow B]) = \lambda(x : [A \rightarrow \mu(B)]) : (\forall(y : A) : \Pi(B)(x(y))) \)

where \( \Pi(S) \) is the collection of predicates that constrain a subtype \( S \) relative to its maximal supertype \( \mu(S) \).

If during a proof an instantiation is performed with an element that should be of the `oper_tr` type (as it is often the case in our proofs), then the type-checker creates a new subgoal that requires proving that this element is of the correct type. The assertion of the new subgoal follows the constraining rules presented above.

When applied to the `oper_tr` type the first rule yields the following constraint:

\[ \Pi(\text{oper}\_\text{type}) = \lambda(x : \mu(\text{correct}\_\text{tr})) : (\Pi(\text{correct}\_\text{tr})(x) \land P_o[x/y]) \]

For the `correct_tr` type:

\[ \Pi(\text{correct}\_\text{tr}) = \lambda(x : \mu(\text{transform})) : (\Pi(\text{transform})(x) \land P_c[x/y]) \]

The `transform` type is a function type, so the second constraining rule is applied:

\[ \Pi(\text{transform}) = \lambda(x : [c\_\text{graph} \rightarrow \mu(c\_\text{graph})]) : (\forall(y : c\_\text{graph}) : \Pi(c\_\text{graph})(x(y))) \]

where \( \Pi(c\_\text{graph}) \), according to the definition of `c_graph` in Section 2.3.1, requires that every element of the list \( x(y) \) is a `well-formed` register transfer.

If an element `ot` is claimed to be of the `oper_tr` type, then the corresponding constraining predicate has to be true:
\[ \Pi(\text{oper\_tr})(ot) = \Pi(\text{correct\_tr})(ot) \land P_o[ot] = \\
(\Pi(\text{transform})(ot) \land P_c[ot]) \land P_o[ot] = \\
(\forall (y : \text{c\_graph}) : \Pi(\text{c\_graph})(ot(y))) \land P_c[ot] \land P_o[ot] \]

The above formula reads: an element \( ot \) is of the \text{oper\_tr} type if when applied to any control graph it results in a control graph, i.e., in a list of transfers where all transfers are well-formed (the \( P_c \) predicate), it preserves the behavior of the graph to which it is applied (\( P_c \) is satisfied), and it belongs to the set of operational transformations defined by the \( P_o \) predicate. This condition corresponds to the invariants we wanted to state from the beginning.

In suggesting the proof strategies used for discharging subgoals of the first lemma we commonly used a succinct explanations of the form: “The proof was derived by appending the T transformation to the witness sequence supplied by ...” But any of the transformations described so far are defined for a particular tuple consisting of: a control graph, the position of the affected register transfer, a particular operator or register. For instance, the OIS transformation definition has the form:

\[
\text{OIS}(cg:c\_graph,n:below[length(cg)], subst:(op?), opr:operator) : c\_graph
\]

The \( T \) transformation actually appended in our proofs is a lambda expression that defines a function that can be expressed as an operational transformation. For example, in the proof derived for the \text{reduce\_outregs\_sequence} lemma, when we append an \text{RTD} transformation to the witness sequence supplied by the \text{preconditioned\_sequence} lemma, we actually append the following transformation:

```
rdtf(cg:(nonsimple?)) : oper\_tr = 
LAMBDA(x:c\_graph) : 
  if x = preconditioned(cg) then 
    if empty?(nth(preconditioned(cg),1)) then preconditioned(cg) 
    else let r:(reg?) = choose(nth(preconditioned(cg),1)\'outregs) in 
      RTD(preconditioned(cg),1,r) 
    endif 
  else iden\_tr(x) endif
```

This transformation leaves the argument \( x \) unchanged if \( x \) does not equal \text{preconditioned}(cg) or when the second register transfer of \( cg \) is empty, and it applies the \text{RTD} function otherwise. Upon performing this instantiation within the interactive proof-checker environment, the type-checker generates the TCC obligation shown bellow. This TCC is extremely long but it only contains the three properties that should hold for an operational transformation: the well-formedness of the
The specification files would become very large and hard to follow if these three invariants were explicitly expressed. We showed in this section that, even if not explicitly stated, these invariants are not overlooked. They appear as TCC obligations and have been discharged using the proof strategy shown in Appendix D.2.

\[
\text{rdtf\_TCC3: OBLIGATION}
\]

\[
\begin{align*}
&\text{FORALL (cg: \{nonsimple\}): (FORALL (cg\_259: c\_graph):} \\
&\quad \text{wellformed\_cg?}(\text{IF cg\_259 = preconditioned(cg)} \\
&\quad \text{THEN IF empty?}(\text{nth[wellformed\_rt]}(\text{preconditioned}(cg), 1)) \text{ THEN preconditioned(cg)} \\
&\quad \text{ELSE RTD}(\text{preconditioned}(cg), 1, \\
&\quad \quad \text{choose([^\{\text{reg?}\}]){\text{nth[wellformed\_rt]}(\text{preconditioned}(cg), 1)'outregs}}) \\
&\quad \text{ENDIF} \\
&\quad \text{ELSE iden\_tr(cg\_259)} \\
&\quad \text{ENDIF)} \\
&\quad \text{AND} \\
&\quad (\text{FORALL (cg\_260: c\_graph, r: (output\_registers(cg\_260))):} \\
&\quad \text{extracted\_behavior}(\text{reg[register, operator]}(r), cg\_260) = \\
&\quad \text{extracted\_behavior}(\text{reg[register, operator]}(r), \\
&\quad \text{IF cg\_260 = preconditioned(cg)} \\
&\quad \text{THEN IF empty?}(\text{nth[wellformed\_rt]}(\text{preconditioned}(cg), 1)) \text{ THEN preconditioned(cg)} \\
&\quad \text{ELSE RTD} \\
&\quad \quad (\text{preconditioned}(cg), 1, \\
&\quad \quad \text{choose([^\{\text{reg?}\}]){\text{nth[wellformed\_rt]}(\text{preconditioned}(cg), 1)'outregs}}) \\
&\quad \text{ENDIF} \\
&\quad \text{ELSE iden\_tr(cg\_260)} \\
&\quad \text{ENDIF)}) \\
&\quad \text{AND} \\
&\quad (\text{FORALL (cg\_258: c\_graph):} \\
&\quad (\text{IF cg\_258 = preconditioned(cg)} \\
&\quad \text{THEN IF empty?}(\text{nth[wellformed\_rt]}(\text{preconditioned}(cg), 1)) \text{ THEN preconditioned(cg)} \\
&\quad \text{ELSE RTD}(\text{preconditioned}(cg), 1, \text{choose([^\{\text{reg?}\}]){\text{nth[wellformed\_rt]}(\text{preconditioned}(cg), 1)'outregs}}) \\
&\quad \text{ENDIF} \\
&\quad \text{ELSE iden\_tr(cg\_258)} \\
&\quad \text{ENDIF} \\
&\quad = \text{iden\_tr(cg\_258)} \\
&\quad \text{OR} \\
&\quad (\text{EXISTS (n: below[length(cg\_258)], r: (nth[wellformed\_rt]}(cg\_258, n)'outregs)):} \\
&\quad \text{IF cg\_258 = preconditioned(cg)} \\
&\quad \text{THEN IF empty?}(\text{nth[wellformed\_rt]}(\text{preconditioned}(cg), 1)) \text{ THEN preconditioned(cg)} \\
&\quad \text{ELSE RTD}(\text{preconditioned}(cg), 1, \text{choose([^\{\text{reg?}\}]){\text{nth[wellformed\_rt]}(\text{preconditioned}(cg), 1)'outregs}}) \\
&\quad \text{ENDIF} \\
&\quad \text{ELSE iden\_tr(cg\_258)} \\
&\quad \text{ENDIF} \\
&\quad = \text{RTD}(cg\_258, n, r)) \\
&\quad \text{OR} \\
&\quad \text{.............}
\end{align*}
\]
5.2.3 Proof Strategy for LEMMA_2

The second subgoal states that any two equivalent RTL designs (designs implementing the same behavior) consisting of only one register transfer, can be derived one from the other through a sequence of operational transformations.

\[
\forall (rtl1, rtl2 : c\_graph) :
\]
\[
(length(rtl1) = length(rtl2) = 1 \land c\_behavior(rtl1) = c\_behavior(rtl2))
\]
\[
\Rightarrow (\exists (s : ot\_seq) : rtl2 = apply(s)(rtl1))
\]

The equivalence of two designs (expressed below as designs belonging to the same single_rt_cgs set) means that each register in the set of output registers has the same extracted behavior. Therefore, what distinguishes such equivalent designs is only a possibly different sharing of operators and a different assignment of operators to physical instances. In what follows we will commonly refer to equivalent control graphs consisting of only one register transfer as simply equivalent register transfers.

```haskell
single_rt_cgs(bd:beh_description, output_regs: finite_set[(reg?)]) : set[c\_graph] =
{cg:c\_graph | cons?(cg) and % these two conditions imply that
 length(cg) < 2 and % cg has length 1
 outregs(car(cg)) = output_regs and
 forall(r:(output_regs)) : extracted_behavior(reg(r),cg) = bd(r)}

LEMMA_2: LEMMA
forall(bd:beh_description, output_regs: finite_set[(reg?)]) :
(forall(cg1,cg2:(single_rt_cgs(bd,output_regs))) :
 exists (s:oper_tr_sequence) : eq?(apply_sequence(s)(cg1),cg2))
```

Consider the example of the two register transfers shown in Figure 5.15. The output registers O1 and O2 have the same extracted behavior in the two transfers (rt1 and rt2), so these two transfers are equivalent. After substituting the appearances of OP1 and OP2 in rt1 with OP3 and OP10 respectively, and after performing a copy operation for operator OP7 with operator OP9, the resulted register transfer equals rt2.
This is a small example where the sequence of transformations that transforms \( \text{rt1} \) into \( \text{rt2} \) could be visually identified. In order to prove \textbf{LEMMA 2} we need to define a systematic sequence of transformations that can be proved to transform one register transfer into any other equivalent one.

The approach of Vemuri \cite{36} in proving completeness makes use of a Normal Form Structure definition, which is a unique implementation of a given behavior description. The Normal Form Structure is in the \textit{Maximum Operators Form}, that is, all operator sharings are eliminated (Figure 5.16.a). There is an important annotation to this definition: the Normal Form is unique \textit{within naming of the hardware components}. When formally specifying in a strict formal reasoning system like PVS, such a normal form is not a unique design, but a class of designs that implement the same behavior in a \textit{particular form} that defines a set of \textit{isomorphic structures}. By \textit{particular form} we mean the set of designs defined by a certain predicate.

![Diagram of Two Equivalent Register Transfers](image)

\begin{figure}
\centering
\includegraphics[width=\textwidth]{figure515.png}
\caption{Two Equivalent Register Transfers}
\end{figure}

We postulate that:

\begin{enumerate}
\item \textit{Two register transfers having isomorphic structures can be transformed into one another by a finite sequence of Operator Instance Substitutions (OIS).}
\end{enumerate}

We claim that this postulate implies the \textit{uniqueness within naming of the hardware components}. Indeed, the correct application of OIS when the \textit{substituting} operator instance is not already in use in the register transfer has the effect of renaming the \textit{substituted} operator.

Using this Postulate, a proof for \textbf{LEMMA 2} can be derived from the following three subgoals:

\begin{enumerate}
\item Any design consisting of only one register transfer can be brought to a particular form \( F \) by applying a finite sequence of operational transformations.
\end{enumerate}
2. Any two equivalent designs that implement the behavior in the form $F$ are structurally isomorphic.

3. For any design $cg$ consisting of one register transfer there exists an equivalent design in the form $F$ from which $cg$ can be derived by applying a finite sequence of operational transformations.

The remaining of this section is organized as follows: In the first subsection we define the particular form of implementation ($F$) as a Minimum Operators Form, and prove the first subgoal for Lemma 2. The second subsection defines the structural isomorphism for register transfers and proves this isomorphism for implementations in the Minimum Operators Form. The third subsection sketches the proof derived for Lemma 2 from the three subgoals.

**Proving The First Subgoal of Lemma 2**

**Definition.** A register transfer $rt$ is said to be in the Minimum Operators Form if for any two expressions $e_1$ and $e_2$ in $rt$, if $e_1$ and $e_2$ have the same functionality and the same right and left sources, respectively, then $e_1 = e_2$.

In other words, the sharing of operators in $rt$ is maximized (and the number of operator instances is minimized). Figure 5.16.b shows a register transfer equivalent to that in Figure 5.16.a, but in the Minimum Operators Form. In the following PVS specification this particular form is the set of implementations defined by the predicate `minform?`. Predicate `minform_exp?` defines the set of operator expressions in a register transfer `rt` that does not share sources with any other

---

Figure 5.16: Equivalent Transfers in Particular Forms of Implementation
functionally equivalent operator in \( rt \). For clarity, we refer to register transfers for which the \( \text{minform?} \) predicate is true as “\( \text{minform?} \) register transfers”.

\[
\text{minform?}(rt:\text{wellformed}_rt) : \text{bool} = \\
\forall(e1,e2:\text{filter}_op(rt'exp_set)) : \\
(\text{opfn}(\text{op}(e1)) = \text{opfn}(\text{op}(e2)) \text{ and} \\
\text{source1}(e1) = \text{source1}(e2) \text{ and source2}(e1) = \text{source2}(e2)) \Rightarrow e1 = e2
\]

\[
\text{minform}_\text{exp?}(rt:\text{transfer}, e:\text{filter}_op(rt'exp_set)) : \text{bool} = \\
\forall(e1:\text{filter}_op(rt'exp_set)) : \\
(\text{opfn}(\text{op}(e1)) = \text{opfn}(\text{op}(e)) \text{ and} \\
\text{source1}(e1) = \text{source1}(e) \text{ and source2}(e1) = \text{source2}(e)) \Rightarrow e1 = e
\]

The use of a Minimum Operators Form as the particular form of implementation is motivated by a very useful property: for every operator expression of a \( \text{minform?} \) register transfer \( rt \) there is no other expression in \( rt \) having the same extracted behavior. This property is expressed by lemma \textbf{minform\_lemma} below. As a corollary of this property, a one-to-one mapping between elements of two \( \text{minform?} \) register transfers exists. This mapping was used in defining the isomorphism between two implementations, as it will be seen in the next section.

\[
\text{minform\_lemma}: \text{LEMMA} \\
\forall(cg:(\text{simple?}), e1,e2:(\text{car}(cg)'exp_set)) : \\
\text{minform?}(cg) \Rightarrow (\text{extracted\_behavior}(e1) = \text{extracted\_behavior}(e2) \Rightarrow e1 = e2)
\]

The first subgoal of \textbf{LEMMA} 2 states that every register transfer can be brought to the Minimum Operators Form through a finite sequence of correct Operator Instance Substitutions. The PVS specification of this subgoal is expressed by the \texttt{cg\_to\_minform} lemma. The sequence of transformations is constructed by the recursive function \texttt{to\_minform} that applies OIS to each operator that shares its sources with another operator that has the same functionality. The \texttt{same\_sources(rt,e)} function returns the set of expressions in \( rt \) that can be substituted with \( e \), i.e., have the same functionality and same sources as \( e \). Informally, \texttt{to\_minform} checks recursively for each operator if there exists a correct OIS transformation involving that operator, and if so, it applies this transformation. After each application of OIS, the cardinality of the operators set is decremented, and the recursion terminates when the base case is met. In this case, all operator expressions are \( \text{minform\_exp?} \), so the register transfer is \( \text{minform?} \).

A sketch proof for lemma \texttt{cg\_to\_minform} is presented in Appendix E.
to_minform(cg:(simple?)) : RECURSIVE c_graph =
if null?(cg) then cg
else if forall (e:(filter_op(car(cg)`exp_set))) : minform_exp?(car(cg),e) then cg
else let e: (op?) = choose(o:(filter_op(car(cg)`exp_set)) | not minform_exp?(car(cg),o)),
    s: (op?) = choose(same_sources(car(cg),e)) in
to_minform(OIS(cg,0,e,op(s)))
endif
endif
MEASURE card(operators(cg))
cg_to_minform: LEMMA
forall(cg:(simple?), e1,e2:(car(cg)`exp_set)) :
exists(s:oper_tr_sequence) : minform?(apply_sequence(s)(cg))

Proving the Second Subgoal of LEMMA 2

According to Postulate (1), if the minform? predicate defines a set of isomorphic structures, then there exists a finite sequence of Operator Instance Substitutions that transforms any minform? implementation into any other equivalent minform? implementation.

An isomorphism between two sets requires a one-to-one correspondence between their elements which also preserves structures. The formal definition of the isomorphic correspondence between sets is [48]:

**Definition.** Let X and Y be two sets, and S and T two relations on elements of sets X, respectively Y. We say that function R establishes an isomorphism between sets X and Y for relations S and T, and write: \( (X,S) \isR (Y,T) \), if and only if:

\[
\begin{align*}
(R \text{ is bijective}) \land \text{domain}(R) &= X \land \text{range}(R) = Y \land \\
\forall(x,y \in X)(z,u \in Y) : xRz \land yRu &\Rightarrow (xSy \equiv zTu)
\end{align*}
\]

For example, consider the sets of even natural numbers corresponding to set X, the set of odd natural numbers corresponding to Y, the less then relation on natural numbers corresponding to S and T. The increment function defines an isomorphism between sets of even and odd numbers for the less then relation.

The definition above is stated only for binary relations, but it can be easily extended to relations with arbitrary number of variables [48].

The relations for which we want to define register transfers isomorphism refers to the structural
properties of their constituent operator expressions. Two sets of operator expressions are structurally equivalent if they contain the same number of operators, with the same functionalities and interconnected in identical patterns. According to our PVS definition for operator expressions (Section 2.3.1), this interconnection is preserved if corresponding operator expressions have corresponding left and right sources. Figure 5.17 shows two structural isomorphic register transfers. The corresponding operators have the same functionality.

Figure 5.17: Two Isomorphic Register Transfers

We express the relations corresponding to \( S \) and \( T \) in the isomorphism definition as such properties concerning the interconnection configuration; they are ternary relations on the expression sets of the two register transfers:

\[
S : (rt1'\text{exp}_\text{set} \times rt1'\text{exp}_\text{set} \times rt1'\text{exp}_\text{set}) \rightarrow \text{bool}; \\
S(x, y, z) \triangleq (\text{source}1(x) = y \wedge \text{source}2(x) = z)
\]

\[
T : (rt2'\text{exp}_\text{set} \times rt2'\text{exp}_\text{set} \times rt2'\text{exp}_\text{set}) \rightarrow \text{bool}; \\
T(m, n, p) \triangleq (\text{source}1(m) = n \wedge \text{source}2(m) = p)
\]

To prove structural isomorphism between two equivalent register transfers that are \texttt{minform}\? for relations \( S \) and \( T \), we need to find a bijective (one-to-one) function that maps expressions from one register transfer to functionally equivalent expressions from the other (and vice versa), while preserving the structural relations (2) and (3). The \texttt{bijective\_mapping\_exp} lemma in the PVS specification below states the existence of such a function. It was proved using the fact that in a \texttt{minform}\? register transfer there exists only one expression having a given extracted behavior (the \texttt{minform\_lemma} above), and from the definition of equivalent transfers as having the same extracted behavior. The \texttt{clean\_rt?} predicate in \texttt{bijective\_mapping\_exp} states that every control graph is...
clean of floating operators - operators which are not used in computing any output value (their outputs are not connected to any other component of the register transfer). It was reasonable to assert that the two register transfers compared are clean. This does not restrict the models, it only means that, when comparing two register transfers, we are ignoring any not-connected operators.

---

**bijective_mapping_exp: LEMMA**

\[
\text{forall}(cg1,cg2:(\text{minform}?) : \\
(\text{equiv}_\text{rt?}(\text{car}(cg1),\text{car}(cg2)) \text{ and clean}_\text{rt?}(\text{car}(cg1)) \text{ and clean}_\text{rt?}(\text{car}(cg2))) \Rightarrow \\
\text{exists } (f:[(\text{car}(cg1)'\text{exp_set})\rightarrow(\text{car}(cg2)'\text{exp_set})]) : \\
\text{bijective?}(f) \text{ and } \\
\text{forall}(e:(\text{car}(cg1)'\text{exp_set})) : \text{extracted}_\text{behavior}(e) = \text{extracted}_\text{behavior}(f(e))
\]

The **bijective_mapping_exp** lemma supplies a witness for defining the bijective function used in proving the isomorphism of equivalent \text{minform}? transfers, \( f(cg1,cg2) \). The \( \text{equiv}_\text{class}(cg1) \) function defines the set of all \text{minform} transfers that are equivalent to \( cg1 \). The \( f_{\text{opfn}} \) lemma states that the bijective function \( f \) establishes a one-to-one relation between operators having the same functionality, or identical registers.

\[
f(cg1:(\text{minform}?),cg2:(\text{equiv}_\text{class}(cg1))) : [(\text{car}(cg1)'\text{exp_set})\rightarrow(\text{car}(cg2)'\text{exp_set})] = \\
\text{choose}(\text{fun}:[(\text{car}(cg1)'\text{exp_set})\rightarrow(\text{car}(cg2)'\text{exp_set})] | \\
\text{bijective?}(\text{fun}) \text{ and } \\
\text{forall}(e:(\text{car}(cg1)'\text{exp_set})) : \\
\text{extracted}_\text{behavior}(e) = \text{extracted}_\text{behavior}(\text{fun}(e)))
\]

**f_{\text{opfn}} : LEMMA**

\[
\text{forall}(cg1:(\text{minform}?),cg2:(\text{equiv}_\text{class}(cg1)), e:(\text{car}(cg1)'\text{exp_set}) : \\
\text{if } (\text{op?}(e)) \text{ then } \text{opfn}(\text{op}(e)) = \text{opfn}(f(cg1,cg2)(\text{op}(e))) \text{ else } e = f(cg1,cg2)(e) \text{ endif}
\]

**iso_relation: LEMMA**

\[
\text{forall}(cg1:(\text{minform}?),cg2:(\text{equiv}_\text{class}(cg1))) : \\
\text{forall}(x,y,z:(\text{filter}_\text{op}(\text{car}(cg1)'\text{exp_set})), \\
m,n,p:(\text{filter}_\text{op}(\text{car}(cg1)'\text{exp_set}))) : \\
\text{let } \text{bf} : [(\text{car}(cg1)'\text{exp_set})\rightarrow(\text{car}(cg2)'\text{exp_set})] = f(cg1,cg2) \text{ in } \\
((\text{bf}(x) = m \text{ and } \text{bf}(y) = n \text{ and } \text{bf}(z) = p) \Rightarrow \\
((\text{source1}(m)=n \text{ and } \text{source2}(m)=p) \text{ iff } (\text{source1}(x)=y \text{ and } \text{source2}(x)=z)))
\]

Finally, the **iso_relation** lemma states that the bijective function \( f(cg1,cg2) \), that maps expressions in \( cg1 \) to expressions in \( cg2 \) having the same functionality, establishes an isomorphic relation.
between the two equivalent minform? transfers for the structural relations \( S \) and \( T \) defined by formulas (2) and (3). The proof was done by simple expansion of function definitions, propositional simplifications, and by appropriately instantiating lemma \( f_{\text{bijective}} \).

According to Postulate (1), and having proved that two equivalent minform? designs are structurally isomorphic, then there exists a sequence of OIS transformations that transform any minform? design into any other minform? design equivalent to it. This is stated by the following axiom:

\[
\begin{align*}
cg1\_to\_cg2: \text{AXIOM} \\
\forall (cg1: \text{minform?}, cg2: \text{equiv_class}(cg1)) : \\
\exists (f: \text{car}(cg1)'\text{exp_set}) \rightarrow \text{car}(cg2)'\text{exp_set}) : \\
\text{bijective?}(f(cg1,cg2)) \\
\text{and} \\
\forall (e: \text{car}(cg1)'\text{exp_set}) : \\
\text{extracted_behavior}(e) = \text{extracted_behavior}(f(cg1,cg2)(e)) \\
\Rightarrow \\
\exists (s: \text{oper_tr_sequence}) : \text{apply_sequence}(s)(cg1) = cg2
\end{align*}
\]

This axiom is actually provable: a sequence of OIS transformations that \textit{rename} operators in two isomorphic structures can be constructively defined by recurring on the set of operators, and performing substitutions with identical operators for corresponding expressions of the two transfers. However, we did not prove it, instead we used the Postulate (1) and used the reasoning about isomorphic structures presented in this section.

The second subgoal is expressed by lemma \textit{minform_to_minform}, and is easily derived using appropriate instantiations for Postulate (1) (the \textit{cg1_to_cg2} axiom).

\[
\begin{align*}
\text{minform\_to\_minform}: \text{LEMMA} \\
\forall (cg1: \text{minform?}, cg2: \text{equiv_class}(cg1)) : \\
\exists (s: \text{oper_tr_sequence}) : \text{apply_sequence}(s)(cg1) = cg2
\end{align*}
\]

\textbf{Conclusion of the Proof Strategy for LEMMA 2}

\textit{LEMMA 2} was proved by appropriately instantiating lemmas and axioms corresponding to the three subgoals enumerated before: the \textit{cg_to_minform} lemma (corresponding to the first subgoal), the \textit{minform_to_minform} (corresponding to the second subgoal), and the \textit{inverse lemma} below which states that every operational transformation admits as inverse a sequence of operational transformations. A proof for \textit{inverse lemma} is presented in the next section.
cг_to_minform: LEMMA
forall(cг:(simple?)) : exists(s:oper_tr_sequence) : minform?(apply_sequence(s)(cг))

minform_to_minform: LEMMA
forall(cг1:(minform?), cг2:(equiv_class(cг1))) :
exists(s:oper_tr_sequence) : apply_sequence(s)(cг1) = cг2

inverse_lemma: AXIOM forall(t:oper_tr,cг:c_graph) :
exists(s:oper_tr_sequence) : eq?(apply_sequence(s)(t(cг)), cг)

5.2.4 Proof for LEMMA_3

The third subgoal states that for any RTL design cг there exists an equivalent design consisting of only one register transfer from which cг can be derived through a sequence of operational transformations.

∀(rtl : c_graph) : (∃(rs : c_graph, s : ot_seq) : length(rs) = 1 ∧ apply(s)(rs) = rtl)

LEMMA_3: LEMMA
forall(cг:cons_graph) :
exists (cgs:c_graph,s:oper_tr_sequence) :
single_rt_cgs(cг_behavior(cг), output_registers(cг))(cgs) and apply_sequence(s)(cgs) = cг

A proof for this subgoal was derived from LEMMA_1 (constructively proved as described in Section 5.2.1), and the sublemma inverse_sequence_lemma below stating that each sequence of operational transformations admits as inverse a sequence of operational transformations.

The inverse_sequence_lemma was, obviously, proved by induction on the length of the sequence, and using the inverse_lemma axiom stated in the previous Section.
In order to justify the inverse lemma axiom, we prove that each operational transformation, when applied in the presence of a set of preconditions, admits as inverse a sequence of operational transformations. The preconditions required for proving the existence of inverse sequences are weaker than those required for correctness, in the sense that they are satisfied when the correctness preconditions are also satisfied. Since all operational transformations used in proving the first two subgoals are applied only when the correctness preconditions are satisfied, then they also admit as inverses sequences of operational transformations, as defined in the remaining of this chapter.

5.2.5 Proving the Existence of Inverse Sequences for Each Operational Transformation

Informally, each transformation admits an inverse sequence of transformations as follows:

- The inverse of the OC transformation is OIS. The inverse for OIS is a sequence of OC and OIS transformations.
- The inverse of RTS is RTM; the inverse of RTM is a sequence of RTS and RIS transformations.
- The inverse of RTD is RTC, and vice-versa.
- The inverse of RIS is also an RIS, or, when the substituted register is a primary output, a sequence of RIS and RTM.
- The inverse of DRT is RTS applied when the split set is empty or it contains only register expressions.
- The inverse of CO is AO, and vice-versa.

While proving the existence of inverse sequences, the mechanized approach revealed some particular cases in transformations’ application that were overlooked in a paper-and-pencil reasoning style. The precise definitional specification approach, as well as the rigorous mechanized proof, are able to account exhaustively for each particular case, while in the paper-and-pencil style we tend to subjectively consider only some cases. For example, the RTM transformation as defined by Vemuri [36] does not admit as inverse the simple application of RTS, but a sequence of transformations including register substitution, decomposition and merging.

In the following sections we present the inverse sequences and the methodology used for proving the existence of sequence transformations for each operational transformation.
The Inverse Sequence for the OC Transformation

OC admits as inverse a simple transformation, OIS. The oc_inv lemma asserts that by substituting the copy operator with the copied operator results in a register transfer equivalent to the initial one.

\[
\text{oc_inv: LEMMA} \quad \forall (rt: \text{wellformed}_rt, \text{new}_op: \text{operator}, l: \text{op}_i_link) : \\
\quad \text{wellfound_link}(l, \text{new}_op, rt) \Rightarrow \text{eq}(\text{ois}(\text{oc}(rt, \text{new}_op, l), \text{op}_\text{copy}_\text{exp}(l', \text{new}_op), \text{op}(l')), rt)
\]

In order to prove this lemma we first proved by induction on operator trees that each expression in the initial register transfer \( rt \) is transformed into itself by a sequence of \( \text{oc} \) and \( \text{ois} \) transformations. A proof for \( \text{oc}_\text{inv} \) was then derived by expanding definitions, case analysis and rewriting.

A higher-level specification for the existence of an inverse sequence for OC is stated by the \( \text{OC}_\text{inv} \) lemma. Its obvious proof makes use of the \( \text{oc}_\text{inv} \) lemma, and the identification of OIS as belonging to the operational set.

\[
\text{OC}_\text{inv: LEMMA} \quad \forall (rt: \text{wellformed}_rt, \text{new}_op: \text{operator}, l: \text{op}_i_link) : \\
\quad \text{let cg: c_graph = cons(rt, null) in} \\
\quad \text{wellfound_link}(l, \text{new}_op, rt) \Rightarrow \\
\quad \exists (s: \text{op}_\text{tr}_\text{sequence}) : \text{eq}(\text{apply_sequence}(s)(\text{OC}(cg, 0, \text{new}_op, l)), cg)
\]

The Inverse Sequence for the OIS Transformation

For a clear treatment of the OIS inverse sequence we consider three cases in the correct application of OIS, as illustrated in Figure 5.18:

1. **Case 1:** The substituting operator is not already in use in the current register transfer (in Figure 5.18.a \( OP3 \) is substituted with \( \text{Copy} \)).

2. **Case 2:** The substituting operator is used in the current register transfer, and the substituted operator has only one target expression (in Figure 5.18.b \( OP3 \) is substituted with \( OP2 \)).

3. **Case 3:** The substituting operator is used in the current register transfer, and the substituted operator has more than one target expression (in Figure 5.18.c \( OP3 \) is substituted with \( OP2 \)).
Note that for the correct application of OIS, the corresponding precondition has to be satisfied: the substituting operator must perform the same function as the substituted operator and, for the last two cases, it must have the same left and right sources.

The existence of an inverse is obvious in the first two cases: the effect of OIS is undone either by another OIS (in the first case), or by an OC transformation (in the second case). The following two lemmas formalize this statement; they were easily proved using induction on operator trees, functions expansion, propositional simplification and rewriting.
ois_1: LEMMA
forall(rt:wellformed_rt, subst:(filter_op(rt`exp_set)), opr:operator) :
  (not operators(rt)(opr)) =>
  eq?(ois(ois(rt,subst,opr), op(opr,source1(subst),source2(subst)),
   op(subst)), rt)

ois_2: LEMMA
forall(rt:wellformed_rt, subst,wth:(filter_op(rt`exp_set))) :
  (precondition_ois?(rt, subst, op(wth)) and
  singleton?(targets(rt,subst))) =>
  (let l: op_i_link = choose(targets(rt,subst)),
   l1: op_i_link = (ois.new_exp(l`1,subst,op(wth)),l`2,wth) in
  eq?(oc(ois(rt,subst,op(wth)),op(subst),l1),rt))

Remember that OC was defined for only one link (connected to one target of the copied operator instance), hence the application of OIS in the third case can not be undone by only one OC transformation. Even if OC was defined for a set of links, proving that OC is the inverse of OIS in this case would be difficult, since we should account for each target of the substituted operator. This is straightforward in an informal reasoning style, where a set of operator expressions and links can be labeled or enumerated, but very cumbersome in an abstract definitional specification. We describe here a proof strategy that deals with the abstract nature of our model. It consists of the following steps, each corresponding to a sublemma specified and proved as shown in Appendix F:

- Prove that for each register transfer rt, each substituting operator wth, and each substituted operator subst, if the precondition for OIS is satisfied and subst has more than one target, then there exists an invertible sequence of OTs that result in an equivalent register transfer where subst has one target less, such that by applying OIS to the result of this sequence has the same effect as applying OIS to the initial transfer.

∀(rt : transfer, subst, wth : operator) :
  precondition_ois(rt, subst, wth) ∧ card(targets(subst, rt)) > 1 ⇒
  ∃(s : seq) : (OIS(rt, subst, wth) eq? OIS(apply(s)(rt), subst, wth)) ∧
  (card(targets(subst, apply(s)(rt))) = card(targets(subst, rt)) − 1) ∧
  (∃(s1 : seq) : apply(s1)(apply(s)(rt)) eq? rt))

This step is illustrated in Figure 5.19 and is proved by instantiating the consequent with the following sequence of OTs: 1) choose an arbitrary link corresponding to one target of the substituted operator and apply OC to this link (OC(rt,l,subst) in Figure 5.19.b), and 2) substitute the copy operator with the substituting operator wth (OIS(rt,copy,wth) in Fig-
ure 5.19.c). This sequence is invertible, since both OC and OIS (here applied in the conditions of the second case) are proved to have an OT inverse. Also, the OIS transformation applied to the subst and wth operators in Figure 5.19.c results in the same configuration as if OIS was applied to the same operators, but in the initial register transfer.

Figure 5.19: The Sequence of OTs Used in Proving the First Step

- Using the above step, prove by induction (on the cardinality of the target set of subst) that there exists an invertible sequence of OTs that result in an equivalent register transfer where subst has only one target, such that by applying OIS to its result has the same effect as applying OIS to the initial transfer.

\[ \forall (rt : transfer, subst, wth : operator) : \]
\[ \text{precondition}_{\text{ois}}(rt, subst, wth) \land \text{card}(\text{targets}(subst, rt)) > 1 \Rightarrow \]
\[ \exists s : \text{seq} : OIS(rt, subst, wth) \equiv OIS(\text{apply}(s)(rt), subst, wth) \land \]
\[ \exists s : \text{seq} : \text{singleton?}(\text{targets}(subst, \text{apply}(s)(rt))) \land \]
\[ \exists s1 : \text{seq} : \text{apply}(s1)(\text{apply}(s)(rt)) \equiv rt \]

- Using the above step and lemma ois_2 prove that there exists an inverse of the OIS transformation correctly applied when the substituting operator is already in use in the current register transfer, irrespective of the cardinality of the target set of the substituted operator.

The ois_inverse1 lemma expresses the existence of an inverse sequence of OTs for the OIS transformation. As in the case of each transformation, a higher-level specification of the existence of inverses does not explicitly specifies what sequences of OTs form the inverse, since this is not necessary in justifying the inverse axiom.
The Inverse Sequence for the RTS Transformation

RTS admits as inverse an RTM transformation. The existence of an inverse was proved similarly to the case of the OC transformation: we first proved by induction on operator trees that each expression in the initial register transfer $rt$ is transformed into itself by a sequence of $rts$ and $rtm$ transformations. A proof for $RTS_{RTM}$ was derived by expanding function definitions, performing case analysis and rewriting.

The Inverse Sequence for the RTM Transformation

In our constructive proof for $LEMMA_{1}$ we used the RTM transformation only in the particular case when the merged register transfer (the transfer that is executed first) has only one output register. It is then sufficient to prove the existence of an inverse of RTM in this particular case.

Intuitively, the inverse sequence for RTM should contain at least one RTS (to undo the effect of merging), one RIS (to rename the temporary registers introduced by RTS), and a CO transformation (to delete any temporary registers appearing as output registers after the application of RIS). During the mechanized proof exercise, this intuitive sequence of transformations was refined in order to account for every case in the application of RTM. We found that the inverse sequences are distinct
in the following cases:

1. **Case 1**: The output of the merged register transfer reads a value from an operator expression (Figure 5.20.a). The inverse sequence consists of RTS, RIS and a CO transformation.

   \[
   \text{RTM}_1\text{\_inv: \textsc{Lemma}} \\
   \forall (\text{cg}: \text{cons\_graph}, n: \text{below}[\text{length}(\text{cg})-1]) : \\
   (\text{singleton?(nth(\text{cg}, n+1)'outregs)} \text{ and} \\
   \text{rtm\_precondition(nth(\text{cg}, n), nth(\text{cg}, n+1)}) \text{ and} \\
   \text{clean\_rt?(nth(\text{cg}, n))}) \text{ and} \\
   \text{clean\_rt?(nth(\text{cg}, n+1))}) \Rightarrow \\
   (\text{let} v:(\text{reg?}) = \text{choose(nth(\text{cg}, n+1)'outregs)}, \\
   a: \text{expression} = \text{nth(\text{cg}, n+1)'regassign(v) in} \\
   \text{op?}(a) \Rightarrow \\
   \text{let} t:(\text{reg?}) = \text{choose(nth(RTS(RTM(\text{cg}, n), n, nth(\text{cg}, n+1)'exp\_set), n+1)'outregs)} \text{ in} \\
   (\text{eq?}(\text{clean\_outputs1\_r(RIS(RTS(RTM(\text{cg}, n), n, nth(\text{cg}, n+1)'exp\_set), n, t, v), n+1, t), cg) \text{ or} \\
   \text{eq?}(RIS(RTS(RTM(\text{cg}, n), n, nth(\text{cg}, n+1)'exp\_set), n, t, v), cg)))
   \]

2. **Case 2**: The output of the merged register transfer reads a value from itself (Figure 5.20.c). The inverse sequence consists of an RTS transformation followed by \texttt{add\_output}.

   \[
   \text{RTM}_2\text{\_inv: \textsc{Lemma}} \\
   \forall (\text{cg}: \text{cons\_graph}, n: \text{below}[\text{length}(\text{cg})-1]) : \\
   (\text{singleton?(nth(\text{cg}, n+1)'outregs)} \text{ and} \\
   \text{rtm\_precondition(nth(\text{cg}, n), nth(\text{cg}, n+1)}) \text{ and} \\
   \text{clean\_rt?(nth(\text{cg}, n))}) \text{ and} \\
   \text{clean\_rt?(nth(\text{cg}, n+1))}) \Rightarrow \\
   (\text{let} v:(\text{reg?}) = \text{choose(nth(\text{cg}, n+1)'outregs)}, \\
   a: \text{expression} = \text{nth(\text{cg}, n+1)'regassign(v) in} \\
   (\text{reg?}(a) \text{ and } v = a) \Rightarrow \\
   \text{eq?}(\text{add\_outputs(RTS(RTM(\text{cg}, n), n, nth(\text{cg}, n+1)'exp\_set), n+1, a), cg}))
   \]

3. **Case 3**: The output of the merged register transfer reads a value from a register expression, distinct from itself (See Figure 5.20.b). The inverse of RTM in this case is a new transformation, Register Copy (RC). The RC transformation completes the set of operational transformations; its introduction was required due to the restricted definition.
a) The output of the second register transfer reads from an operator expression.

b) The output of the second register transfer reads from itself.

c) The output of the second register transfer reads a value from a register.

Figure 5.20: Three Cases in the Application of RTM
of RTS. The RTS formally specified as in Section 4.4 inserts a new temporary register only when the transferred value is the output of an operator expression.

Figure 5.20.c shows the particular case when the RC transformation is required as an inverse of RTM. Here, the input register of the merged register transfer also appears as an input in the first register transfer. Note that if RTS was inserting temporary registers for every link that connects components from the \textit{split set} with components outside of it, then the RC transformation would not be needed. In this thesis we used the simplified definition for RTS in order to ease the proofs.

\begin{verbatim}
RTM_3_inv: LEMMA
forall(cg:cons_graph,n:below[length(cg)-1]) :
(singleton?(nth(cg,n+1)'outregs) and rtm_precondition(nth(cg,n),nth(cg,n+1)) and
clean_rt?(nth(cg,n)) and clean_rt?(nth(cg,n+1)))=>
(let v:(reg?) = choose(nth(cg,n+1)'outregs), a:expression = nth(cg,n+1)'regassign(v) in
(reg?(a) and v /= a) =>
let lset:set[reg_i_link] =
1:reg_i_link | 1'3 = a and
(exists (l1:(r_targets(nth(cg,n),v))) : if reg?(l1'1) then l1'1 = l'1
else l'1 = new_exp(nth(cg,n),nth(cg,n+1),l1'1) and
l'2 = l1'2 endif) in
eq?(RTCOPY1(RTM(cg,n),n,lset,v,a),cg))
\end{verbatim}

\textbf{The Register Copy (RC) Transformation}

We describe in this sub-section the Register Copy (RC) transformation. As outlined before, the RC transformation was needed in order to define an inverse sequence for the RTM transformation applied in a particular case. In this particular case, the RTS transformation cannot be used to define an inverse sequence due to simplifications in its definition.

Following the same formal reasoning approach used in this thesis for the other ten transformations, RC was formally specified and proved correct in PVS. We also proved formally that the RTM transformation is the inverse of RC.

RC creates a copy of an input register instance, and a new register transfer where the only value transfer performed is from the copied register instance to the copy instance. In Figure 5.21, register \( T \) is introduced as a copy for register \( R2 \) corresponding to the link defined by the tuple \( (R2,R3) \). In order to preserve the computational behavior of the design, the value of the copied register \( R2 \) is previously transfer into the copy register, \( T \).
Figure 5.21: The RC Transformation

Inputs:
1. a control graph $cg$.
2. a register transfer $rt$ in $cg$.
3. an input register $r$ in $rt$.
4. a register $r_1$. 5. a set of links, $lset$.

Function: Creates a copy $r_1$ of $r$ for the links in $lset$, and inserts a new register transfer where the only transfer performed is from register $r$ to $r_1$.

Precondition:
1. $lset$ is a set of links connecting $r$ in $rt$.
2. $r_1$ is not alive in $rt$.

Correctness Criteria:

$RC_{-Precondition}(rt, lset, r, r_1) \Delta$

$\forall (l \in lset) : (l'2 = r \land$

$(\exists (e \in rt) : (\text{operator}(e) \land l'1 = e \land (\text{source}_1(e) = l'2 \lor \text{source}_2(e) = l'2)) \lor$

$\exists (ri \in \text{out}_\text{regs}(rt)) : (l'1 = ri \land \text{f}_\text{reg}(ri) = l'2)) \land$

$\text{not}_\text{alive}?(r_1, (rt))$)

$RC_{-Precondition}(rt, lset, r, r_1) \Rightarrow \text{comp}_\text{behavior}(cg) = \text{comp}_\text{behavior}(RC(cg, rt, l, \text{new}_\text{op}))$
Formal Specification of the RC Transformation. The RC transformation admits as inputs a set of links that have a register expression as the third projection \( lset \), a register expression \( r \) to be copied, and a register \( rc \) as the copy instance. The set of expression trees is updated recursively considering the insertion of the new register, and according to the set of links \( lset \) such that: every operator reading a value from \( r \) through a link contained in \( lset \), will read a value from a copy of it, \( rc \).

![Figure 5.22: The Effect of the RC Transformation on the Expression Trees](image)

The \( \text{rtcopy1} \) function defines the fields of the transformed register transfer: it adds the copy instance as an expression of the \( \text{exp_set} \) and updates the interconnections between the output registers and expression trees in the \( \text{exp_set} \). In order to maintain the computational behavior of the design, the \( \text{RTCOPY} \) function also inserts a new register transfer right before the transformed transfer. In this inserted transfer, \( \text{simple}_\text{rt}(rc,r) \) in the following definition, no computation is performed; the only transfer is between the copied and the copy instance.

```
rtcopy1(rt:transfer,lset:set[reg_i_link],rc,r:(reg?)) : transfer =
    (# exp_set := add(rc,new_exp_set(rt'\text{exp_set},lset,rc,r)),
        outrregs := rt'outrregs,
        regassign := LAMBDA (x:(reg?):) : if exists(l:(lset)): l'1 = x then rc
            else new_exp(rt'\text{regassign}(x),lset,rc,r) endif #)

RTCOPY(cg:list[transfer],n:below[length(cg)],lset:set[reg_i_link],rc,r:(reg?)) :
    list[transfer] = replace(cg,n,(rtcopy1(nth(cg,n),lset,rc,r),simple_\text{rt}(rc,r)))
```

\( \text{reg}_i\_\text{link}: \text{TYPE} = \{il:i_{\text{link}} \mid \text{reg?(il'}3)\} \)

\( \text{new}_\text{exp}(e:\text{expression},lset:set[\text{reg}_i\_\text{link}],rc,r:(\text{reg?})) : \)

\[
\text{RECURSIVE expression} = \\
\text{if reg?(e) then e else}
\text{(if (lset((e,left,r)) and lset((e,right,r))) then op(op(e),rc,rc)
else
(if lset((e,left,r)) and (not lset((e,right,r)))
    then op(op(e),rc,new_exp(source2(e),lset,rc,r))
else
    (if lset((e,right,r)) and (not lset((e,left,r)))
        then op(op(e),new_exp(source1(e),lset,rc,r),rc)
    else op(op(e),new_exp(source1(e),lset,rc,r),
        new_exp(source2(e),lset,rc,r)) endif)
else
    MEASURE reduce_nat ((LAMBDA (r:register) :0) ,
    (LAMBDA (op:operator), (n1:nat), (n2:nat) : n1+n2+1))(e)
```

110
The preconditions required for preserving the computational behavior are:
1. The set of links \( \text{ls}\) is well-founded, that is, it consists of tuples \([\text{expression, side, expression}]\) defined as links in the register transfer \( rt \) affected by the transformation.
2. The copy instance \( rc \) is not an input register in \( rt \).

```plaintext
wellfounded_link_set(lset:set[reg_i_link],r:(reg?),rt:transfer) : bool =
forall(l:(lset)) :
    ( rt\'exp_set(r) and instances(rt)(l'1) and l'3 = r and
    (if reg?(l'1) then rt\'regassign(l'1) = l'3
     else if l'2 = left then source1(l'1) = l'3 else source2(l'1) = l'3 endif endif))

rtcopy1_precondition(rt:transfer,lset:set[reg_i_link],rc,r:(reg?)) : bool =
    wellfounded_link_set(lset,r,rt) and not inputs(rt)(rc)
```

**Conclusion on The Inverse Sequence for RTM**

The details in determining the inverse sequence for RTM are not obvious in an informal reasoning style, and were discovered upon reaching unprovable subgoals during the mechanization of the proof. One case that we had to deal with separately during the proof is when one register appears as an input in both register transfers to be merged. We analyzed this case for the RTM and RTS transformations as defined by Vemuri [36]. Here, the RTS transformation inserts a temporary register for each link in a given set of links, \( e\)\(\text{set} \), that partitions the register transfer into exactly two connected graphs. (In comparison, the RTS defined in this thesis inserts a temporary register for every operator expression whose output value has to be saved for computations in the next register transfer). Consider the example shown in Figure 5.23. Because register \( B \) appears as an input in \( rt1 \) and \( rt2 \), the register transfer merging cannot admit a single register transfer split as an inverse. An inverse sequence in this case consists of decomposition (for the \( T2 \) output), merge the new register transfer with \( rt1 \), and substitute \( T1 \) with \( C \). We emphasize the fact that the rigorous mathematical specification and proofs conducted in PVS could not have overlooked such details.

**The Inverse Sequence for the RTD Transformation**

A proof for the existence of an inverse sequence for RTD is straightforward. We proved that RTC is the inverse of RTD using case analysis, expanding function definitions an rewriting.
The Inverse Sequence for the RTC Transformation

Since the RTD transformation was defined only for one output, the inverse of RTC can not be a simple RTD transformation. One way to define the inverse sequence for RTC would make use of RTS followed by a sequence of RIS transformations (to rename the temporary registers introduced by RTS). But this approach requires proofs that are as difficult as those derived for proving the existence of an inverse sequence for RTM. To avoid this, we define an extension of the RTD transformation, RTD\_EXT, that allows decomposition for more than one output register at a time. RTD is then a particular case of RTD\_EXT (lemma \texttt{RTD\_from\_extended\_lemma}), and RTD\_EXT is the inverse of RTC (lemma \texttt{RTC\_RTD\_EXT}).
The Inverse Sequence for the RIS Transformation

As illustrated in Figure 5.24, there are two cases in the application of RIS: 1) when the substituted register is not a primary input (it was written at a previous register transfer), and 2) when the substituted register is a primary input. In the first case, the inverse sequence consists of an RIS followed by a CO transformation. In the second case, the inverse is a simple RTM transformation. The RIS inv lemma bellow formalizes the existence of these sequences; a proof was derived using induction on expression trees, case analysis and function definitions expansion.

The Inverse Sequences for the CO and AO Transformations

The CO and AO transformations are inverses of each other. In the specifications and proofs described so far we used the clean outputs, clean output1 r and add outputs transformations, which are particular cases in the application of CO and AO. The following two lemmas formalize the existence of inverse transformations in these general cases.
Figure 5.24: The Inverse Sequences for RIS

**clean_outputs_sr_inv**: LEMMA
\[
\text{forall}(cg:\text{c_graph}, n:\text{below}[\text{length}(cg)], \text{rset}:\text{set}[(\text{nth}(cg,n)\text{'}\text{outregs})]):
\text{let } f: [(\text{rset})\rightarrow(\text{nth}(cg,n)\text{'}\text{exp_set})] = \text{LAMBDA}(x:(\text{rset})) : \text{nth}(cg,n)\text{'}\text{regassign}(x) \text{ in}
\text{eq?}(\text{add_outputs_sr}(\text{clean_outputs1_sr}(cg,n,\text{rset}),n,\text{rset},f),cg)
\]

**add_outputs_sr_inv**: LEMMA
\[
\text{forall}(cg:\text{c_graph}, n:\text{below}[\text{length}(cg)], \text{rset}:\text{finite_set}[(\text{reg}?)],
\text{f: } [(\text{rset})\rightarrow(\text{nth}(cg,n)\text{'}\text{exp_set})]) :
\text{empty?}(\text{intersection}(\text{nth}(cg,n)\text{'}\text{outregs},\text{rset})) \Rightarrow
\text{eq?}(\text{clean_outputs1_sr}(\text{add_outputs_sr}(cg,n,\text{rset},f),n,\text{rset}),cg)
\]
The Inverse Sequences for the DRT Transformation

The application of the RTS transformation when the \textit{split set} is empty has the effect of introducing a delay node in the current control step. Remember that \textit{split set} represents the set of operations (and their corresponding implementation) scheduled to be performed at the current control step. It follows that RTS applied when \textit{split set} is empty is the inverse of DRT. The \texttt{delete\_rt\_inv} lemma shown below specifies this. A proof was derived using induction on the expression trees, definitions expansion, case analysis and rewriting.

\begin{verbatim}
\texttt{delete\_rt\_inv: LEMA}
\texttt{forall(cg:cons\_graph,n:below[length(cg)-1]) :}
\texttt{empty?(nth(cg,n+1)) => eq?(RTS(delete\_rt(cg,n+1),n,emptyset),cg)}
\end{verbatim}

5.3 Summary

In this chapter we formally defined the completeness property for a finite set of RTL transformations, and presented the approach used in proving subgoals of the Completeness Theorem. Three main subgoals from which a proof for completeness can be derived were stated. The first subgoal was proved by algorithmically constructing a witness sequence to be used for instantiating of the consequent formula (Section 5.2.1). A proof for the second subgoal was derived starting from an axiom that postulates the existence of a sequence of OIS transformations that transforms one register transfer into a structural isomorphic one. The third subgoal was proved using a lemma stating that each operational transformation admits an inverse sequence of operational transformations, and instantiating the first subgoal. We presented constructive proofs for the existence of inverse transformations sequences for each particular case in the application of each transformation.
Chapter 6

Practical Uses of the Completeness Result

6.1 Checking Behavior Equivalence of Two Designs

In proving completeness we used the general idea of Vemuri [36, 34], based on the existence of a normal form structure, which is a unique implementation of a given behavior. In this thesis we define the normal form as a class of isomorphic structures satisfying a predicate that defines a Minimum Operators (MinOP) Form: a MinOP Form of a design rtl is a structure implementing the same behavior as rtl, but where all computations are performed in the same control step, and the sharing of operators is maximized. Using this notion of a normal form, one can check the computational behavior equivalence of two designs if the following propositions stand:

**Proposition 1 (Reachability).** For any design implementation there exists an algorithmically created sequence of structural changes that brings it to a MinOP form.

**Proposition 2 (Uniqueness).** All the MinOP implementations of a design belong to the same class of isomorphic structures.

**Proposition 3 (Behavior Equivalence).** If two designs can be algorithmically associated with isomorphic normal form structures, then they implement the same computational behavior.

A proof for Proposition 1 follows from the following two formal results:

1) According to LEMMA_1 (See Section 5.2.1), for any design implementation there exists a sequence of operational transformations that brings it to an equivalent design where all operations are performed
in the same control step:

\[
\forall (rtl : c\_graph) : (\exists (s1 : ot\_seq) : length(apply(s1)(rtl)) = 1)
\]  

(1)

2) According to the \texttt{cg\_to\_minform} lemma proved as shown in Section 5.2.3, for any design implementation consisting of only one register transfer (all operations are performed in the same control step) there exists a sequence of operational transformations that brings it to its equivalent MinOP form.

\[
\forall (rtl : c\_graph) : length(rtl) = 1 \Rightarrow \exists (s2 : ot\_seq) : MinOF(apply(s2)(rtl))
\]  

(2)

where \(s1\) and \(s2\) are sequences of operational transformations, and \(MinOF\) denotes the predicate that defines the minimum operators normal form structure. The above two lemmas were proved using a constructive approach where the witness sequence used to instantiate the consequent formulas were defined by a terminating algorithm. The concatenation of these sequences represent a witness for proving reachability.

Proposition 2 states the uniqueness of the MinOP form up to isomorphism, and is expressed by the \texttt{iso\_relation} lemma in Section 5.2.3, which states that if two designs implement the same behavior description and are in the MinOP form then they have isomorphic structures:

\[
\forall (rtl1, rtl2 : c\_graph) : (length(rtl1) = length(rtl2) = 1 \land
\]

\[
\text{behavior}(rtl1) = \text{behavior}(rtl2) \land
\]

\[
(\text{MinOP}(rtl1) \land \text{MinOP}(rtl2))
\]

\[
\Rightarrow < rtl1 > is_R < rtl2 >
\]  

(3)

The \texttt{minf\_to\_beh} lemma shown bellow asserts that any two structurally isomorphic expression trees implement the same extracted behavior, and it was easily proved by induction on operator trees.

\begin{verbatim}
minf_to_beh: LEMMA
forall(e1,e2:expression) :
(exists(f:[[cone(e1)]->[cone(e2)]]) :
bijective?(f) and f(e1) = e2 and
(forall(e:(cone(e1))) :
    (op?(e) => (op?(f(e)) and opfn(op(f(e))) = opfn(op(e)) and
            f(source1(e)) = source1(f(e)) and
            f(source2(e)) = source2(f(e))) and
    (reg?(e) => (reg?(f(e)) and f(e) = e))) =>
        extracted_behavior(e1) = extracted_behavior(e2)
end(minf_to_beh)
\end{verbatim}
Proposition 3 extends this result to MinOP forms, which are register transfers, i.e., collections of output registers reading values from expression trees. The contrapositive of \texttt{minf to beh} lemma states that two operator expressions implementing different behaviors cannot have isomorphic structures, hence two designs implementing different behaviors have different normal forms.

The behavior equivalence checking methodology based on the above results would consist of two steps: in the first step the two designs to be compared are brought to the MinOP normal form. If the corresponding MinOP forms are structurally isomorphic, then the two designs are behavior equivalent. Though sound, this approach is not justified by its complexity. The algorithm used for deriving the normal form structure is seemingly more complex than the symbolic behavior extraction along operator trees in successive register transfers. Extracting the computational behavior from each design and then comparing is more efficient. In the next section we present a brief analysis on the upper bound on the length of transformations sequences that bring a design implementation to its MinOP normal form. This bound gives an idea about the time complexity of an equivalence checking methodology based on the uniqueness of a normal form, as described above.

6.2 Complexity Analysis of the Algorithm Used for Deriving the MinOP Normal Form

A sequence of transformations that results in the MinOP form is the concatenation of two sequences: a sequence $S_1$ that transforms a design into a behavior equivalent implementation where all computations are performed in the same control step (as constructed in order to prove \texttt{LEMMA 1}), and a sequence $S_2$ of OIS transformations used to maximize sharings of operators (as constructed to prove \texttt{cg to minform} lemma in Section 5.2.3). To express the length of these sequences we use the following notations:

- $N_{IR}$ is the average number of input registers that appear in a register transfer.
- $N_{OR}$ is the average number of output registers that appear in a register transfer.
- $N_{OP}$ is the total number of operators in the design.
- $N_{EXP}$ is the average number of operators used in a register transfer.
- $N$ is the average number of operators that appear in an expression tree.
- $L$ is the length of the control graph associated with the design.
With these notations, the upper bound on the length of the transformations sequence that derives the MinOP form is:

\[ ST_{length} = S_1 + S_2 = (L - 1) \cdot ((N_{IR}(RIS) + N_{OR} \cdot ((RTD) + N(OIS) + (RTM/RTC)) + DRT) + N_{OP}(OIS) \]  \tag{4} \]

where in round brackets we indicate the transformation applied. The length of \( S_1 \) was derived from the definitions of each step used in proving the first subgoal (See Section 5.2.1):

1. In the worst case, the \textit{preconditioning for RTD} step applies the RIS transformations for each input register in the second register transfer, and adds the \( N_{IR}(RIS) \) term in formula (4).

2. In the second step, for one output register of the second register transfer the following sequence of transformations are applied: decomposition (RTD), substitution for each operator in the generated transfer (\( N(OIS) \)), and combining the first two register transfer through merging (RTM) or composing (RTC). This step adds the \( (RTD) + N(OIS) + (RTM/RTC)) \) term in formula (4).

3. The previous sequence is repeated for each output of the second register transfer, until it becomes \textit{empty} \( N_{OR} \cdot ((RTD) + N(OIS) + (RTM/RTC)) \).

4. Delete the empty register transfer (DRT).

5. Repeat the previous steps until the length of the graph becomes equal to one \( (L - 1) \cdot ((N_{IR}(RIS) + N_{OR} \cdot ((RTD) + N(OIS) + (RTM/RTC)) + DRT)) \).

To express the time complexity of the algorithm defined for bringing an RTL design to its normal form, we consider the length \( L \) and the average number of operators used in a register transfer, \( N_{EXP} \) as measures of the design complexity. Both measures can be seen as performance constraints imposed by the user on an RTL implementation: \( L \) is the maximum latency, and \( N_{EXP} \) expresses an area constraint. In this complexity analysis we consider a binary tree representation for expression trees, and an array (or list) representation for sets of registers and sequences of register transfers. With the exception of RIS, all transformations can be considered atomic (executed in constant time). To execute RIS, a search through the control graph array is performed in order to find the last register transfer where a value was written in the substituted register instance. In the worst case, the entire graph is searched. The number of input registers in a register transfer, \( N_{IR} \), is at most double the average number of operators \( N_{EXP} \), and the number of output registers in a register transfer, \( N_{OR} \), equals in the worst case \( N_{EXP} \). On average, \( N_{OR} \) is the number of
operators $N_{EXP}$ divided by the average number of operators in an expression tree, $N$. With these assumptions, the complexity of the algorithm becomes proportional to:

$$\zeta_{S_1} \sim \sum_{i=1}^{L}(N_{IR} \cdot i + N_{OR} \cdot (1 + N + 1) + 1)$$
$$\sim \sum_{i=1}^{L}(2 \cdot N_{EXP} \cdot i) + N_{OR} \cdot N + N_{OR}$$
$$\sim (L - 1)(L - 2) \cdot N_{EXP} + (L - 1) \cdot N_{EXP} + 2 \cdot \frac{N_{EXP}}{N}$$  \hspace{1cm} (5)

For a given area constraint (fixed $N_{EXP}$), the algorithm has quadratic complexity in terms of latency $L$. If we consider the total number of operators in the implementation, $N_{OP}$, as a measure for the size of the design, an average latency is $L = \frac{N_{OP}}{N_{EXP}}$, and the complexity is interpreted as quadratic in the total number of operators ($\zeta_{S_1} \sim \frac{N_{OP}^2}{N_{EXP}} + N_{OP}$).

The complexity of an algorithm that brings an expression tree to its normal form (corresponding to sequence $S_2$), is on average quadratic in $N_{OP}$, since any visit of an operator node in an expression tree is followed by a search for a node having the same sources and function. If a collision-free hash table could be used for storing such information on every node in the expression trees, the complexity may become linear. The trade-off between time and memory complexities is beyond the scope of this analysis. Sufﬁces it to say that the complexity of an algorithm that brings an expression tree to its normal form does not justify its use in an equivalence checking methodology.

On the other hand, extracting the computational behavior of a design also has a quadratic complexity in the length of the graph. Within a register transfer, the behavior extraction is linear in the number of operator nodes appearing in that transfer ($N_{EXP}$). The quadratic term is introduced by the search along the control graph list array for each input register. The complexity of the behavior extraction is:

$$\zeta_{BE} \sim \sum_{i=1}^{L}(N_{IR} \cdot i + N_{EXP}) = (L - 1)(L - 2) \cdot N_{EXP} + (L - 1) \cdot N_{EXP}$$  \hspace{1cm} (6)

where the first term in the sum expresses the search in the control graph for every input register, and the second expresses the behavior extraction within a register transfer. Similar complexities for extracting behavior and for the algorithm defined in the first subgoal were expected, since the former traverses the expression trees and the control graph, while the latter performs the somehow opposite: it unrolls the control graph list and brings all computations in the same register transfer, where computational behavior equivalence is extracted in linear time.

Overall, the behavior extraction approach is better: although the transformations (with the exception of RIS) are performed in constant time, they are more complex than checking if a register was written in a register transfer (required by the search performed for extracting behavior). We conclude that a transformational approach is not efficient for checking behavior equivalence of designs. The advantages of this approach lie in the completeness property, which can be better employed
in checking the correctness of a synthesis software implementation, as it will be seen in the next section.

6.3 Using the Completeness Property for Synthesis Software Debugging

A high-level synthesis (HLS) system traditionally consists of three tasks: 1) a scheduling task, where operations are assigned to control steps, 2) an allocation task that chooses functional units and storage elements from a library of components, and 3) a binding task which assigns operations to function units, variables to registers and data transfers to wires and buses. The synthesis result is constrained by user imposed measures, like area or speed. Complex heuristic algorithms are used for each task in order to fulfill these performance constraints.

The correctness of the HLS algorithms implementations is crucial for the synthesis result correctness and constitutes the subject of many research efforts [17]. We show in this section how the completeness property for a set of RTL transformations can be used in debugging HLS algorithms implementations. Informally, if a complete set of transformations exists, then any synthesis result can be obtained by applying a finite sequence of correct RTL transformations that have the same effect as the synthesis algorithms. If a witness sequence of correct transformations cannot be identified, then the synthesized design is incorrect. Information about what led to an incorrect synthesis result can be obtained by analyzing the attempted witness sequence derived. In this section we give some insights on the systematic identification of such witness sequences of RTL transformations.

Completeness for the Scheduling Step. Radhakrishnan et al. [24] propose a Witness Generation (WG) synthesis software debugging methodology. The approach is used for debugging the scheduling algorithms implementation by attempting to identify a sequence of correct Register Transfer Split (RTS) transformations that have the same effect as the algorithm used. The main advantage of this method is that identifying such sequences of transformations can be done by examining the binding data structures supplied by the synthesis tool, without any knowledge about the synthesis algorithm used to perform the task.

Remember that the RTS transformation is applied to a register transfer \( rt \) of a control graph, and a subset, \( \text{split\_set} \), of operator expressions in \( rt \). RTS replaces \( rt \) with a sequence of two register transfers: in the first transfer the result of the computations performed by operators in \( \text{split\_set} \) are saved into intermediate registers; the second register transfer contains operators in \( rt \) that are not in \( \text{split\_set} \) and correspond to computations deferred to the next control step. RTS was proved to be correct if and only if the following precondition is satisfied:
In the WG method, each entry in the scheduling table is automatically identified with an RTS transformation and recorded as such in the witness sequence. At the same time, the transformation is also applied to an initial design implementation where it can be checked if its precondition is satisfied, and if not, record the failed precondition. This failed precondition gives information about an error in the synthesis software. The algorithm that implements the WG method is:

**Algorithm:** Scheduling WG

**Input:** S: scheduling table, ID: initial design

**Output:** WS: witness sequence

```plaintext
WS = null;
foreach (cs = 0; cs + 1; cs < MAXSTEP)
    OP = getOperators(S[cs]);
    if RTS_Precondition(OP,D) == false
        then printError(OP);
        D = UpdateDesign(D,RTS(OP))
        WS = add(RTS(OP), WS);
endfor.
```

Each control step in the scheduling table $S$ is associated with an RTS transformation applied to the set $OP$ of operators that appear in the register transfer executed at control step $cs$. If the precondition for RTS is not satisfied ($RTS\_Precondition(OP,D) == false$), then an error was encountered. Otherwise, RTS is applied to the initial design $D$, on which a new identification of a new RTS transformation is attempted.

This algorithm is based on the following formal results, and on the correctness of the RTS transformation based on a weakest precondition. Their formalization and mechanized proofs are presented in Appendix G.

1) Any schedule $s$ of a behavior description $bd$ can be derived from an initial schedule of $bd$ through a sequence of RTS transformations. An initial schedule assigns all computations to the same control step. In order to derive any schedule from any different one, a merge operation should be added to
the split operation to form a complete set:

\[
\forall (bd : beh\_description, sched : schedule[bd]) : \\
\exists (s : seq(RTS)) : apply\_seq(s)(init\_schedule(bd)) = sched
\]

(9)

2) Any schedule \( s \) of a behavior description \( bd \), either has length equal to one (the initial schedule), or there exists a unique schedule \( s_1 \) from which \( s \) can be derived by a split transformation applied to its last control step. The split set is also unique:

\[
\forall (bd : beh\_description, sched : schedule[bd]) : \\
\max\_step(sched) = 1 \lor \\
\exists (s_1 : sched[bd], split\_set : set[expression]) : s = RTS(s_1, max\_step(s_1), split\_set)
\]

(10)

where schedule defines a function type assigning to each operation in the behavior description a natural number corresponding to its scheduled control step, and \( \max\_step \) is the maximum control step.

Hence, any correct schedule of a behavior description can be obtained by applying a unique sequence of correct split transformations, such that each transformation is applied at the last control step. In conclusion, the algorithm Scheduling WG is sound.

Note that the linear complexity of the WG algorithm is possible only when the intermediate binding data structure supplies information about the labeling of operators before and after the high-level synthesis step considered for debugging.

**Completeness for the Registers Binding Step.** Existing register optimization algorithms, such as left edge and clique partitioning, can be expressed as sequences of Register Instance Substitution (RIS) transformations. However, the completeness of the RIS transformation for any possible register optimization algorithm cannot be formally established for two reasons: first, the precondition for RIS is not the weakest precondition, and second, is hard to give a precise formal definition of what a register optimization task would do. Consider the example in Figure 6.1 which shows a small design before and after register optimization.

Initially, \( B \) was alive until the last control step, hence unavailable for any reuse during the first and second control step. After saving the value of \( B \) into register \( D \) in the first control step, \( B \) becomes available for reuse in the second control step. A sequence of transformations that derives Design B from Design A is (See Figure 6.2):

\(< RIS(rt2, E \rightarrow A), RIS(rt1, B \rightarrow D), RTC(rt1^a, rt2), RTD(rt2, C), RTC(rt2^a, rt1), RIS(rt1, C \leftrightarrow B) >\)

To identify a witness sequence of transformations as defined by Vemuri [36, 34] is even more complicated because the only way to obtain the addition of an output register is through merging and then splitting two register transfers that have a common input (as explained in Section 5.2.4).
This example shows that RIS is not complete for any possible register binding result. We conclude that in order to determine a complete set of transformations for the register binding task, one needs information about the algorithm used to perform the optimization. For widely used algorithms, such as the left-edge or clique partitioning algorithms, the software implementation can be checked using only RIS.

6.4 Summary

In this chapter we discussed on the practical uses of the correctness and completeness formal result. Using the notion of a unique normal form structure for RTL designs we formally defined a transformational methodology for checking behavior equivalence of two designs. A complexity analysis for this methodology and for a proven correct algorithm that brings a design to its normal form was presented and compared against the complexity for extracting behavior. We concluded that, even if not significantly more complex, the transformational methodology for checking design equivalence is not justified. A more interesting application of the completeness result refers to debugging the software implementations for HLS algorithms. The debugging methodology is based on the systematic identification of witness sequences of correct transformations that emulate the effect of synthesis algorithms. We reasoned about the completeness of sub-sets of transformations for particular synthesis tasks: scheduling and register binding.
Figure 6.2: The Sequence of Transformations that Derives Design A from Design B
Chapter 7

Conclusions

We presented here a formal treatment of correctness and completeness for a set of RTL transformations. The formal specification and verification effort is conducted within the PVS [7] theorem proving environment.

Abstract formal models were defined for RTL designs and behavior descriptions. The goal in defining the abstract models was to find a minimal specification that captures all the important features of a design while abstracting out the irrelevant details. An overspecified model would considerably complicate the proofs for the more complex tasks of this work: defining and proving correctness and completeness for a set of transformations. On the other hand, an underspecified model might admit RTL descriptions that should not be allowed in practice.

Based on these models we further specified a set of ten uninterpreted RTL transformations, and proved that each transformation preserves the behavior of the design provided a set of preconditions is satisfied. In specifying the transformations we used a definitional approach, which is more secure and descriptive than an axiomatic approach. 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 and considerably reduced the proof effort.

A constructive proof for completeness was mechanically derived within the PVS proving environment. The proof is constructive in the sense that a finite sequence of transformations (belonging to the complete set), which transforms a certain design implementation into another one having the same behavior, is algorithmically created.
The formal specification and proof mechanizing effort is considerable, due to the strictness of the formal reasoning system used, and to the complexity of the proof goal. But the reward compensates the effort spent:

- Any mechanized proof that ends in a Q.E.D is incontestable, it does not need to be carefully checked by others, as an English description would do.

- By formally defining an abstract RTL design and imposing restrictions on its structure, we actually defined the class of designs for which the proved properties stand; the interplay between specification and verification, using mainly the PVS Type-Checker, were iterative refinements on an initially underspecified abstract model.

- The rigorousness of a typed formal language which allows a definitional approach specification style is not only a safe notation, but it can actually help in a more succinct and clear specification of the invariants.

- The mechanized approach revealed some particular cases in transformations’ application that were overlooked in a paper-and-pencil reasoning style. The precise definitional specification approach, as well as the rigorous mechanized proof, are able to account exhaustively for each particular case, while in the paper-and-pencil style we tend to subjectively consider only some cases.

As a measure of the proof effort involved, we show in Table 7.1 the number of formulas proved for each goal. The proof time is the CPU time taken to execute each proof script on a Sun Sparc processor operating at 296 MHz with a 384MB RAM.

### 7.1 Practical Uses of the Presented Work

1. A *transformational derivation system* based on this core set of RTL transformations will yield correct-by-construction designs, *if correctly implemented*. The completeness property of this set ensures that any implementation of a given behavior can be reached using only transformations from this set. By this we understand that only a limited and small number of transformations need to be implemented by this system.

2. A *design validation* methodology that checks the implementations’ correctness by identifying correct applications of sequences of elementary transformations is based on the completeness property for the set of transformations to be identified.
<table>
<thead>
<tr>
<th>Proving Goal</th>
<th>No. of formulas</th>
<th>Proof time (cpu) (sec)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Model Properties</td>
<td></td>
<td></td>
</tr>
<tr>
<td>register transfers</td>
<td>63</td>
<td>75.09</td>
</tr>
<tr>
<td>control graphs</td>
<td>144</td>
<td>156.92</td>
</tr>
<tr>
<td>extracted behavior</td>
<td>45</td>
<td>197.79</td>
</tr>
<tr>
<td>Soundness of</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Transformations</td>
<td></td>
<td></td>
</tr>
<tr>
<td>OC</td>
<td>26</td>
<td>41.93</td>
</tr>
<tr>
<td>OIS</td>
<td>26</td>
<td>42.19</td>
</tr>
<tr>
<td>RTS</td>
<td>40</td>
<td>145.74</td>
</tr>
<tr>
<td>RTM</td>
<td>26</td>
<td>197.98</td>
</tr>
<tr>
<td>RTD</td>
<td>15</td>
<td>34.74</td>
</tr>
<tr>
<td>RTC</td>
<td>18</td>
<td>40.02</td>
</tr>
<tr>
<td>RIS</td>
<td>88</td>
<td>1347.79</td>
</tr>
<tr>
<td>CO</td>
<td>20</td>
<td>43.43</td>
</tr>
<tr>
<td>AO</td>
<td>15</td>
<td>61.28</td>
</tr>
<tr>
<td>DRT</td>
<td>14</td>
<td>52.42</td>
</tr>
<tr>
<td>Completeness</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Proof</td>
<td></td>
<td></td>
</tr>
<tr>
<td>First Subgoal</td>
<td>166</td>
<td>688.82</td>
</tr>
<tr>
<td>Second Subgoal</td>
<td>50</td>
<td>273.74</td>
</tr>
<tr>
<td>Third Subgoal</td>
<td>81</td>
<td>349.88</td>
</tr>
<tr>
<td>Total number of formulas and total proof time</td>
<td>837</td>
<td>3479.04</td>
</tr>
</tbody>
</table>

Figure 7.1: Table 1

### 7.2 Limitations and Future Work

There are several limitations of our work that we intend to address in the future:

1. We consider only implementations for behavior descriptions consisting of straight-line code blocks. 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. The set of RTL transformations presented here can be extended in order to deal with conditional constructs.

2. We do not guarantee that the preconditions used in proving correctness of the core set of transformations are *weakest* preconditions. This affects the effectiveness of a software debugging methodology based on witness generation: if it is possible to derive a sequence of correct transformations which have the same result on an initial implementation as the synthesis tool,
then the synthesized design is correct. Otherwise, if the precondition of a transformation is
violated, the synthesized design might still be correct. However, an explicit warning stating
the precondition violated, and the specific elements of the design for which this precondition
failed, can help in diagnosing a possible error in the software implementation. For some of
the transformations we were able to find a weakest precondition. The weakest precondition
found for the RTS transformation is used in defining a witness sequence of transformations
that emulates the effects of any scheduling algorithm (as explained in Section 6.3).

3. A synthesized RTL implementation consists of a datapath and a controller. Each component
in the data path is obtained by instantiating operators and registers with corresponding
modules from a cell library. The instantiation is done according to the operations performed
by operators and to the *bit-width* of the input and output variables. In the formal models
presented here we implicitly considered that all variables are correctly mapped to operators
and registers inputs and outputs. However, there are important practical issues which were
overlooked by this simplification. Consider the example in Figure 7.2.

![Figure 7.2: Implementation of Addition Considering the Bit-Width](image)

An addition operation performed on two input variables of bit sizes 7 and respectively 5, and
returning an 8 bit variable is implemented using an *8-bit* adder in Figure 7.2.a, and using two
*4-bit* adders in Figure 7.2.b. A correct implementation considers the mapping of variables
to hardware components: in Figure 7.2.a, the input variables should be mapped to the least
significant bits of the adder; for the implementation in Figure 7.2.b is not only the mapping of
variables that matters, but also a correct interconnection between the two adders through a
*carry* signal. In our models the operators are atomic, hence the implementation in Figure 7.2
is not allowed.
4. 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. An operator simply accepts as inputs two uninterpreted variables and returns an uninterpreted variable. The correctness of a transformation is thus defined with respect to a relatively strict structural similarity between two RTL designs. This will limit the optimization capability of a synthesis tool based on our set of transformations; optimizations that make use of properties such as commutativity or associativity of arithmetic operations are not considered.
Appendix A

Brief Introduction to PVS

PVS (Prototype Verification System) is a specification and verification environment based on classical typed higher-order logic. It consists of a specification language integrated with support tools, and a theorem prover.

There are several important issues that determined us to use the PVS environment. First, the problem we are dealing with is inherently higher-order: we prove completeness for a set of RTL transformations. The RTL transformations are defined as functions operating on abstract representations for RTL designs. The completeness property is a predicate defining a complete set of functions. Second, as an alternative to an error prone axiomatic approach, we opted for a definitional specification methodology. Although axioms may be introduced freely in PVS, its rich language also allows a wide variety of definitional constructs. Being based on a typed logic, PVS language makes use of constructs for which typechecking can guarantee conservative extension.

A.1 PVS Language

PVS specifications are organized in theories. Each theory represents the scope for the names of types, constants, axioms, definitions, and theorems introduced in the theory. A theory can be parametric in certain types, or it can be built on other theories. For example, in theory rt (Section 2.3) we specify operator trees as binary trees where nodes are of type operator and leaves are of type register. Binary trees are specified in a separate theory (expression in Section 2) parameterized on the types for nodes and leaves.
Declarations are used to introduce types, constants, variables, formulas, judgments and conversions. The declarations introduced in one theory can be referenced in another using IMPORTING or EXPORTING keywords. In the remaining of this section we summarize some of the declaration constructs we used in the specification exercise. For a full description of the PVS language we refer the reader to the PVS Language Reference [38].

1. **Type Declarations.** We illustrate through examples the different PVS type declarations we used.

   • **Uninterpreted Types:**
     
     ```
     register: TYPE+ %nonempty type
     wellformed\_rt : TYPE = {rt:transfer \mid wellformed\_rt?(rt)} % predicate subtype
     ```

   • **Function Types:**
     
     ```
     transform : TYPE = [c\_graph->c\_graph]
     ```

   • **Tuple Types:**
     
     ```
     link : TYPE = [expression, side, expression]
     ```
• **Record Types:**

```plaintext
transfer : TYPE = [# exp_set:finite_set[expression],
               outregs:finite_set[(reg?)],
               regassign:[(reg?)->expression] #]
```

• **Enumeration Types:**

```plaintext
side : TYPE = {left,right}
```

• **Datatypes**

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

• **Dependent Types:**

```plaintext
schedule(e_set: set[expression]) : TYPE+ = [(e_set)->nat]
```

2. **Variable Declarations** introduce new variables and associate a type with them. Variable declarations also appear bound to universal quantifiers (FORALL and EXISTS) or lambda expressions. Such local declarations shadow any previous ones:

```plaintext
e: VAR expression
```

```plaintext
sources_in_set: LEMMA
  forall (e_set:(wellformed_set?),e:(filter_op(e_set))):
    e_set(source1(e)) and e_set(source2(e))
```

3. **Constant Declarations.**

```plaintext
one: int = 1
```

4. **Recursive Definitions:**

```plaintext
size(cg:list[transfer]) : RECURSIVE nat =
  if null?(cg) then 0 else size(car(cg)) + size(cdr(cg)) endif
MEASURE length(cg)
```

5. **Library Declarations** introduce a new PVS context into a specification. When such a declaration is encountered, the system checks if the specified library exists and that it has a PVS context file (.pvscontext).

```plaintext
specs_lib : LIBRARY = "../specs/
IMPORTING specs_lib@extracted_behaviors1
```

133
A.2 PVS Prove Checker

The PVS Prove Checker [39] is interactive and highly mechanized. At the same time, it provides automated procedures to assist the user with the more tedious parts of the proof. The automated features of the PVS prover include decision procedures for arithmetic and equality, a BDD-based propositional simplifier, efficient hash-based automatic conditional rewriting, and induction.

The PVS prover is based on a sequent-style proof representation consisting of an antecedent, as a conjunction of formulas, and a consequent, as a disjunction of formulas. The user interactively applies *proof commands* that result in constructing a *proof tree* for the proof goal. The proof is successful if the generated proof tree is complete, that is, all its leaves are recognized as true.

A proof command either recognize the current sequent as true (and terminate the corresponding branch of the proof tree), or it adds more child nodes to the current sequent. The result of applying a proof command is called a *proof step*. Combinations of such proof steps form *strategies*.

The commands implemented by the PVS proof checker can be classified as:

- Help (*help*).
- Annotation (e.g., *comment*, *label*, *unlabel* and *with-labels*).
- Control (e.g., *fail*, *postpone*, *quit*, *rewrite-msg-off*, *rewrite-msg-on*, *trace*, *undo*, etc).
- Structural rules (e.g, *copy*, *delete*, *hide*, *reveal*).
- Propositional rules (e.g., *case*, *iff*, *flatten*, *split*, *bddsimp*).
- Quantifier rules:
  1. Existential (e.g., *instantiate*, *inst*, *inst?*).
  2. Universal (e.g., *skolem*, *skosimp*, *detuple-boundvars*, *generalize*).
- Equality rules (e.g., *beta*, *replace*, *name*, *case-replace*).
- Rules for Definitions and Lemmas:
  1. Definition expansion (*expand*, *expand*).
  2. Introducing lemmas (*lemma*, *use*, *forward-chain*).
  3. Rewriting with definitions and lemmas (e.g., *rewrite*, *rewrite-lemma*).
- Extensionality rules (e.g. *apply-eta*, *eta*, *extensionality*, *replace-eta*).
• Induction rules (e.g., induct, induct-and-simplify, rule-induct, measure-induct).

• Rules for simplification using decision procedures and rewriting (e.g., assert, bash, ground, simplify, grind, do-rewrite).

• Installation and removal of rewrite rules (e.g., auto-rewrite, stop-rewrite-theory).

• Making type constraints explicit (typepred, typepred!).

• Model checking (model-check, musimp).

• Converting a strategy to a rule (apply).

• Strategies (e.g., else, if, let, branch, rerun, repeat).

PVS supports construction of readable proofs. An interactively generated proof is saved into a readable file that can be further edited and rerun. This feature was very useful in our proof exercise: uniform specifications for each RTL transformations allowed a uniform treatment of the well-formedness and correctness proofs. The proofs completed interactively for one transformations were modified and rerun for other transformations.

A.3 PVS Typechecker and the Predicate Subtypes

In the PVS environment the theorem prover and the typechecker interacts such that proof obligations generated upon typechecking (type correctness condition, or shortly TCCs) can be automatically proved and discharged. The TCCs can also appear as subgoals during proof checking; the typechecker is invoked for every expression or quantifier instantiation introduced by the user.

Predicate subtypes are language constructs that associate a property or a predicate with a subtype. Their use help not only in writing expressive specifications but, when associated with the typechecker, in expressing invariants. We found this feature especially useful in our specifications and proofs: the well-formedness, correctness and completeness invariants were automatically expressed as TCCs upon typechecking the specification files where sequences of correct operational transformation were defined (Section 5).
Appendix B

Proof for the General Lemma

eb\_cg\_cg\_reg

The general lemma was proved by appropriately instantiating the following two sublemmas (corresponding to (b) and respectively (a) in the figure above):

---

### eb\_cg\_cg12\_reg: LEMMA

```
forall((r:register, cg1,cg2,cge: c_graph)) :
  extracted_behavior(r, cg1) = extracted_behavior(r, cg2) =>
  extracted_behavior(r, CG1a) = extracted_behavior(r, CG2a)
```

---

### eb\_cg\_cg\_reg: LEMMA

```
forall((r:register), cg1,cg2,cgs:c_graph)) :
  r appears in cg AND
  extracted_behavior(r,cg1) = extracted_behavior(r,cg2) =>
  extracted_behavior(r, CG1b) = extracted_behavior(r, CG2b)
```
The following excerpt from the PVS proof checker shows the sequent for proving the General Lemma from the above sublemmas, after skolemization and simplification of the consequent. The first two antecedent formulas correspond to the two sublemmas above. The third formula becomes part of the hypothesis after applying skolemization and simplifications on the proof goal.

Repeatedly Skolemizing and flattening, this simplifies to:
\[ eb_{cg \_cg \_cg} : \]

\[ \begin{align*}
\{-1\} \ & \text{FORALL } (r: (reg?), cg1, cg2, cg: c\_graph): \\
& \quad \text{extracted\_behavior}(reg(r), cg1) = \text{extracted\_behavior}(reg(r), cg2) \\
& \quad \Rightarrow \\
& \quad \text{extracted\_behavior}(reg(r), append(cg1, cg)) = \\
& \quad \text{extracted\_behavior}(reg(r), append(cg2, cg)) \\
\{-2\} \ & \text{FORALL } (cg1, cg2: c\_graph, cg: cons\_graph): \\
& \quad (\text{FORALL } (r: (registers(cg))): \\
& \quad \quad \text{extracted\_behavior}(reg(r), cg1) = \\
& \quad \quad \text{extracted\_behavior}(reg(r), cg2)) \\
& \quad \Rightarrow \\
& \quad (\text{FORALL } (ri: (registers(cg))): \\
& \quad \quad \text{extracted\_behavior}(reg(ri), append(cg, cg1)) = \\
& \quad \quad \text{extracted\_behavior}(reg(ri), append(cg, cg2))) \\
\{-3\} \ & \text{FORALL } (r: (registers(cgs!1))): \\
& \quad \text{extracted\_behavior}(reg(r), cg1!1) = \\
& \quad \text{extracted\_behavior}(reg(r), cg2!1) \\
\end{align*} \]

[1] \[ \text{extracted\_behavior}(reg(ri!1), append(cgs!1, append(cg1!1, cge!1))) = \]
\[ \text{extracted\_behavior}(reg(ri!1), append(cgs!1, append(cg2!1, cge!1))) \]

The proof derived for \( eb_{cg \_cg \_cg} \) using appropriate instantiation is:

\[ (\text{eb}_{cg \_cg \_cg} \quad "\quad (SKOSIMP*) \\
\quad ("" \quad (LEMMA "eb_{cg \_cg\_cg2\_reg}"
\quad ("" (INSTANTIATE -1 (append(cg1!1, cge!1)) "append(cg2!1, cge!1)" "cgs!1"))
\quad ("1" (ASSERT)) ("1" (SPLIT) ("1" (INST -1 "ri!1") NIL NIL)
\quad ("2" (LEMMA "eb_{cg \_cg\_reg}"
\quad ("2" (SKOSIMP*)
\quad ("2" (INSTANTIATE -1 ("r!1" "cg1!1" "cg2!1" "cge!1")))
\quad ("2" (ASSERT)) ("2" (HIDE 2) ("2" (INST -1 "r!1") NIL NIL)
\quad NIL))NIL)NIL))NIL)))NIL))NIL))NIL))NIL)
\quad "2" (ASSERT) ("2" (EXPAND "append" 2 1) ("2" (EXPAND "append" 2 2)
\quad ("2" (LEMMA "eb_{cg \_cg\_reg}"
\quad ("2" (INST -1 "r!1" "cg1!1" "cg2!1" "cge!1"))
\quad ("2" (ASSERT)) ("2" (INST -1 "r!1") NIL NIL)
\quad NIL)))NIL))NIL)))NIL)))NIL))NIL)))NIL))NIL)
\]

In what follows we described the strategies used in proving these two sublemmas.
B.1 Proof for lemma eb_cg_cg12_reg

The extracted_behavior function applied to an expression (operator or register) is defined recursively on operator trees and on the length of the control graph. The extracted behavior of a register instance is non-inductively defined using the definition for extracting the behavior for expressions (see definitions on Chapter 2.3.4). In order to prove any property about extracted behaviors we need to induct on both induction variables used: expressions and length of the graph. It follows that a sublemma which refers to the extracted behavior of expressions from which output registers read their values, has to be proved first. This lemma states that any operator or register expressions belonging to the first register transfer of cg has the same extracted behavior in the control graphs obtained by appending cg to cg1 and respectively cg2 if all registers in the “header” graph have the same behaviors in cg1 and cg2.

\[
\text{eb_cg_cg12: LEMMA} \\
\text{forall(cg1, cg2:c_graph, cg:cons_graph)} : \\
(\text{forall(r:(registers(cg)))}) : \\
\quad \text{extracted_behavior(reg(r), cg1) = extracted_behavior(reg(r), cg2)} => \\
\quad (\text{forall (e:(car(cg)`exp_set))}) : \\
\quad \text{extracted_behavior(append(cg,cg1),e) = extracted_behavior(append(cg,cg2),e))}
\]

\[
\text{forall(cg1, cg2:c_graph, cg:cons_graph)} : \\
(\text{forall(r:(registers(cg)))}) : \text{extracted_behavior(reg(r), cg1) = extracted_behavior(reg(r), cg2)} => \\
(\text{forall(ri:(registers(cg)))}) : \text{extracted_behavior(reg(ri), append(cg, cg1)) = extracted_behavior(reg(ri), append(cg, cg2)))}
\]

- case analysis
  1. null?(match(r,cg))
     - extracted_behavior(reg(ri), append(cg, cg1)) = extracted_behavior(reg(ri), cg1)
     - extracted_behavior(reg(ri), append(cg, cg2)) = extracted_behavior(reg(ri), cg2)
  2. cons?(match(r,cg))
     - apply lemma eb_cg_cg12
     - instantiate cg with match(r,cg)
     - instantiate e with e1 = car(match(r,cg))’regassign(r)
     2.1
     2.2 (TCC) car(match(r,cg))’exp_set(e1)
        - use wellformedness of register transfers

Figure B.1: Proof Tree for the eb_cg_cg12_reg Lemma

Figure B.1 shows a simplified proof tree for the eb_cg_cg12_reg lemma derived using eb_cg_cg12
lemma above.

The first subgoal corresponds to the case when the register expression for which we want to compare the extracted behavior does not appear in \(cg\). In this case, its behavior is extracted only from \(cg_1\) and \(cg_2\), and the subgoal is discharged since equality of behaviors in this case is stated in the hypothesis.

The following excerpt from the PVSs’ interactive proof-checker shows the sequent of the second subgoal in Figure B.1. The first formula in the antecedent is the \(eb_{cg\_cg12}\) lemma. The next two formulas state useful properties about the \(match\) function. The fourth formula come from the hypothesis. Formula [4] in the consequent is proved by instanitating formula \([-1]\) with \(ri!1\) and (after simplification) with expression \(car(match(ri!1, cg!1))\)'regassign(ri!1).

Rule? (inst -1 "cg!1" "cg2!1" "match(ri!1, cg!1)")

Instantiating the top quantifier in -1 with the terms:
\(cg!1\), \(cg2!1\), \(match(ri!1, cg!1)\),

this simplifies to:
\(eb_{cg\_cg12\_reg.2}\):

\([-1]\) \((\forall (r: (registers(match(ri!1, cg!1))))):
\quad extracted\_behavior(reg(r), cg!1)) =
\quad extracted\_behavior(reg(r), cg2!1))
=>
\((\forall (e: (car(match(ri!1, cg!1))\{'exp\_set\})):
\quad extracted\_behavior(append(match(ri!1, cg!1), cg!1), e)) =
\quad extracted\_behavior(append(match(ri!1, cg!1), cg2!1), e))

\([-2]\) match(ri!1, append(cg!1, cg2!1)) = append(match(ri!1, cg!1), cg2!1)

\([-3]\) match(ri!1, append(cg!1, cg!1)) = append(match(ri!1, cg!1), cg1!1)

\([-4]\) \((\forall (r: (registers(cg!1)))):
\quad extracted\_behavior(reg(r), cg1!1)) =
\quad extracted\_behavior(reg(r), cg2!1))

\[null?(match(ri!1, append(cg!1, cg2!1)))\]
\[null?(match(ri!1, append(cg!1, cg!1)))\]
\[null?(match(ri!1, cg!1))\]
\[extracted\_behavior(append(match(ri!1, cg!1), cg1!1),
\quad car(match(ri!1, cg!1))\{'regassign(ri!1)\}) =
\quad extracted\_behavior(append(match(ri!1, cg!1), cg2!1),
\quad car(match(ri!1, cg!1))\{'regassign(ri!1)\})\]

B.2 Proof of the \(eb_{cg\_cg12}\) lemma

The proof for \(eb_{cg\_cg12}\) is more involved since it requires strong induction on the length of the control graph \(cg\), and simple induction on the expression trees. A simplified proof tree is showed in Figure B.2. The first proof command is the application of the strong induction mechanism to
the cg control graph:

(INDUCT "cg" :NAME wf_induction[list[wellformed_rt], list_adt[wellformed_rt].<<])

In the above, \texttt{wf\_induction} (built-in in PVS) axiomatizes the strong induction scheme for a generic type and a generic well-founded relation. We instantiate \texttt{wf\_induction} with the control graph type (which is a list of \texttt{wellformed\_rts}) and with the well-founded relation (pre-defined in a \texttt{list} theory of PVS). This generates three subgoals: the base case, the inductive step and a subgoal which requires to prove that << is well-founded. We will briefly explain how each subgoal was proved:

- **Subgoal 1.** The base case (cg is the null list) is automatically discharged using an aggregate command of PVS, \texttt{ASSERT}. \texttt{ASSERT} performs arithmetic and equality decision procedures, rewrting and other simplification techniques [7].

![Proof Tree for the eb_cg_cg12 Lemma](image-url)
Subgoal 2 is the inductive step. The following excerpt from the proof-checker environment shows a skolemized and simplified version of this subgoal.

\[-1\] FORALL (y: list[wellformed_rt]):
  y << x!1 IMPLIES
  cons?(y) IMPLIES
  (FORALL (cg1, cg2: c_graph):
    (FORALL (r: (registers(y))):
      extracted_behavior(reg(r), cg1) =
      extracted_behavior(reg(r), cg2))
    =>
    (FORALL (e: (car(y)'exp_set)):
      extracted_behavior(append(y, cg1), e) =
      extracted_behavior(append(y, cg2), e)))

\[-2\] cons?(x!1)

\[-3\] FORALL (r: (registers(x!1))):
  extracted_behavior(reg(r), cg1!1) =
  extracted_behavior(reg(r), cg2!1)

\[----\]

\[1\] FORALL (e: (car(x!1)'exp_set)):
  extracted_behavior(append(x!1, cg1!1), e) =
  extracted_behavior(append(x!1, cg2!1), e)

The first formula in the antecedent is the induction hypothesis. The next prover command automatically applies induction on operator trees: \texttt{INDUCT 'e'}. It generates two subgoals:

- **Subgoal 2.1** is the base case when the inductive variable is a register leaf. There are two sub-cases:
  
  * **Subgoal 2.1.1**, where the register expression does not appear as an output register in the control graph \texttt{cg}. This case is discharged using appropriate instantiations (subgoal 2.1.1.1), but it generates a type-checking condition (TCC)(subgoal 2.1.1.2): that the register used in instantiation is a register that appear in \texttt{cg}. The TCC is due to the “predicated” choose for \texttt{r} as belonging to \texttt{register(cg)}.
  
  * **Subgoal 2.1.2** considers the case when \texttt{r} appears as an output register in \texttt{cg}. Here we instantiate the strong induction hypothesis with the sub-list \texttt{match(r,cg)} (defined in Section 2.3.4; this generates an additional subgoal (2.1.2.2) which requires to prove that \texttt{match(r,cg)} is indeed in a sbterm relation with respect to \texttt{cg}. The other subgoal (2.1.2.1) is further instantiated with the \texttt{e1 = car(match(r,cg))'regassign(r)} expression. This instantiation also generates a TCC: that \texttt{e1} belongs to the set of expressions of the first register transfer in \texttt{match(r,cg)}. This TCC is discharged using the well-formedness property for register transfers.
- **Subgoal 2.2** is the inductive step when the expression is an operator tree. It is easily discharged, but two TCCs are generated; they require that both sources of the operator expression belong to the expression set of the first register transfer in \( cg \): 
  \[
  \text{car}(cg) \cdot \exp_set(source1(e)) \quad \text{and} \quad \text{car}(cg) \cdot \exp_set(source1(e)).
  \]
  The TCCs are discharged using the well-formedness property for register transfers.

- **Subgoal 3** refers to the relation \( \ll \) which has to be well-founded in order to allow application of strong induction. A built-in axiom for the well-founded relation on lists discharged this subgoal.

### B.3 Proof for lemma \( \text{eb}_c\text{g}_c\text{g}_\text{reg} \)

Following the same reasoning as for the \( \text{eb}_c\text{g}_c\text{g}_1\text{g}_2\text{reg} \) lemma, in order to prove \( \text{eb}_c\text{g}_c\text{g}_\text{reg} \) we first prove a sublemma which deals with the extracted behavior of the expressions from which the output registers read their values. We first proved a sublemma for the case when the \( cg \) control graph contains only one register transfer, then we extended the proved property to control graphs with arbitrary lengths using simple induction on the list length. This sub-lemma is:

```
\( \text{eb}_c\text{g}_\text{rt}: \ \text{LEMMA} \)
\[
\forall (cg1,cg2: \text{cons_graph}, rt: \text{wellformed_rt}) : \\
\forall (e1: (\text{car}(cg1) \cdot \exp_set), e2: (\text{car}(cg2) \cdot \exp_set)) : \\
\quad \text{extracted_behavior}(cg1,e1) = \text{extracted_behavior}(cg2,e2) \Rightarrow \\
\quad \text{extracted_behavior}(\text{append}(cg1, \text{cons}(rt, \text{null})), e1) = \\
\quad \text{extracted_behavior}(\text{append}(cg2, \text{cons}(rt, \text{null})), e2)
\]
```

To prove \( \text{eb}_c\text{g}_\text{rt} \) we used the same general approach as in the case of the \( \text{eb}_c\text{g}_c\text{g}_1\text{g}_2 \) lemma: apply strong induction on the length of the control graph, followed by simple induction on the expression trees. Here the proof was complicated by the fact that inductions are applied to four variables: \( cg1, cg2, e1 \) and \( e2 \). The application of induction schemes are illustrated in Figure B.3. The proofs for the generated subgoals are similar to those described for \( \text{eb}_c\text{g}_c\text{g}_1\text{g}_2 \)
forall\( (cg_1, cg_2: \text{cons\_graph}, \ rt: \text{wellformed\_rt}) : \)
forall\((e_1: (\text{car}(cg_1)\ \text{exp\_set}), \ e_2: (\text{car}(cg_2)\ \text{exp\_set})) : \)
extracted\_behavior\( (cg_1, e_1) \) = extracted\_behavior\( (cg_2, e_2) \) \( \Rightarrow \)
extracted\_behavior\( (\text{append}(cg_1, \ \text{cons}(rt, \text{null})), \ e_1) \) =
extracted\_behavior\( (\text{append}(cg_2, \ \text{cons}(rt, \text{null})), \ e_2) \)

- apply strong induction on the length of \( cg_1 \)

1 Base Case 2 Inductive Step

- apply strong induction on the length of \( cg_2 \)

2.1 Base Case 2.2 Inductive Step

- apply simple induction on \( e_1 \)

2.2.1 Base Case 2.2.2 Inductive Step

- apply simple induction on \( e_2 \)

Figure B.3: Proof Tree for the eb\_cg\_rt Lemma
Appendix C

Proofs for the Wellformedness and Behavior Preserving Correctness

C.1 Example for the OC Transformation

For each transformation we proved three lemmas which assert that: (1) the wellformedness property for the transformed register transfer is preserved,(2) the extracted behavior of the output registers of the affected register transfer is preserved, and (3) the resulting control graph is equivalent with the initial graph. We illustrate the proof strategy with an example for the OC transformation.

C.2 Proof for the Wellformedness Property

The following figure shows the sequent for the wellformed_oc lemma (see Section refocfor). The first formula in the antecedent is obtained by making the type of rt!1 (initial transfer) explicit. This is achieved by the TYPEPRED command in PVS. The {-2} formula expresses the precondition for OC.

```
Rule? (typepred "rt!1")
Adding type constraints for rt!1,
this simplifies to:
wellformed_oc :

{-1} wellformed_rt?(rt!1)
[-2] wellfound_link?(l!1, new_op!1, rt!1)
|--------
[1] wellformed_rt?(oc(rt!1, new_op!1, l!1))
```
After expanding all definitions, the property to be proved and the hypothesis properties become visible in the sequent as:

- the first two formulas in the antecedent correspond to the first and second wellformedness condition imposed on sets (See PVS Definition in Section 2.3.1).
- the third formula in antecedent correspond to the wellformedness condition imposed on the sources of the output registers (see PVS Definition in Section 2.3.1).
- the last two formulas in the antecedent and the first formula in the consequent result after expanding the wellfound_link? function. They are preconditions for well-formedness and read: ({-4}) 1 should exist as a link in the initial register transfer, ({-5}) the new operator an the copied operator should have the same functionality, and ({{1}}) the wellformedness condition is satisfied, or new_op!1 appears as an operator instance in rt!1.
- formula [2] in the consequent requires to prove that the wellformedness conditions also stand for the transformed register transfer (oc(rt!1, new_op!1, l!1)).

The sketch of the proof after splitting the consequent is suggerated in Figure C.1. To prove that the first wellformedness condition for expression sets stands, we apply induction on variable e and perform case analysis after expanding the recursive definition of new_exp. The induction hypothesis asserts that if the sources of the expression e belong to the exp_set field of the transformed register transfer, then e also belongs to this field. The second and third subgoal are discharged using appropriate instantiations.
C.3 Proof for the Correctness Property

To prove correctness of the OC transformation we first proved that each expression in the exp.set field of the affected register transfer has the same extracted behavior as that expression in the initial transfer from which it was generated (through the new.exp function:

\[
\text{new.exp\_behavior: LEMMA} \\
\forall (rt:\text{wellformed\_rt}, \text{new\_op}:\text{operator}, \text{l}:\text{op\_link}, e:(rt'\text{exp\_set})) : \\
\text{wellfound\_link?}(\text{l},\text{new\_op},rt) \Rightarrow \\
\text{extracted\_behavior}(e) = \text{extracted\_behavior(new\_exp(e,\text{new\_op},l))}
\]

The new.exp\_behavior lemma was proved by induction on the expression variable e. By applying induction on operator trees we obtain the following sequent which states that: if the sources of an operator expression e generated by the new.exp function have the same extracted behavior as their correspondant expressions in the initial transfer, then e also has the same extracted behavior as that expression from which it was generated. This subgoal is discharged by expanding the definition of new.op in consequent and performing case analysis.

\[\begin{align*}
[-1] \quad & \text{extracted\_behavior(op2\_var!1)} = \\
& \quad \quad \text{extracted\_behavior(new\_exp(op2\_var!1, new\_op!1, 1!1))} \\
[-2] \quad & \text{extracted\_behavior(op3\_var!1)} = \\
& \quad \quad \text{extracted\_behavior(new\_exp(op3\_var!1, new\_op!1, 1!1))} \\
[-3] \quad & \text{rt!1'\text{exp\_set}(op(op1\_var!1, op2\_var!1, op3\_var!1))} \\
[-4] \quad & \text{wellfound\_link?}(1!1, new\_op!1, rt!1) \\
\end{align*}\]

\[\begin{align*}
[1] \quad & \text{extracted\_behavior(op(op1\_var!1, op2\_var!1, op3\_var!1)) =} \\
& \quad \quad \text{extracted\_behavior(new\_exp(op(op1\_var!1, op2\_var!1, op3\_var!1), new\_op!1, 1!1))}
\]
C.3.1 Proof for the Equivalence of Control Graphs Before and After the Transformation - The Correctness Theorem

The following sequent shows the correctness goal after performing: (1*) simplification procedures, (2*) rewriting a lemma which transforms the replacement of a register transfer in a control graph with a sequence of append operations (see Section 4.2 for the original formulation of the Correctness Theorem), and (3*) rewriting a lemma which transforms a control graph \( \text{cg}!1 \) in a concatenation (append) of sublists.

\[
\text{[{-1}]} \begin{array}{l}
\text{wellfound_link?}(l!1, \text{new_op}!1, \text{nth}(\text{cg}!1, n!1)) \\
\end{array}
\begin{array}{l}
\text{[1]}\quad \text{extracted_behavior}(r!1, \text{append} (\text{start_cg}(\text{cg}!1, n!1),) \\
\text{\hspace{0.5in}} \begin{array}{c}
\text{cons}(\text{nth}(\text{cg}!1, n!1), \text{rest_cg}(\text{cg}!1, n!1))) \\ 
\text{hspace*}{0.5in} \{\textbf{(3*)} \}
\end{array}
\end{array}
\begin{array}{l}
\text{\hspace{0.5in} =} \\
\text{extracted_behavior}(r!1, \\
\text{\hspace{1in} append (start_cg}(\text{cg}!1, n!1),) \\
\text{\hspace{1.5in} append (cons (oc(nth}(\text{cg}!1, n!1), \text{new_op}!1, l!1), \text{null}),} \\
\text{\hspace{1.5in} rest_cg}(\text{cg}!1, n!1))) \\
\text{\hspace{1.5in} hspace*}{1in} \{\textbf{(2*)} \})
\end{array}
\]

The \( \text{start_cg}(\text{cg}!1, n!1) \) function defines the list of transfers performed after control step \( n!1 \), and \( \text{rest_cg}(\text{cg}!1,n!1) \) - the sublist of transfers performed before \( n!1 \). We analyze two cases which generate the following subgoals:

- **Subgoal 1** corresponds to the case when the register \( r!1 \) does not appear as an output an input in the \( \text{start_cg}(\text{cg}!1, n!1) \) list, i.e., it is not an element of \( \text{registers}(\text{start}(\text{cg}!1,n!1)) \). In this case, the behavior of \( r!1 \) is extracted only from:

\[
\text{append} (\text{cons (oc(nth}(\text{cg}!1, n!1), \text{new_op}!1, l!1), \text{null}), \text{rest_cg}(\text{cg}!1, n!1))
\]

This subgoal is discharged by appropriately instantiating in the \( \text{eb_cg\_cg\_reg} \) lemma as follows:

- substitute \( \text{cg1} \) with \( \text{append} (\text{cons (nth}(\text{cg}!1, n!1)), \text{null}) \).
- substitute \( \text{cg2} \) with \( \text{append} (\text{cons (oc(nth}(\text{cg}!1, n!1), \text{new_op}!1, l!1), \text{null})) \).
- substitute \( \text{cg} \) with \( \text{rest_cg}(\text{cg}!1, n!1) \).
- substitute \( \text{r} \) with \( r!1 \).

- **Subgoal 2** corresponds to the case when \( r!1 \) appears in \( \text{start_cg}(\text{cg}!1,n!1) \). Then, the \( \text{eb_cg\_cg\_cg\_reg} \) lemma is used. The instantiations are performed very similar to the previous case.
It was necessary to consider two separate cases because the register variables used in defining \( eb_{cg}CGCG_REG \) are of the \( (registers(start_{cg}(CG!1, N!1))) \) subtype. This generates a TCC obligation for the second subgoal which cannot be discharged when \( R!1 \) does not appear in \( start_{cg}(CG!1, N!1) \).

### C.4 Proof for Correctness for the RIS Transformation

The wellformedness preserving property was proved similarly for all previous transformations. In proving the Correctness Theorem for OIS, RTS, RTM, RTD and RTC we used approaches similar to that described here for OC. The application of RIS differs from the other transformations: it is applied locally to that register transfer where the substituted register appear as an input, and than it searches for the last transfer were this register was written and applies a transformation at that transfer. Hence, it was not possible to reuse the same strategy in proving correctness for RIS.

RIS checks for two cases: (1) when the substituted register, \( S \), is a primary input, and (2) when \( S \) is written at another control step \( m \), with \( n < m \) (see also Section 4.8). When \( S \) is a primary input then the behavior preserving correctness can be easily proved by simply expanding the definitions in the sequent. If \( S \) is not a primary input then RIS performs a replacement at that register transfer where \( S \) was last written. This second case required proof by strong induction on the length of the control graph. We first proved the particular case when \( S \) is substituted in the first register transfer (case (a) in Figure C.2). The following lemma refering to the particular case when \( n = 0 \), states that forall: register transfers \( cg \), register expressions \( subst \) and register expressions \( wth \), all output registers \( r \) except \( wth \) have the same extracted behavior in the initial and in the transformed control graph, provided a precondition is satisfied. This precondition is weaker then in the case of an arbitrary \( n \).

```ocaml
RIS_preserve_behavior_r01: LEMMA
forall(cg: cons_graph, subst:(inputs(car(cg))), wth:(reg?),
        r:(output_registers(cg))) :
        cons?(match(subst,cdr(cg))) =>
        (let n1 : below[length(cg)] = match_pos(cdr(cg),subst) in
         (((forall (n:below[n1]) : not inputs(nth(cg,n))(wth)
          AND
          forall (n:below[n1]) : not outregs(nth(cdr(cg),n))(wth))
          AND
          r /= wth) =>
          (extracted_behavior(reg(r),cg) = extracted_behavior(reg(r),RIS(cg,0,subst,wth)))))
```
Two additional lemmas prepare the application of a strong induction scheme for the case when RIS is applied at an arbitrary control step \( n \).

The \texttt{RIS} \_n lemma below defines the application of RIS at an arbitrary control step \( n \) as a concatenation of two lists, one of which being the particular case \( n = 0 \).

The \texttt{ris\_precondition} \_n lemma asserts that: if the RIS precondition (see Section 4.8) stands for a pair of substituted-substituting registers in the control graph \( cg \) at position \( n \), then it also stands for the same registers in the control graph \( \text{match}(r, \text{cdr}(cg)) \) (see Figure C.2) at position \( n - \text{match\_pos}(\text{cdr}(cg), r) \)). The \texttt{match\_pos} function defines for a register \( r \) the position of the register transfer where \( r \) first appears as an output (\( r \) was last written).
RIS\_n: lemma
\[
\forall (cg: \text{cons\_graph}, n: \text{below}[\text{length}(cg)], \text{subst}:(\text{inputs}(\text{nth}(cg,n))), wth:(\text{reg}\?)) : \\
\text{RIS}(cg,n,\text{subst},wth) = \text{append}(\text{start\_cg}(cg,n),\text{RIS}(\text{cons}(\text{nth}(cg,n),\text{rest\_cg}(cg,n)),0,\text{subst},wth))
\]

ris\_precondition\_n: LEMMA
\[
\forall (cg: \text{cons\_graph}, n: \text{below}[\text{length}(cg)], \text{subst}:(\text{inputs}(\text{nth}(cg,n))), wth:(\text{reg}\?), r:(\text{reg}\?)) : \\
\left( n > 0 \land \text{cons?}(\text{match}(r,\text{start\_cg}(\text{cdr}(cg),n-1))) \Rightarrow \\
\text{ris\_precondition}(cg,n,\text{subst},wth) \Rightarrow \\
\text{ris\_precondition}(\text{match}(r,\text{cdr}(cg)),n-\text{match\_pos}(\text{cdr}(cg),r),\text{subst},wth) \right)
\]

The correctness proof for RIS follows the same strategy used in proving the \text{eb\_cg\_cg12} lemma. The strong induction applied on the length of \(cg\) generates a trivial base case and an inductive step. For the inductive step we check if the inductive register variable appears in \text{start\_cg}(\text{cdr}(cg),n) as an output register; this generates two subgoals:

- **Subgoal 1.** If the inductive register variable \(r\) does not appear in \text{start\_cg}(\text{cdr}(cg),n) as an output register (\text{null?}(\text{match}(r,\text{start\_cg}(\text{cdr}(cg),n)))) then its behavior is extracted from: \(\text{RIS}(\text{cons}(\text{nth}(cg,n),\text{rest\_cg}(cg,n)),0,\text{subst},wth)\). This case was dealt with by the \text{RIS\_preserve\_behavior\_r01} lemma.

- **Subgoal 2.** If the inductive variable \(r\) appears in \text{start\_cg}(\text{cdr}(cg),n) (predicate \text{cons?}(\text{match}(r,\text{start\_cg}(\text{cdr}(cg),n))) evaluates to \text{true}) then we instantiate the inductive hypothesis with that sublist whose first register transfer contains \(r\) as an output register: \text{match}(r,\text{cdr}(cg)). As illustrated in Figure C.2, this sublist is in a subterm relation with \(cg\), i.e., its length is smaller.

Note that the substituting register \(wth\) (\(W\) in Figure C.2) is not alive at control step \(n\), as enforced by the \text{precondition\_ris} predicate. As a result, if the inductive variable \(r\) equals \(wth\) and it appears as an input in \text{start\_cg}(cg,n) (Figure C.2.b), then \(r\) \(wth\) is last written after control step \(n\). This corresponds to **Subgoal 2**, so the \text{RIS\_preserve\_behavior\_r01} lemma (which excluded \(wth\)) is not used.
C.5 Proof for Correctness for the CO Transformation

Because of the global precondition predicate, the proof for the behavior preserving correctness of CO requires induction on both expression trees and control graph list. Like in the case of RIS, we employ strong induction for traversing the control graph, and simple induction for the expression trees. We first prove a lemma stating that each expression of the first register transfer have the same extracted behavior in the initial and in the transformed design.

\[\text{clean_outputs1_sr_exp_PB : LEMMA}\]
\[\forall (cg: \text{cons}\_\text{graph}, n: \text{below}[\text{length}(cg)], rset: \text{set}[(\text{reg?})]) :\]
\[\forall (e: (\text{car}(cg)\text{'exp_set})) :\]
\[\text{precondition}_\text{clean}(cg, n, rset) \Rightarrow\]
\[\text{extracted}\_\text{behavior}(cg, e) = \text{extracted}\_\text{behavior}(\text{clean}\_\text{outputs1}\_\text{sr}(cg, n, rset), e)\]

The proof strategy is shown in Figure C.3. We briefly explain the proof tree generated and how each subgoal was discharged. The first command in the proof environment invokes strong induction on the control graph list \(cg\). Three subgoal are generated:

1. **Subgoal 1** corresponds to the base case \(cg\) is the null list) and is automatically discharged by an ASSERT built-in strategy.

2. **Subgoal 2** corresponds to the inductive step. At this stage of the proof we apply a further induction scheme, on the expression trees. This generates two subgoals:

   - **Subgoal 2.1** corresponds to the base case when expression \(e\) is a register. We analyze two cases:

     - **Subgoal 2.1.1** corresponds to the case when \(e\) does not appear as an output register in the first \(n-1\) register transfers \((\text{null?}(\text{match}(e, \text{start}\_cg(cg, n)))\)). Two more subgoals deal with the cases when:

       - **Subgoal 2.1.1.1** the register expression \(e\) belongs to \(rset\), that is, it is one of the registers to be removed. The hypothesis \(\text{null?}(\text{match}(e, \text{start}\_cg(cg, n)))\) and the precondition for CO, \(\text{precondition}\_\text{clean}(cg, n, rset)\) derive a contradiction. The precondition states that if a register from \(rset\) appear as an input before the \(n\)'th register transfer, then it should also appear as an output before the \(n\)'th control step. This contradiction discharges this subgoal.

       - **Subgoal 2.1.1.2** the register expression \(e\) does not belong to \(rset\). A proof
1. Base Case: \( \text{cg} = \text{null} \)
2. Inductive Step
   - apply induction on \( \text{e} \)
3. "\( << \)" is a well-founded relation
   - use a built-in axiom

2.1. Base Case: \( \text{e} \) is leaf (register)
   - apply induction on \( \text{cg} \)
2.2. Inductive Step: \( \text{e} \) is node (operator)
   - use the strong induction hypothesis
     - instantiate antecedent with \( \text{car} \)
     - use the well-formedness condition of transfers

2.1.1. If \( \text{rset}(\text{e}) = \text{true} \)
   - (derive contradiction from the CO precondition)

2.1.2. If \( \text{rset}(\text{e}) = \text{false} \)
   - use \( \text{eb_null_match} \)
     & \( \text{eb_rt_cg} \)

2.1.2.1. Use a collection of lemmas
2.1.2.2 (TCC) (instantiation satisfies the well-founded relation \( << \))
2.2.1 (TCC) (source1(\( \text{e} \)) and source2(\( \text{e} \)) should belong to \( \text{car}(\text{cg})'\text{exp_set} \))
2.2.2 (TCC) use the well-formedness condition of transfers

Figure C.3: Proof Tree for the Correctness of CO

for this subgoal was derived by appropriately instantiating the following two sublemmas:

\[
\text{clean_outputs1_sr_exp_PB : LEMMA}
\forall (\text{cg} : \text{cons_graph}, n : \text{below}[\text{length(\text{cg})}], \text{rset} : \text{set}[(\text{reg}?)]) :
\forall (\text{e} : \text{car}(\text{cg})'\text{exp_set}) : 
\text{precondition_clean(\text{cg}, n, \text{rset})} \Rightarrow
\text{extracted_behavior(\text{cg}, \text{e})}
= \text{extracted_behavior(\text{clean_outputs1_sr(cg,n,rset)}, \text{e})}
\]

\[
\text{apply strong induction on \text{cg}}
\]

1. Base Case: \( \text{cg} = \text{null} \)
2. Inductive Step
   - apply induction on \( \text{e} \)
3. "\( << \)" is a well-founded relation
   - use a built-in axiom

2.1. Base Case: \( \text{e} \) is leaf (register)
   - apply induction on \( \text{cg} \)
2.2. Inductive Step: \( \text{e} \) is node (operator)
   - use the strong induction hypothesis
     - instantiate antecedent with \( \text{car}(\text{cg})'\text{exp} \)
     - use the well-formedness condition of transfers

2.1.1. If \( \text{rset}(\text{e}) = \text{true} \)
   - (derive contradiction from the CO precondition)

2.1.2. If \( \text{rset}(\text{e}) = \text{false} \)
   - use \( \text{eb_null_match} \)
     & \( \text{eb_rt_cg} \)

2.1.2.1. Use a collection of lemmas
2.1.2.2 (TCC) (instantiation satisfies the well-founded relation \( << \))
2.2.1 (TCC) (source1(\( \text{e} \)) and source2(\( \text{e} \)) should belong to \( \text{car}(\text{cg})'\text{exp_set} \))
2.2.2 (TCC) use the well-formedness condition of transfers

\[
\text{eb_null_match : LEMMA}
\forall (\text{cg1}, \text{cg2} : \text{c_graph}, \text{r} : (\text{reg}?)) :
\text{null?}(\text{match}(\text{r}, \text{cg1})) \Rightarrow
\text{extracted_behavior}(\text{reg}(\text{r}), \text{append}(\text{cg1}, \text{cg2})) = \text{extracted_behavior}(\text{reg}(\text{r}), \text{cg2})
\]

\[
\text{eb_rt_cg : LEMMA}
\forall (\text{rt1} : \text{wellformed_rt}, \text{rt2} : \text{wellformed_rt}, \text{cg} : \text{c_graph}) :
\forall (\text{e1} : (\text{rt1}'\text{exp_set}), \text{e2} : (\text{rt2}'\text{exp_set})) :
\text{extracted_behavior}(\text{cons}(\text{rt1}, \text{null}), \text{e1}) =
\text{extracted_behavior}(\text{cons}(\text{rt2}, \text{null}), \text{e2}) \Rightarrow
\text{extracted_behavior}(\text{cons}(\text{rt1}, \text{cg}), \text{e1}) =
\text{extracted_behavior}(\text{cons}(\text{rt2}, \text{cg}), \text{e2})
\]

The first sublemma states that the extracted behavior of a register \( \text{r} \) in the concatenation of two control graphs \( (\text{append}(\text{cg1}, \text{cg2})) \) is the same as the ex-
tracted behavior of \( r \) in one of the two control graphs if \( r \) does not appear as an output in the other. The second lemma states that if two expressions have the same extracted behavior in two different register transfers, then they have the same extracted behavior in the control graphs resulted by appending these transfers to the same control graph.

- **Subgoal 2.1.2** when the register expression \( e \) appears as an output in at least one of the first \( n-1 \) register transfers. To discharge this subgoal we instantiate the strong induction hypothesis with the sub-list that starts at the first register transfer where \( e \) appears as an output: \( \text{match}(e, cg) \). The instantiation generates one type-checking condition, so the proof tree splits in two more subgoals:

  * **Subgoal 2.1.2.1** is the base subgoal. A further instantiation with \( \text{car}(\text{match}(e, cg)) \cdot \text{regassign} \) discharges the subsequent, but generates well-formedness type-checking conditions. These are further proved by expanding well-formedness definitions.

  * **Subgoal 2.1.2.2** corresponds to a type-checking condition requiring that the control graph used for instantiating the inductive step satisfies the well-founded relation. The rewriting of a pre-proved lemma discharges this subgoal.

- **Subgoal 2.2** corresponds to the inductive step for expression trees. It was proved by expanding the definition of \texttt{extracted.\texttt{behavior}} in the consequent and applying extensionality. Two type-checking conditions require that the sources of the operator expression we induct on belong to \( \text{car}(cg) \cdot \text{exp.set} \). They were discharged by expanding well-formedness definition and using a pre-proved lemma.

3. **Subgoal 3** is generated as a type-checking condition and requires that the \( \ll \) relation used for strong induction is well-founded. A built-in axiom generated for the \texttt{list} abstract datatype asserts the well-foundedness of this relation.
Appendix D

Proof Script for the Completeness Theorem

A proof for the Completeness Theorem was derived from LEMMA.1, LEMMA.2 and LEMMA.3 using appropriate instantiations. Here is the sequence of PVS Prover commands form proving the completeness theorem. The apply_sequence_append lemma states properties about appending lists of generic types.

(COMPLETNESS "" (SKOSIMP*)
  ("" (LEMMA "LEMMA.1")
  ("" (LEMMA "LEMMA.2")
  ("" (LEMMA "LEMMA.3")
   ("" (INST -3 "cg!1") ("" (SKOSIMP*) ("" (INST -1 "cg2!1") ("" (SKOSIMP*)
     ("" (INST -3 "cg_behavior(cg!1)"
        "output_registers(cg!1)" "apply_sequence(s!1)(cg!1)" "cgs!1")
     ("" (SKOSIMP*)
       ("" (INST 1 "append(s!1,append(s!3,s!2))")
       ("" (LEMMA "apply_sequence_append")
         ("" (INST -1 "s!1" "append(s!3,s!2)" "cg1!1") ("" (REPLACE -1 1)
           ("" (LEMMA "apply_sequence_append")
             ("" (INST -1 "s!3" "s!2" "apply_sequence(s!1)(cg!1)")
               ("" (ASSERT NIL NIL)NIL)NIL)NIL)NIL)NIL)NIL)NIL)NIL)NIL)NIL)NIL)
   ("" (ASSERT) ("" (HIDE 2) ("" (EXPAND "single_rt_cgs")
     ("" (ASSERT) ("" (GROUND) ("" (EXPAND "equiv_cg?")
       ("" (PROPAX NIL NIL) NIL)
     ("" (SKOSIMP*) ("" (INST -4 "r!1") ("" (INST -9 "r!1") ("" (ASSERT)
       ("" (REPLACE -4 1) ("" (EXPAND "equiv_cg")
         ("" (PROPAX NIL NIL)NIL)NIL)NIL)NIL)
     ("" (ASSERT) ("" (TYPEPRED "r!1")
       ("" (EXPAND "equiv_cg")
         ("" (PROPAX NIL NIL)NIL)NIL)NIL)NIL)NIL)) ....

154
D.1 Preconditioning for RTD - Proof Strategy

We first proved that the `break_cycles` function applied to a control graph `cg` has the same effect as the application of a sequence of operational transformations:

```
break_cycles_sequence: LEMMA
forall(cg: c_graph, n: below[|length(cg)|]) :
    exists (s: oper_tr_sequence) : break_cycles(cg, n) = apply_sequence(s)(cg)
```

The `break_cycles` function applied to the `n`th register transfer of a control graph `cg (nth(cg, n))` is defined recursively for the elements of the set of register expressions that appear as both inputs and outputs in that register transfer (`loop_set(nth(cg, n))`). Hence, in order to prove any property about `break_cycles` we need to induct on the cardinality of this set. As mentioned in Section 2.3.1, the PVS underlying logic does not allow partial function definitions, so all recursive definitions will terminate. Consequently, the sets on whose elements the induction is performed need to be finite. A collection of inductive schemes applicable on finite sets is available in a distributed PVS library. In proving the `break_cycles_sequence` lemma we used a pre-defined induction scheme for finite sets, namely the `finite_sets_induction_rest` lemma which states that for all finite sets `SS` and any predicate `P` on finite sets, if `P` stands for the empty set and for the `rest(SS)` set, then it also stands for the set `SS`. The `rest(SS)` set is the set resulted by removing an arbitrary element from `SS`.

```
finite_set_induction_rest: THEOREM P(emptyset[T])
    AND (FORALL SS : P(rest(SS)) IMPLIES P(SS)) IMPLIES (FORALL S : P(S))
```

We instantiated this induction scheme with the following predicate:

```
forall(cg: c_graph, n: below[|length(cg)|], ss: finite_set[(reg?)]) :
    ss = loop_set(nth(cg, n)) => EXISTS (s: oper_tr_sequence):
        break_cycles(cg, n) = apply_sequence(s)(cg)
```

Note that the direct application of induction on the `loop_set` is not possible since this set does not universally quantify the formula on which induction is applied. The following excerpt from the PVS proof checker presents the sequents corresponding to the two subgoals resulted after instantiating...
the *finite_sets_induction_rest* hypothesis with the predicate above, and after simplifications:

break_cycles_sequence.1 :

[-1] \text{FORALL} (\text{cg: c_graph}, n: \text{below[length(cg)]}, ss: \text{finite_set[(reg?)]}):
    \[
    ss = \text{loop_set(nth(cg, n))} \Rightarrow
    (\text{EXISTS (s: oper_tr_sequence)}):
    \text{break_cycles(cg, n)} = \text{apply_sequence}(s)(cg)
    \]

[1] \text{EXISTS (s: oper_tr_sequence)}:
    \text{break_cycles(cg!1, n!1)} = \text{apply_sequence}(s)(cg!1)

break_cycles_sequence.2 :

[\text{-------}]

\{1\} \text{FORALL (cg: c_graph}, n: \text{below[length(cg)]}, ss: \text{finite_set[(reg?)]}):
    \[
    ss = \text{loop_set(nth(cg, n))} \Rightarrow
    (\text{EXISTS (s: oper_tr_sequence)}):
    \text{break_cycles(cg, n)} = \text{apply_sequence}(s)(cg)
    \]

[2] \text{EXISTS (s: oper_tr_sequence)}:
    \text{break_cycles(cg!1, n!1)} = \text{apply_sequence}(s)(cg!1)

The first subgoal is discharged by instantiating the antecedent formula with: \text{cg!1}, \text{n!1} and respectively \text{loop_set(nth(cg!1, n!1))}. We prove the second subgoal by induction on \text{ss} (now appearing as a bound variable in the universally quantified formula). The automatic application of the *finite_set_induction_rest* induction scheme yields a base case - where \text{ss} is the empty set, and an inductive step. In the base case, \text{loop_set} is empty, so the *break_cycles* function leaves the control graph unchanged. The consequent is instantiated with the transformations sequence consisting of only one transformation : the identity transformation (\text{idnt_tr}).

The sequent corresponding to the inductive step, after expanding the definition for *break_cycles* in formula \{1\} and hiding formula [2], and after propositional simplifications, is:

[-1] \text{FORALL (cg: c_graph, n: below[length(cg)]):}
    \[
    \text{rest}(SS!1) = \text{loop_set(nth(cg, n))} \Rightarrow
    (\text{EXISTS (s: oper_tr_sequence)}):
    \text{break_cycles}(cg, n) = \text{apply_sequence}(s)(cg)
    \]

[-2] SS!1 = \text{loop_set(nth(cg!2, n!2))}

[\text{-------}]

\{1\} \text{EXISTS (s: oper_tr_sequence)}:
    \text{break_cycle(RISPLUS(cg!2, n!2, choose(loop_set(nth(cg!2, n!2))),
    \text{new_reg(registers(cg!2))},
    n!2)}
    = \text{apply_sequence}(s)(cg!2)

We instantiate the first formula in the antecedent with the following control graph:
RISPLUS(cg!2, n!2, choose(loop_set(nth(cg!2, n!2))), new_reg(registers(cg!2))).

After simplifications, this results in the following two subgoals:

break_cycles_sequence.2.2.1.1 :

{-1} EXISTS (s: oper_tr_sequence):
    break_cycles(RISPLUS(cg!2, n!2, choose(loop_set(nth(cg!2, n!2))),
    new_reg(registers(cg!2))),
    n!2)
    =
    apply_sequence(s)
    (RISPLUS(cg!2, n!2,
    choose(loop_set(nth(cg!2, n!2))),
    new_reg(registers(cg!2))))

 diverse

SS!1 = loop_set(nth(cg!2, n!2))

|-------

[1] EXISTS (s: oper_tr_sequence):
    break_cycles(RISPLUS(cg!2, n!2, choose(loop_set(nth(cg!2, n!2))),
    new_reg(registers(cg!2))),
    n!2)
    = apply_sequence(s)(cg!2)

break_cycles_sequence.2.2.1.2 :

[-1] SS!1 = loop_set(nth(cg!2, n!2))

|-------

[1] rest(SS!1) =
    loop_set(nth(RISPLUS(cg!2, n!2, choose(loop_set(nth(cg!2, n!2))),
    new_reg(registers(cg!2)))))

The first subgoal is discharged using the following rules:

- Skolemize formula {-1}. This operation yields a witness sequent, s!1.
- Instantiate formula [1] with the sequence obtained by appending the RISPLUS transformation to the sequence s!1.

The second subgoal reads: the loop_set of the control graph resulted after the RISPLUS transformation contains one register less. This is true since RISPLUS always substitutes an existing register with a new one.

The proof for the preconditioned_sequence lemma consists of instantiating the consequent with a sequence of transformations obtained by appending the clean_outputs function to the witness sequence supplied by the break_cycles_sequence.
Note that an important type checking condition will be generated for each subgoal, which requires to prove that RISPLUS is a sequence of (correct) operational transformations.

D.2 Proof Strategy for Discharging Type-Checking Conditions (TCCs)

Each TCC generated upon type-checking for an operational transformation requires proving three subgoals:

- The well-formedness of the generated control graph.
- The behavior preserving correctness of the transformation.
- The ability to express the transformation as one of the seven transformations presented in Chapter 4.

The well-formedness and correctness subgoals for the \( \text{rdtf} \) function in Section 5.2.2 after skolemization are:

\[
\text{rdtf}_{\text{TCC3.1}} :
\]

\[
\{1\} \quad \text{IF } \text{cg}!2 = \text{preconditioned(cg}!1) \\
\quad \text{THEN IF empty?(nth[wellformed}\_\text{rt}\{\text{preconditioned(cg}!1), 1\}) \\
\quad \quad \text{THEN wellformed}\_\text{cg?}(\text{preconditioned(cg}!1)) \\
\quad \quad \text{ELSE wellformed}\_\text{cg?}(\text{RTD}(\text{preconditioned(cg}!1)), \\
\quad \quad \quad 1, \\
\quad \quad \quad \text{choose}[[\text{(reg?)}}] \\
\quad \quad \quad \text{nth[wellformed}\_\text{rt}\{\text{preconditioned(cg}!1), 1\}^\text{outregs})]) \\
\quad \quad \qquad \text{ENDIF} \\
\quad \text{ELSE wellformed}\_\text{cg?}(\text{idem}\_\text{tr(cg}!2)) \\
\quad \text{ENDIF}
\]

\[
\text{rdtf}_{\text{TCC3.2}} :
\]

\[
\{1\} \quad \text{IF } \text{cg}!2 = \text{preconditioned(cg}!1) \\
\quad \text{THEN IF empty?(nth[wellformed}\_\text{rt}\{\text{preconditioned(cg}!1), 1\}) \\
\quad \quad \text{THEN extracted}\_\text{behavior}(\text{reg[register, operator]}(r!1), \text{cg}!2) \\
\quad \quad \quad = \\
\quad \quad \quad \text{extracted}\_\text{behavior}(\text{reg[register, operator]}(r!1), \text{preconditioned(cg}!1)) \\
\quad \quad \qquad \text{ELSE extracted}\_\text{behavior}(\text{reg[register, operator]}(r!1), \text{cg}!2) =
\]

158
For each transformation, the well-formedness and the correctness properties are subject to a set of preconditions, usually weaker for well-formedness. Actually, the RTD and the RISPLUS transformations return always well-formed control graphs. For the example of the RTD transformation presented here, the well-formedness subgoal was discharged by simply expanding the definitions for iden_tr and wellformed_cg. To preserve the behavior, the preconditions are created by applying the preconditioned function, as defined in Section 5.2.1. We proved by induction on the cardinality of the loop set corresponding to the second register transfer that the break_cycles function creates the preconditions for decomposition. This is stated by the following lemma:

break_cycles_to_rtd_precond : LEMMA
forall(cg: c_graph, n: below[length(cg)]):
  rtd_precondition(nth(break_cycles(cg, n), n))

The proof for break_cycles_to_rtd_precond is very similar to the one described in Appendix D.1. The correctness subgoal is proved by appropriately instantiating the above lemma and the correctness theorem for RTD as defined in Section 4.6(first formula in the antecedent in the sequent bellow):

{-1}  FORALL (cg: cons_graph, n: below[length(cg)],
          r: (nth(cg, n)`outregs), ri: register):
         rtd_precondition(nth(cg, n)) =>
             extracted_behavior(ri, cg) = extracted_behavior(ri, RTD(cg, n, r))
[-2]  FORALL (cg: c_graph, n: below[length(cg)]):
         rtd_precondition(nth(break_cycles(cg, n), n))
[-3]  cg!2 = preconditioned(cg!1)
|-------
[1]   empty?(nth[wellformed_rt](preconditioned(cg!1), 1))
[2]   extracted_behavior(reg[register, operator](r!1), cg!2) =
         extracted_behavior(reg[register, operator](r!1),
                          RTD(preconditioned(cg!1), 1),
choose(((reg?)))

(thn[wellformed rt]

(preconditioned(cg!1), 1)'outregs)))

The first formula is instantiated with:\("\text{preconditioned}(c\!\!g!1)" \"1\" \text{choose}(\text{nth}(\text{preconditioned}(c\!\!g!1), 1)'\text{outregs})\"), and the second formula with ("\text{clean\_outputs}(c\!\!g!1, 1)" \"1\")

After skolemization, simplification and hiding the irrelevant consequent formulas, the subsequent for the third subgoal becomes:

\text{rdt\_TCC3.3} :

[-1] \text{cons?}[\text{wellformed rt}](c\!\!g!2)

---------

[1] IF c\!\!g!2 = \text{preconditioned}(c\!\!g!1)

THEN IF empty?(\text{nth}[\text{wellformed rt}](\text{preconditioned}(c\!\!g!1), 1))

THEN \text{preconditioned}(c\!\!g!1)

ELSE \text{RTD}(\text{preconditioned}(c\!\!g!1), 1,

\text{choose}(((\text{reg})))

(\text{nth}[\text{wellformed rt}]

(\text{preconditioned}(c\!\!g!1), 1)'\text{outregs}))

ENDIF

ELSE iden\_tr(c\!\!g!2)

ENDIF

= iden\_tr(c\!\!g!2)

[2] \text{EXISTS} (n: \text{below}[\text{length}(c\!\!g!2)],

r: (\text{nth}[\text{wellformed rt}](c\!\!g!2, n)'\text{outregs}):

IF c\!\!g!2 = \text{preconditioned}(c\!\!g!1)

THEN IF empty?(\text{nth}[\text{wellformed rt}](\text{preconditioned}(c\!\!g!1), 1))

THEN \text{preconditioned}(c\!\!g!1)

ELSE \text{RTD}(\text{preconditioned}(c\!\!g!1), 1,

\text{choose}(((\text{reg})))

(\text{nth}[\text{wellformed rt}]

(\text{preconditioned}(c\!\!g!1), 1)'\text{outregs}))

ENDIF

ELSE iden\_tr(c\!\!g!2)

ENDIF

= \text{RTD}(c\!\!g!2, n, r)

This subgoal is proved by instantiating the second formula in the consequent with 1 and respectively \text{choose}(\text{nth}(\text{preconditioned}(c\!\!g!1), 1)'\text{outregs}), followed by simplifications and lemmas about the lengths of the transformed graphs.
Appendix E

Proof Strategy for the to_minform_sequence Lemma

In the following PVS specifications, to_minform lemma states that the result of applying function to_minform to a control graph of length one is in the “canonical” minform?. to_minform_sequence asserts that there exists a sequence of operational transformations having the same effect on a control graph as to_minform.

```plaintext
to_minform(cg:(simple?)) : RECURSIVE c_graph =
  if null?(cg) then cg
  else if forall (e:(filter_op(car(cg)`exp_set))) : minform_exp?(car(cg),e)
    then cg
    else let e: (op?) = choose({o:(filter_op(car(cg)`exp_set)) |
      not minform_exp?(car(cg),o)}),
      s: (op?) = choose(same_sources(car(cg),e)) in
      to_minform(OIS(cg,0,e,op(s)))
    endif
  endif
MEASURE card(operators(cg))

to_minform_lemma: LEMMA forall(cg:(simple?)) : minform?(to_minform(cg))

to_minform_sequence: LEMMA
  forall(cg:(simple?)) :
    exists(s:oper_tr_sequence) : to_minform(cg) = apply_sequence(s)(cg)
```

Both lemmas were proved by simple induction on natural numbers. The following excerpt from the
proof-checker shows the sequent for the `to_minform_lemma` after instantiating the induction axiom with the following predicate:

\[
\text{forall}(cg:(\text{simple?}), n: \text{nat}) : n = \text{card}(\text{operators}(cg)) \Rightarrow \\
\text{EXISTS} (s: \text{oper_tr_sequence}) : \text{to_minform}(cg) = \text{apply_sequence}(s)(cg)
\]

The reasoning in choosing this predicate for induction is similar to that used in Section D.1

to_minform_sequence.1 :

\[
\begin{align*}
&\{1\} \ (\text{FORALL} \ (cg: (\text{simple?}), n: \text{nat}) : \\
&\quad n = \text{card}(\text{operators}(cg)) \Rightarrow \\
&\quad \text{EXISTS} (s: \text{oper_tr_sequence}) : \\
&\quad \text{to_minform}(cg) = \text{apply_sequence}(s)(cg))
\end{align*}
\]

\text{AND}

\[
\begin{align*}
&\{1\} \ (\text{FORALL} \ j : \\
&\quad (\text{FORALL} \ (cg: (\text{simple?}), n: \text{nat}) : \\
&\quad n = \text{card}(\text{operators}(cg)) \Rightarrow \\
&\quad \text{EXISTS} (s: \text{oper_tr_sequence}) : \\
&\quad \text{to_minform}(cg) = \text{apply_sequence}(s)(cg)))
\end{align*}
\]

\text{IMPLIES}

\[
\begin{align*}
&\{1\} \ (\text{FORALL} \ (cg: (\text{simple?}), n: \text{nat}) : \\
&\quad n = \text{card}(\text{operators}(cg)) \Rightarrow \\
&\quad \text{EXISTS} (s: \text{oper_tr_sequence}) : \\
&\quad \text{to_minform}(cg) = \text{apply_sequence}(s)(cg))
\end{align*}
\]

\text{AND}

\[
\begin{align*}
&\{1\} \ \text{FORALL} \ (cg: (\text{simple?}) : \\
&\quad \text{EXISTS} (s: \text{oper_tr_sequence}) : \text{to_minform}(cg) = \text{apply_sequence}(s)(cg)
\end{align*}
\]

After propositional simplifications, two subgoals are generated:

\[
\begin{align*}
&\{1\} \ \text{FORALL} \ (cg: (\text{simple?}), n: \text{nat}) : \\
&\quad n = \text{card}(\text{operators}(cg)) \Rightarrow \\
&\quad \text{EXISTS} (s: \text{oper_tr_sequence}) : \\
&\quad \text{to_minform}(cg) = \text{apply_sequence}(s)(cg)
\end{align*}
\]
to_minform_sequence.1.2 :

|-------
{1} FORALL (cg: (simple?), n: nat):
  n = card(operators(cg)) =>
  (EXISTS (s: oper_tr_sequence):
    to_minform(cg) = apply_sequence(s)(cg))
{2} FORALL (cg: (simple?!)):
  EXISTS (s: oper_tr_sequence): to_minform(cg) = apply_sequence(s)(cg)

The first subgoal is discharged after instantiating antecedent formula with ("cg!1" "card(operators(cg!1))"). For the second subgoal we apply simple induction on variable "n" of the first consequent formula. This yields a base case - when the cardinality of the set of operators is zero (the set of operators is empty), and an inductive step. These subgoals are:

to_minform_sequence.1.2.1 :

{-1} 0 = card(operators(cg!1))
|-------
{1} EXISTS (s: oper_tr_sequence):
  to_minform(cg!1) = apply_sequence(s)(cg!1)

to_minform_sequence.1.2.2 :

{-1} FORALL (cg: (simple?!)):
  j!1 = card(operators(cg)) =>
  (EXISTS (s: oper_tr_sequence):
    to_minform(cg) = apply_sequence(s)(cg))
{-2} j!1 + 1 = card(operators(cg!1))
|-------
{1} EXISTS (s: oper_tr_sequence):
  to_minform(cg!1) = apply_sequence(s)(cg!1)

The first subgoal considers the case when the set of operators is empty; this corresponds to the second case of definition of to_minform, where all operator expressions are in the Minimum Operators Form:

forall (e:(filter_op(car(cg)\'exp_set))) : minform_exp?(car(cg),e)

The application of to_minform does not change the initial graph, so the witness sequence to instantiate the consequent consists of only one element, namely iden_tr.

The second subgoal (the inductive step) was proved by:

1) instantiating the first antecedent formula with:
2) skolemize in order to obtain a witness sequence of transformations, s!1.
3) instantiate the consequent with the sequence obtained by appending an OIS transformation to the witness sequent obtained from 2).

To prove to_minform lemma we also used simple induction on natural numbers, and instantiated the inductive hypothesis with the corresponding predicate.

From the above two lemmas, a proof for cg_to_minform was derived using appropriate instantiations. Here is the proof script which contains a list of proof checker commands:

(|cg_to_minform| "" (SKOSIMP*)

("" (LEMMA "to_minform_sequence")
  ("" (INST -1 "cg!1")
    ("" (SKOSIMP*)
      ("" (INST 1 "s!1")
        ("" (ASSERT)
          ("" (LEMMA "to_minform_lemma")
            ("" (INST -1 "cg!1")
              ("" (ASSERT NIL NIL))))))))

164
Appendix F

Strategy for Proving the Existence of an Inverse Sequence of OIS

The proof strategy for the existence of a sequence that performs the inverse of OIS consists of the following steps:

1. Prove that there exists a sequence of invertible operational transformations which results in decreasing the cardinality of the target set of the substituted operator, such that by applying OIS to its result has the same effect as applying OIS to the initial transfer.

```latex
\[
\text{OIS\_constructed: LEMMA}
\]

\[
\forall (rt: \text{wellformed\_rt}, subst, wth: (\text{filter\_op}(rt'exp\_set))) : \text{let cg: c\_graph = cons(rt, null) in}
\]

\[
(\text{singleton?}(\text{targets}(rt, subst)) \text{ or } (\text{precondition\_ois?}(rt, subst, op(wth)) \text{ and } \text{nonempty?}[\text{op\_i\_link}](\text{targets}(rt, subst)) =>
\]

\[
(\exists (s:\text{op\_tr\_sequence}) : \text{length}(\text{apply\_sequence}(s)(cg)) = 1 \text{ and}
\]

\[
\text{eq?}(\text{OIS}(cg, 0, subst, op(wth)),
\]

\[
\text{OIS}(\text{apply\_sequence}(s)(cg), 0, subst, op(wth))) \text{ and}
\]

\[
\text{card}(\text{targets}(rt, subst)) - 1 =
\]

\[
\text{card}(\text{targets}(\text{car}(\text{apply\_sequence}(s)(cg)), subst)) \text{ and}
\]

\[
(\exists (s1:\text{op\_tr\_sequence}) :
\]

\[
\text{eq?}(\text{apply\_sequence}(s1)(\text{apply\_sequence}(s)(cg)), cg) \text{ and}
\]

\[
\text{nonempty?}(\text{targets}(\text{car}(\text{apply\_sequence}(s)(cg)), subst)) \text{ and}
\]

\[
(\text{precondition\_ois?}(\text{car}(\text{apply\_sequence}(s)(cg)), subst, op(wth)) \text{ and}
\]

\[
\text{car}(\text{apply\_sequence}(s)(cg))'exp\_set(wth) \text{ and}
\]

\[
\text{car}(\text{apply\_sequence}(s)(cg))'exp\_set(subst)))
\]

165
For easier proof, this lemma is asserted only for control graphs of length equal to one. The results obtained in this particular case are extended to control graphs of arbitrary length in later steps of this strategy. The last four lines in the OIS\textsubscript{constructed} lemma are needed in the next step, where a sequence of transformations is defined as the recursive concatenation of sequences defined by OIS\textsubscript{constructed}. Since OIS\textsubscript{constructed} is proved for a set of preconditions, its recursive consideration should also satisfy these preconditions. A proof for this lemma was derived from the following sublemmas:

- Prove that after the applying the sequence \langle oc, ois \rangle defined by:
  
  \[
  \text{ois}(\text{oc}(rt, opr, l), \text{op\_copy\_exp}(subst, opr), \text{op}(wth))
  \]
  
  the cardinality of the target set for subst is decremented:

\[
\text{ois\_card: LEMMA}
\]

\[
\forall (rt: \text{wellformed\_rt}, subst, wth: (\text{filter\_op}(rt^\text{exp\_set})), opr: \text{operator}) : \\
( \text{nonempty}(targets(rt, subst)) \land \text{precondition\_ois}(rt, subst, op(wth)) \\
\land \text{not operators}(rt)(opr)) \Rightarrow \\
\text{let } l: \text{op\_i\_link} = \text{choose}(targets(rt, subst)) \text{ in} \\
\text{card}(targets(rt, subst)) - 1 = \\
\text{card}(targets(ois(oc(rt, opr, l), \text{op\_copy\_exp}(subst, opr), op(wth)), subst))
\]

- Prove that subst and wth remain unchanged after applying this sequence.

\[
\text{subst\_same: LEMMA}
\]

\[
\forall (rt: \text{wellformed\_rt}, subst, wth: (\text{filter\_op}(rt^\text{exp\_set})), opr: \text{operator}) : \\
( \text{nonempty}(targets(rt, subst)) \land \text{precondition\_ois}(rt, subst, op(wth)) \\
\land \text{not operators}(rt)(opr) \land \text{opfn}(opr) = \text{opfn}(op(subst)) \land \\
\text{not singleton}(targets(rt, subst))) \Rightarrow \\
\text{let } l: \text{op\_i\_link} = \text{choose}(targets(rt, subst)) \text{ in} \\
\text{new\_exp}(\text{new\_exp}(subst, opr, l), \text{op\_copy\_exp}(subst, opr), \text{op}(wth)) = subst
\]

\[
\text{wth\_same: LEMMA}
\]

\[
\forall (rt: \text{wellformed\_rt}, subst, wth: (\text{filter\_op}(rt^\text{exp\_set})), opr: \text{operator}) : \\
( \text{nonempty}(targets(rt, subst)) \land \text{precondition\_ois}(rt, subst, op(wth)) \\
\land \text{not operators}(rt)(opr) \land \text{opfn}(opr) = \text{opfn}(op(subst)) \land \\
\text{not singleton}(targets(rt, subst))) \Rightarrow \\
\text{let } l: \text{op\_i\_link} = \text{choose}(targets(rt, subst)) \text{ in} \\
\text{new\_exp}(\text{new\_exp}(wth, opr, l), \text{op\_copy\_exp}(subst, opr), \text{op}(wth)) = wth
\]
• Prove that if the targets set of subst has more than one element, then after applying the <oc, ois> sequence of transformations the targets set in nonempty and the precondition for ois stands.

\[
\text{nonempty_precondition: LEMMA}\]
\[
\forall (rt:\text{wellformed}_rt, \text{subst}, wth:\text{filter}_op(rt'\exp\_set), \text{opr}:\text{operator}) : \\
\begin{align*}
&\text{nonempty?(targets(rt, subst)) and precondition\_ois?(rt, subst, op(wth))} \\
&\quad \text{and (not operators(rt)(opr)) and opfn(opr) = opfn(op(subst)) and} \\
&\quad \text{(not singleton?(targets(rt, subst))))} \\
&\Rightarrow \\
&\text{let l:op\_i\_link = choose(targets(rt, subst)) in} \\
&\text{nonempty?(targets(ois(oc(rt, opr, l), op\_copy\_exp(subst, opr), op(wth)), subst))} \\
&\quad \text{and} \\
&\text{precondition\_ois?(ois(oc(rt, opr, l), op\_copy\_exp(subst, opr), op(wth)), subst, op(wth)))}
\end{align*}
\]

• Prove the applying ois to subst and wth in the initial register transfer has the same effect as applying ois to the result of the <oc, ois> sequences, for every operator expression in rt.

\[
\text{same_expression: LEMMA}\]
\[
\forall (rt:\text{wellformed}_rt, \text{subst}, wth:\text{filter}_op(rt'\exp\_set)), \\
\quad e:(rt'\exp\_set), \text{opr}:\text{operator}) : \\
\begin{align*}
&\text{nonempty?(targets(rt, subst)) and precondition\_ois?(rt, subst, op(wth))} \\
&\quad \text{and (not operators(rt)(opr)) and opfn(opr) = opfn(op(subst)) and} \\
&\quad \text{(not singleton?(targets(rt, subst))))} \\
&\Rightarrow \\
&\text{let l:op\_i\_link = choose(targets(rt, subst)) in} \\
&\text{ois.new\_exp(e, subst, op(wth)) =} \\
&\text{ois.new\_exp(ois.new\_exp(ocl.new\_exp(e, opr, l),} \\
&\quad \text{op\_copy\_exp(subst, opr), op(wth)), subst, op(wth)))}
\end{align*}
\]

• Prove the OIS_constructed lemma by instantiating these sublemmas (refering to one register transfer) for the unique register transfer that constitutes cg.

2. Prove that there exists a sequence of invertible operational transformations that results in a transfer where the substituted operator has only one output (target) and such that by applying OIS to its result has the same effect as applying OIS to the initial transfer. This lemma was proved by induction on natural numbers. We induct on the cardinality of the target set for subst, targets(rt, subst):

• In the base case, \( n = 0 \) when targets(rt, subst) is empty. The generated subgoal is automatically discharged since the implication’s precondition is false.
• In the inductive step:
  
  – a subgoal for \( n = 1 \) is proved by instantiating the subsequent formula with the null sequence; in this case, no sequence of transformation is necessary since the register transfer to which OIS is applied is already in a form where the application of OIS admits OC as inverse.
  
  – a subgoal for \( n > 1 \) is proved by instantiating the consequent formula with the concatenation of two sequences: the witness sequence supplied by the inductive antecedent, and a \(<\text{oc}, \text{ois}>\) sequence supplied by the \( \text{OIS\_to\_singleton} \) lemma.

\[
\text{OIS\_to\_singleton: LEMMA}
\]

\[
\text{forall(rt:wellformed\_rt, subst,wth:(filter\_op(rt'exp\_set)))) :}
\]

\[
\begin{align*}
\text{let cg: c\_graph = cons(rt,null) in} & \\
\text{(precondition\_ois?(rt,subst,op(wth))) and} & \\
\text{nonempty?(op\_i\_link)(targets(rt, subst)))} & \\
\text{=>} & \\
\text{(exists(s:op\_tr\_sequence) :} & \\
\text{length(apply\_sequence(s)(cg)) = 1 and} & \\
\text{singleton?(targets(car(apply\_sequence(s)(cg)),subst)) and} & \\
\text{eq?(OIS(cg,0,subst,op(wth)),} & \\
\text{OIS(apply\_sequence(s)(cg),0,subst,op(wth)))) and} & \\
\text{( exists(s1:op\_tr\_sequence) :} & \\
\text{eq?(apply\_sequence(s1)(apply\_sequence(s)(cg)),cg) ) and} & \\
\text{nonempty?(targets(car(apply\_sequence(s)(cg)),subst)) and} & \\
\text{precondition\_ois?(car(apply\_sequence(s)(cg)),subst,op(wth)) and} & \\
\text{car(apply\_sequence(s)(cg))'exp\_set(wth) and} & \\
\text{car(apply\_sequence(s)(cg))'exp\_set(subst))}
\end{align*}
\]

3. Prove by case analysis that OIS applied to a control graph of unit length admits as inverse a sequence of OTs. Three cases deal with substitutions performed with:

• operators that are not in use at the current register transfer.

• operators that are in use in the current register transfer, and the substituted operator expression has only one target.

• operators that are in use in the current register transfer, and the substituted operator expression has more than one target.

In the first two cases the inverse sequence consists of a unique transformation, an OIS in the first case, and OC in the second. In the third case, the corresponding subgoal is discharged by instantiating the consequent formula with the concatenation of two sequences: the witness
sequence supplied by the OIS\textsubscript{to\_singleton} lemma, and the OC transformation as in the second case.

<table>
<thead>
<tr>
<th>OIS_INVERSE: LEMMA</th>
</tr>
</thead>
<tbody>
<tr>
<td>\forall(rt:\text{wellformed_rt}, subst:\text{filter_op}(rt\text{`exp_set})), opr:\text{operator}) :</td>
</tr>
<tr>
<td>( \text{precondition_ois}(rt, subst, opr) \text{ and } \text{clean_rt}(rt) ) \Rightarrow</td>
</tr>
<tr>
<td>\text{let cg: c_graph = cons(rt, null) in}</td>
</tr>
<tr>
<td>\exists(s:\text{op_tr_sequence}) : eq?(apply_sequence(s)(\text{OIS}(cg,0, subst, opr)), cg)</td>
</tr>
</tbody>
</table>

4. Bring this result into the context of the entire control graph using the following sublemma.

<table>
<thead>
<tr>
<th>OIS_replace: LEMMA</th>
</tr>
</thead>
<tbody>
<tr>
<td>\forall(cg: c_graph, n:\text{below}[\text{length}(cg)], subst:\text{filter_op}(\text{nth}(cg,n)\text{`exp_set})), opr:\text{operator}) :</td>
</tr>
<tr>
<td>\text{let cgs: c_graph = cons(nth(cg,n), null) in}</td>
</tr>
<tr>
<td>\text{OIS}(cg,n, subst, opr) = append(start_cg(cg,n), append(OIS(cgs,0, subst, opr), rest_cg(cg,n)))</td>
</tr>
</tbody>
</table>
Appendix G

Formalization and Proofs of RTS Completeness for Scheduling

A schedule is defined in PVS as a function type that assigns natural numbers (corresponding to control steps) to operator expressions. A valid schedule for a set of expressions defines the set of schedules that preserve data dependencies: for every expression in the scheduled set, its sources are assigned to smaller or equal natural numbers (to previous or to the current control step).

\[
\text{schedule}(e\text{-set} : \text{set}[\text{expression}]) : \text{TYPE}^+ = [(e\text{-set})\to \text{nat}]
\]

\[
\text{valid\_schedule}(e\text{-set} : (\text{wellformed\_set}?)) : \text{set}[\text{schedule}(e\text{-set})] = \\
\{\text{vsched} : \text{schedule}(e\text{-set}) \mid (\forall e : (\text{filter\_op}(e\text{-set}))) : \\
\quad (\text{vsched}(\text{source1}(e)) \leq \text{vsched}(e) \text{ and} \\
\quad \text{vsched}(\text{source2}(e)) \leq \text{vsched}(e))\}
\]

The maximum control step generated by a schedule \(s\) for a subset \text{induct\_set} of a well-formed set of expressions \text{e\_set} is defined recursively on the cardinality of \text{induct\_set}. Using only recursive definitions we were able to avoid any axiomatic assertion. The property that no expression in a scheduled set can be assigned a higher control step then \text{max\_step\_recc}(e\text{-set}, s, e\text{-set}) is proved instead of being stated.
The Register Transfer Split (RTS) transformation is formalized by the following function definition for `split`. It admits as inputs a well-formed set of expressions, \( e_{\text{set}} \), a schedule \( s \) for this set, a natural number \( n \) smaller than the maximum control step, and a `split_set` of expressions to be scheduled at control step \( n \), and returns a new schedule for \( e_{\text{set}} \). All expressions not in \( split_{\text{set}} \) previously scheduled at step \( n \) are assigned \( n+1 \), the same for all other expressions previously assigned by \( s \) to steps higher than \( n \).

\[
\begin{align*}
\text{split}(e_{\text{set}} : (\text{wellformed_set}?), \ s : \text{schedule}(e_{\text{set}}), \ n : \text{upto}[\text{max}_\text{step}(e_{\text{set}}, s)], \split_{\text{set}} : \text{set}[(e_{\text{set}})] : \text{schedule}(e_{\text{set}}) = \\
\quad \text{LAMBDA } (x : (e_{\text{set}})) : \\
\quad \quad \text{if } s(x) < n \text{ then } s(x) \\
\quad \quad \text{else if } s(x) = n \text{ then if } \split_{\text{set}}(x) \text{ then } n \text{ else } n+1 \text{ endif else } s(x) + 1 \text{ endif} \end{align*}
\]

The `valid_schedule_lemma` shown below asserts the correctness of `split` if applied in the presence of the weakest precondition: for every expression \( e \) in `split_set`, the sources of \( e \) are either scheduled to previous control steps, or they belong to `split_set`. A proof for `valid_schedule_lemma` was derived using function definitions expansion, instantiations and propositional simplifications.
valid_sched_lemma: LEMMA
forall(e_set:(wellformed_set?), s:(valid_schedule(e_set)),
      n:upto[max_step(e_set,s)], split_set:set[(exp_set_n(e_set,s,n))]) :
      (forall(e:(split_set)) :
       (op?(e) =>
        ((s(source1(e)) < n or split_set(source1(e))) and
         (s(source2(e)) < n or split_set(source2(e)))))
      iff valid_schedule(e_set)(split(e_set,s,n,split_set))

We define the initial schedule of an expression set e_set as the schedule assigning all expressions
in e_set to 0. This corresponds to the initial representation of a design, where all computations
are performed in one control step.

init_schedule(e_set:(wellformed_set?)) : schedule(e_set) = LAMBDA (x:(e_set)) : 0

For a clearer formalization of the completeness property, we define a split_transform(e_set) predicate function subtype dependent on e_set. valid_split?(e_set) is the set of all split transformations that preserves the validity.

split_transform(e_set:(wellformed_set?)) : TYPE =
{s_tr:schedule(e_set)->schedule(e_set) |
 exists(s:schedule(e_set),n:upto[max_step(e_set,s)],split_set:set[(e_set)]) :
  s_tr(s) = split(e_set,s,n,split_set)}

valid_split?(e_set:(wellformed_set?)) : [split_transform(e_set)->bool] =
LAMBDA (x:split_transform(e_set)) :
      forall(s:(valid_schedule(e_set))) : valid_schedule(e_set)(x(s))

The next_schedule_lemma1 states that for any valid schedule s associated with a well-formed
set e_set and having length n, either s is the initial schedule, or there exists a unique set of
expressions split_set and a unique schedule s1 from which s can be obtain by splitting the last
set of expressions scheduled at the maximum control step.
The completeness property for the `split` transformation with respect to scheduling is expressed by the `split_complete` lemma below. It was proved by induction on the maximum control step of a schedule, and by appropriately instantiating `next_schedule_lemma1`.

```plaintext

next_schedule_lemma1: LEMMA
forall(e_set:(wellformed_set?), s:(valid_schedule(e_set))) :
    s = init_schedule(e_set) or
    exists1!(s1:(valid_schedule(e_set)),split_set:set[(e_set)]) :
        (s = split(e_set,s1,max_step(e_set,s1),split_set)
         and (max_step(e_set,s) > 0 => max_step(e_set,s1) = max_step(e_set,s) -1))

split_complete: LEMMA
forall(e_set:(wellformed_set?), s:(valid_schedule(e_set))) :
    exists (s_sqe:list[(valid_split?(e_set))]) :
        s = apply_sequence(e_set, init_schedule(e_set), s_sqe)
```

173
Bibliography


