Formal Derivation
of A Class of Computers

Li-Guo Wang

Doctor of Philosophy
University of Edinburgh
November 1994
Abstract

The aim of this thesis is to investigate how to use logic-based specification, construction, and proof methods to formally derive a class of computers.

Differing from the traditional concepts of specification, verification, and synthesis, the emphasis will be on the formal design process and the notion of derivation. A derivation is a stepwise specification, construction, and proof process. The specification of derivation is a set of specifications which is described by abstract syntax capturing the common structure of the specifications in the set. The construction of derivation is a mapping from the set of specifications to the power set of specifications. The proof of derivation is a guarantee that each construction is total and each constructed specification is correct.

Computers are among the most complex hardware devices. Formal verification researchers have often used computers as their goals or case studies. But there are few reports of the formal construction and proof of a computer. There is no report is about the formal derivation of a class of computers.

In this thesis we present the concepts and methods of derivation, which includes, mainly, abstract-syntax-based specification schemes for hierarchical specifications, formal and nearly mechanical construction, and abstract and general verification for a class of computers. The central concept is the unified derivation model, developed from the predicate model, used to describe the whole derivation process. The central method is logic structural refinement which, complements behavioral, data, temporal, and (spatial) structural abstraction. It concentrates on describing the main aspect of construction: logic decomposition. The key point is the concept of an entry predicate which reveals the basic relationship in logic and time between a specification and its sub-specifications.

We use state-transition-based abstract syntax as the behavior specification scheme to describe a class of computers at the machine instruction level. Starting from the scheme we take 9 steps of constructions and proofs to derive and ob-
tain structural specifications at register transfer level. Each individual step, and thereby the whole construction, is proved to be total and correct.

Our work develops the basic viewpoint that the hierarchical architecture of a computer can be organized through a dynamic formal construction and proof process, in a three-dimensional structure of logic, space and time. The aspect of relation and construction in logic is fundamental and novel. In addition to the traditional understanding of microprograms our work brings up three new levels in microprogramming: global loop, abstract microprogram structure, and linear microprogram. Also, there is a sequence of constructions of time layers in which the usual verification based on two time layers is only a special case. As a corollary we discuss no-clock, variable clock, and constant clock machines.

Gordon’s computer is taken as a case study to illustrate our concepts and methods. We derive three different implementations and compare them with the original one of Gordon’s paper.
Acknowledgments

I would like to thank first of all my supervisor, Professor Mike Fourman, for his encouragement and hints in the development of this thesis, especially, I am grateful to him for his kind help for the chance to come Edinburgh University for my PhD study. I would also like to thank to my second supervisor, Professor Rod Burstall, for his kind help for my scholarship and for his careful checking of parts of this thesis.

I am grateful to Siemens AG Munich for providing scholarship and SERC for further financial support under grant GR/F 35890.

I would like most sincerely to thank Dr Michael Mendler for his sincere encouragement and for proof reading my whole thesis (first version, near 300 pages) word by word, pointing out some mistakes, and improving my English. This is a quite hard work and has taken a lot of his time. Without his kind help I could not have the present thesis.

I would like to thank to Dr Zhaohui Luo for his warmhearted encouragement and kind help during my whole PhD study in Edinburgh and to Dr Liang Chen for his sincere helps.

I own special thanks to Dr Imre Kayhan for his warmhearted help for me to use Macintosh, to Dr Kees G. W. Goossens for his help to check of parts of this thesis, to Miss Eleanor Kerse and Mrs Lorraine Edgar for their many kind helps.

I own sincere thanks to the National Science Foundation of China, Professor Edsger W. Dijkstra, Professor C.A.R. Hoare, Professor Jieh Hsiang, Professor Jifeng He for their helps and invitations which finally lead to my PhD study in LFCS, Department of Computer Science of Edinburgh University.

I own special thanks to Dr Mike Gordon for his providing HOL system, to Dr Ranga Vemuri and Dr Phillip J. Windley for their providing relevant reference materials.
Declaration

This thesis has been composed by myself. The work reported herein, except where indicated in the text, is my own, and has not been presented for any university degree before.

Li-Guo Wang
# Table of Contents

I  Introduction and Concepts ........................................ x

1. Introduction .................................................. 2
   1.1 Motivation ................................................. 2
   1.2 The Contribution of This Work .......................... 3
   1.3 Structure of the Thesis .................................... 4

2. Structure, Construction and Proof ............................ 7
   2.1 Structure ..................................................... 7
   2.2 Construction ............................................... 9
   2.3 Proof .......................................................... 10

3. Derivation Model ............................................... 12
   3.1 Derivation Model ......................................... 13

II  Formal Derivation of a Class of Computer ................. 17

4. Behavior Specification Scheme ................................. 19
   4.1 Syntax ..................................................... 19
4.2 Semantics .................................................. 21
4.3 Constraint .................................................. 22

5. Refining Time ........................................... 25
  5.1 Time in Construction .................................... 25
    5.1.1 Constructing Time .................................... 25
    5.1.2 Time Based on Derivation Model .................... 26
  5.2 Making Time Computable ................................ 27
  5.3 Changing Time Layer .................................... 28
    5.3.1 Construction 2 and Specification Scheme 2 ........ 30
    5.3.2 Totalness and Correctness of Construction 2’ and 2 .... 33
    5.3.3 Discussion: Time Refinement and Predicate $E$ ........ 36

6. Selecting Global Structure .............................. 37
  6.1 Setting State Transition ............................... 38
    6.1.1 Construction 3 and Specification 2’ and 3 .......... 38
    6.1.2 Totalness and Correctness of Construction 3 .......... 40
  6.2 Selecting Common Sub.State Transition ................ 42
    6.2.1 Construction 4 and Specification Scheme 3’ and 4 .... 43
  6.3 Selecting Parallel Term Transition .................... 47
    6.3.1 Construction 5 and Specification Scheme 4’ and 5 .... 51

7. Decomposing to Term Transition ......................... 53
  7.1 Construction 6, Abstract Algorithm 6 and Specification Scheme 5’
    and 6 .................................................... 60
Table of Contents

7.2 Totalness and Correctness of Construction 6 ................. 74

8. Decomposing to Function Unit and Connection ................... 82
  8.1 Construction 7 and Specification Scheme 6' and 7 ............. 84
    8.1.1 Specification 6' ........................................ 84
    8.1.2 Linear Chain ............................................ 85
    8.1.3 Construction 7 ............................................ 88
    8.1.4 Specification Scheme 7 .................................. 104
  8.2 Totalness and Correctness of Construction 7 ................... 106

9. Introducing Register, Bus and Microinstruction .................. 110
  9.1 Introducing Bus, Register and Connection .................... 116
    9.1.1 Introducing Bus ........................................... 116
    9.1.2 Introducing Register ..................................... 120
    9.1.3 Introducing Connection ................................... 122
    9.1.4 Abstract Algorithm 8, Construction 8 and Specification Scheme
         8 ............................................................ 124
  9.2 Introducing Datapath and Microprogram ......................... 128
    9.2.1 Introducing Datapath and Microprogram .................... 130
    9.2.2 About Clock ............................................... 132
    9.2.3 Implied Zero Time Point ................................... 135
    9.2.4 Abstract Algorithm 9 and Specification Scheme 9 ........... 136

10. Structural Specification Scheme ................................. 140
  10.1 Structural Specification Scheme ............................... 140
# Table of Contents

## III Case Study and Further Discussion

<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>11.1</td>
<td>Introduction to Gordon’s Computer</td>
<td>144</td>
</tr>
<tr>
<td>11.2</td>
<td>Formal Derivation</td>
<td>146</td>
</tr>
<tr>
<td>11.2.1</td>
<td>Behavior Specification</td>
<td>146</td>
</tr>
<tr>
<td>11.2.2</td>
<td>Refining Time</td>
<td>148</td>
</tr>
<tr>
<td>11.2.3</td>
<td>Setting State Transition</td>
<td>149</td>
</tr>
<tr>
<td>11.2.4</td>
<td>Resetting Global Structure</td>
<td>152</td>
</tr>
<tr>
<td>11.2.5</td>
<td>Decomposing to Term Transition</td>
<td>155</td>
</tr>
<tr>
<td>11.2.6</td>
<td>Decomposing to Function Unit and Connection</td>
<td>160</td>
</tr>
<tr>
<td>11.2.7</td>
<td>Introducing Bus, Connection and Microinstruction</td>
<td>171</td>
</tr>
<tr>
<td>11.2.8</td>
<td>Structural Specification</td>
<td>183</td>
</tr>
<tr>
<td>11.3</td>
<td>Deriving other Implementations</td>
<td>188</td>
</tr>
<tr>
<td>11.3.1</td>
<td>Two-Bus Implementation</td>
<td>189</td>
</tr>
<tr>
<td>11.3.2</td>
<td>One-Bus Implementation</td>
<td>190</td>
</tr>
<tr>
<td>11.4</td>
<td>Discussion</td>
<td>194</td>
</tr>
</tbody>
</table>

## 12. Proof-Based Architecture of Computer

<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
</table>

## 13. Logic, Space and Time of Hardware Construction

<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>13.1</td>
<td>Logic, Space and Time Structure of Model</td>
<td>200</td>
</tr>
<tr>
<td>13.2</td>
<td>Logic, Space and Time Structure of Construction</td>
<td>202</td>
</tr>
<tr>
<td>13.2.1</td>
<td>Logic Structure of Construction</td>
<td>204</td>
</tr>
</tbody>
</table>
Table of Contents

13.2.2 Space Structure of Construction ...................... 204
13.2.3 Time Structure of Construction ...................... 205
13.3 Case Study ......................................... 207
13.4 Conclusion ........................................ 207

14. Related Work ........................................... 209
14.1 Computer Verification .................................. 209
14.2 Formal Synthesis ..................................... 212
14.3 Other Relevant Work .................................. 214

15. Conclusions and Further Work ............................ 216
15.1 Conclusions .......................................... 216
15.2 Further Work ......................................... 219
  15.2.1 Mechanical Design System ......................... 219
  15.2.2 Mathematical Foundation of Hardware Design .... 220
Part I

Introduction and Concepts
In this part we introduce the concept of derivation, and describe its three aspects: structure, construction and proof. As a foundation for the formal derivation of a class of computers we present a derivation model for formal hardware construction.
Chapter 1

Introduction

In this chapter we introduce the concept of derivation. We discuss two aspects: motivation, and the contribution of this work.

1.1 Motivation

Formal hardware verification is becoming more and more accepted as a means of increasing confidence in the correctness of hardware design. Two aspects are notable: formal methods and experimental case studies, especially, the verification of microprocessors. What we obtain is, in essence, a clearer understanding of the relation between two extremes, specification and implementation, through applying rigorous mathematics.

The inherent complexity of hardware systems stimulates synthesis research, as post-design verification is difficult to deal with [26], [36]. However, synthesis is a somewhat ambiguous concept. In fact, there are several different understandings for synthesis, for example, synthesis as algorithmic optimization [60]; synthesis as a design methodology [27]; synthesis as a logic of design [36]; synthesis as case study [6], [73].

In this thesis instead of the word “synthesis” we use the word “derivation”.

2
Definition 1.1.1. [derivation]:

Derivation is a formal methodology to deal with the specification, construction, and proof of a class of objects.

More concretely, by formal derivation of a class of computers, we mean: using structure (abstract syntax) to describe a computer, using formal construction to describe the design of a computer, and using generic proof to guarantee the totalness and correctness of the construction. We emphasize the application of a formal method to solve a realistic, complex problem.

1.2 The Contribution of This Work

We present an overall specification, construction, and proof process to cover the formal design and verification of a class of computers, from behavioral specification at the machine instruction level, to structural implementation at the register transfer level. The general process is illustrated through a case study of Gordon’s computer.

We present a new, five-level hierarchy for derivation of a microprogram through a dynamic, top-down process of construction and proof:

1. Global loop

2. Abstract microprogram structure

3. Linear microprogram

4. Microinstruction

5. Microcode
Chapter 1. Introduction

We present proof-based computer architecture as a new understanding for computer architecture.

Our main conceptual and methodological contribution is the derivation model. This establishes a foundation for the construction of hardware, and provides a concise and clear picture of the complicated process of computer design. The entire derivation of a class of computers is based on the model. Furthermore based on the model we present a new understanding for the logical, spatial, and temporal aspects of formal hardware construction.

1.3 Structure of the Thesis

The thesis consists of three parts: Part 1, introduction and concepts, Part 2, formal derivation of a class of computers and Part 3, case study and further discussion.

Part 1 consists of three chapters, of which the first is just this introduction. In Chapter 2 we describe the main concepts and methods of the thesis. They include structure, construction, and proof. In Chapter 3 we present the concepts of derivation model and entry predicate. These concepts are the basis of our derivation methods.

Part 2 is the main part of this thesis and studies the formal derivation process of a class of computers. It consists of the seven chapters 4 - 10, which are summarized below.

In Chapter 4 we define exactly the behavior specification scheme for describing a class of computers at machine instruction level.

In Chapter 5 we discuss time refinement. This includes two construction steps: making time computable, and changing time layers. These two construction steps are general, and in fact they are suitable for any level of time and time transformation.
In Chapter 6 we discuss the construction involving selecting the global structure at the behavioral specification level. These selections, especially ones for parallel and sequential execution, are important steps of the design. They will deeply influence the following design.

In Chapter 7 we have an important step of formal derivation: decomposition from the behavioral specification level to the term transition level. We establish, through exact construction and proof, a term transition structure as a general, high-level computer architecture which corresponds to an abstract microprogram structure.

In Chapter 8 we further decompose the term transition structure to function unit and connection level. We present the concept of linear chain and a general proof-based construction process which can act as a framework covering a wide range of allocation and scheduling strategies. An optimization algorithm can be represented within the framework through the selections used in the construction.

In Chapter 9 we discuss a generic process to introduce Bus, Register and Connection to get datapath and microprogram. Meanwhile we prove the zero-time entry predicate $E 0$ can be eliminated.

In Chapter 10 we discuss structural specification schemes.

Part 3 includes five chapters. In Chapter 11, we have a case study. the formal derivation of Gordon’s computer from its machine instruction specification to three different structural implementations at the register transfer level. We discuss and compare the derived machine and [50]’s verified machine. The case study nicely illustrates our concepts and methods of formal derivation. In Chapter 12 we present a new concept of proof-based computer architecture and discuss its basic structure. In Chapter 13 we present and discuss the fundamental concepts: the logic, space and time of hardware construction. We demonstrate that, based on our construction model, we can set up a complete new theoretical system for formal construction of hardware. In Chapter 14 we describe related work about
computer verification and formal synthesis. In Chapter 15 we give the conclusions of the research work of the Thesis and discuss some further research work.
Chapter 2

Structure, Construction and Proof

In this chapter we discuss the concepts of structure, construction and proof on which our formal derivation is based.

2.1 Structure


In this thesis we use “structure” to mean a collection of objects. For the objects the emphasis is on the relations among their parts while any further detail that might be known of these parts is ignored. We take “structure” as a means to describe the common points of a class of objects and their essential aspects.

Jeff Joyce verified a concrete computer, Tamarack-3, and generalized his method [53] by using the concept of “uninterpreted data types and uninterpreted primitives” for a generic specification. He emphasizes that “generic specification plays a dominant role” in his dissertation. We deal with a more general case: a class
of computers versus a concrete computer, and derivation (construction + verification) versus verification. Instead of a generic specification, we deal with a class of objects, and use structure to describe the class.

For the computer construction we call the structure the specification scheme, to emphasize that it, as a scheme, can be instantiated to get a set of specifications. We adopt the set-theoretic abstract syntax approach, BNF, [67] to describe specification schemes. Let us illustrate BNF through an example.

We define a term using BNF as below:

\[
\begin{align*}
\text{Specification Scheme:} & & \text{Abbreviation:} \\
\text{tm} ::= & \ c & \text{tm : term} \\
& | \ v & \text{v : variable} \\
& | \ f \ (tm, \ldots, tm) & \text{f : function} \\
& | \ c & \text{c : constant}
\end{align*}
\]

In BNF the symbol “::=” is read as “may be composed of”. The symbol “|” is read as “or”. In the example, under “specification scheme” we write the BNF rule that a term “tm” may be a constant “c”, a variable “v” or a function “f” whose arguments recursively are terms “tm”. Under “abbreviation” we write syntax domain, for example, “v : variable” that v is a variable. A symbol string is nonterminal if it has an appearance in the left of ::=, otherwise, terminal. For example, tm is a nonterminal and c, v and f are the terminals.

In this thesis we use a convenience that the terminals and nonterminals of a specification scheme can be inherited by another specification scheme. Therefore, it is possible that a terminal (a nonterminal) in a specification scheme may become nonterminal (terminal) in the other specification scheme. For example, term tm can be inherited by another scheme where it becomes terminal as we do not repeat its definition. The constant c may become nonterminal if in another scheme it is defined is the boolean constant T or F.
2.2 Construction

Construction is a transformation from a specification to a specification. The example below is a construction $\Psi$ that the specification body $sb$ of a specification scheme is transformed to the state transition structure $sts$ of another specification scheme.

<table>
<thead>
<tr>
<th>Specification body $sb$:</th>
<th>Abbreviation</th>
</tr>
</thead>
<tbody>
<tr>
<td>$sb ::= os = iss$</td>
<td>$sb$ : specification body</td>
</tr>
<tr>
<td>$iss ::= is$</td>
<td>$iss$ : input state structure</td>
</tr>
<tr>
<td>$\mid$ let $A$ in $iss$</td>
<td>let $A$ in structure</td>
</tr>
<tr>
<td>$\mid P \Rightarrow iss \downarrow iss$</td>
<td>condition structure</td>
</tr>
<tr>
<td>$os$ : output state</td>
<td></td>
</tr>
<tr>
<td>$is$ : input state</td>
<td></td>
</tr>
<tr>
<td>$A$ : assignment</td>
<td></td>
</tr>
<tr>
<td>$P$ : predicate</td>
<td></td>
</tr>
</tbody>
</table>

Construction:

2.2.1. $\Psi (os = is) \equiv os = is$

2.2.2. $\Psi (os = (let A in iss)) \equiv let A in \Psi (os = iss)$

2.2.3. $\Psi (os = (P \Rightarrow \alpha_1 \downarrow \alpha_2)) \equiv P \Rightarrow \Psi (os = \alpha_1) \downarrow \Psi (os = \alpha_2)$

State transition structure $sts$: | Abbreviation:

<table>
<thead>
<tr>
<th>$sts ::= st$</th>
<th>$sts$ : state transition structure</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\mid$ let $A$ in $sts$</td>
<td></td>
</tr>
<tr>
<td>$\mid P \Rightarrow sts \downarrow sts$</td>
<td></td>
</tr>
<tr>
<td>$st ::= os = is$</td>
<td>$st$ : state transition</td>
</tr>
</tbody>
</table>
By the construction equation 2.2.1 the \( os = is \) of \( sb \) is transformed to the same one of \( sts \). By the construction equation 2.2.2 the \( os = (\text{let } A \text{ in } iss) \) of \( sb \) is transformed to the let \( A \) in \( \Psi (os = iss) \) of \( sts \), and similarly for \( P \Rightarrow \alpha_1 \downarrow \alpha_2 \).

We write construction with the Prolog style \([12]\) by two aspects as followings.

**Definition 2.2.1. [match of construction]:**

Given a construction equation \( \Psi (x_1, \ldots x_n) \equiv y \) and a object \( < z_1, \ldots z_n > \) of the same tuple \( (n \geq 1) \). we say they are matchable if for each pair \( x_i \) and \( z_i \) the \( x_i \) is either \( z_i \) or is a variable which is detailed by a definition 6.1.1.1, (see page 39).

[ ]

**Definition 2.2.2. [order of construction]:**

The construction equations of construction are taken as an ordered sequence from 1 to \( n \)

\[ eq_1, eq_2, \ldots, eq_n \]

To construct an object \( < z_1, \ldots z_n > \) is always to check against the order to find a matchable construction equation and then to deal with produced new objects. (As our construction is total such a equation can be always found).

[ ]

For example, in the construction \( \Psi \) each construction equation matches a case of \( sb \) and the construction is done along the order from 1 to 3.

### 2.3 Proof

The domain of a construction is defined by a scheme. A construction is required to be total, namely, it can cover each instance specification matching this scheme. Also a construction is required to be correct, namely, each constructed child specification certainly satisfies its parent specification.

**Definition 2.3.1. [totalness of construction]:**

A construction \( \Psi \) of scheme \( S \) is total if we can prove
\[ \vdash \forall s : S. \Psi(s) \in S' \]

Here and elsewhere, we write \( S \) to represent the set of all specifications which can be produced by the specification scheme \( S \) and we add prime to express \( S' \) is the child scheme of the scheme \( S \) and similarly \( S' \).

**Definition 2.3.2. [correctness of construction]:**

A construction \( \Psi \) of scheme \( S \) is correct if we can prove

\[ \vdash \forall s : S. \Psi(s) \Rightarrow s \]

The main proof method is structural induction [9] as a specification scheme is an abstract syntax structure.
Chapter 3

Derivation Model

In this chapter we present a derivation model which is the foundation of the formal derivation of a class of computers.

In formal hardware verification, a widely accepted and successfully used hardware model [32], which we call the verification model, can be characterized by three features:

1. Component and ports are represented by predicates and free variables.

2. Internal port connection is realized by existential quantification.

3. Composition of components is achieved by logical conjunction.

For example, the delay, inverter and delay-inverter circuits are defined as

Delay \((i, x) \equiv \forall t. x (t + 1) = i t\)

inverter \((x, o) \equiv \forall t. o (t + 1) = \neg(x t)\)

Delay-inverter \((i, o) \equiv \forall t. o (t + 2) = \neg(i t)\)

The structure of the delay-inverter circuit below

\[ \begin{array}{c}
\text{i} \\
\text{Delay} \\
\text{x} \\
\text{Inverter} \\
\text{o}
\end{array} \]
is described as

\[
\text{Delay-Inverter } (i, o) \equiv \exists x. \text{Delay } (i, x) \land \text{Inverter } (x, o)
\]

The verification model concerns the structural abstraction of hardware as verification is a process starting from a given implementation. This is as expressed in [57] (pp. 269): “The type of abstraction most fundamental to hardware verification is structural abstraction – the suppression of information about a device’s internal structure.” The verification model is less suitable for the construction of hardware, particularly, for the construction of microprogrammed computers for this we present a new hardware model, the \textit{derivation model}.

### 3.1 Derivation Model

In contrast to verification, derivation is a process starting from an abstract behavioral specification to construct the structural implementation. In this process the main aspect is logical, structural refinement.

**Derivation Model of Hardware 3.1.1:**

Our framework for specifying hardware behavior is a predicate of the shape

\[
R \left( \vec{i}, \vec{o} \right) [E_i, E_o] \equiv \forall x. \exists y. E_i x \Rightarrow R \left( \vec{i} x, \vec{o} y \right) \land E_o y
\]

Here, \(E_i\) and \(E_o\) are free predicates variables with a time argument. \(E_i\) is called entry predicate and \(E_o\) is called leaving predicate. \(R \left( \vec{i}, \vec{o} \right)\) is a predicate with free variable argument vectors \(\vec{i}\) and \(\vec{o}\) for a hardware device and its input and output ports. We write \(\vec{i} x\) to represent \(i_1 x, \cdots, i_n x\) and similarly for \(\vec{o} y\). Corresponding to \(E_i\) and \(E_o\) we call \(x\) and \(y\) entry and leaving time points respectively.

As a convenience we call the predicate \(R\) or \(R \left( \vec{i}, \vec{o} \right)\) the specification body and call \([E_i, E_o]\) logical boundary. For convenience in this thesis we often, instead of \(R \left( \vec{i}, \vec{o} \right) [E_i, E_o]\), write \(R \left( \vec{i} x, \vec{o} y \right) [E_i, E_o]\) as the latter form is easier to express.
Chapter 3. Derivation Model

With the model an inverter is written as

\[
\text{Inverter } (i, o) [E_i, E_o] \equiv \forall x. \exists y. E_i x \Rightarrow o y = \neg(i x) \land E_o y
\]

In this model, a hardware device is viewed as a process from a time point \( x \) to another time point \( y \). We use entry and leaving predicates \( E_i \) and \( E_o \) to express the controls to enter and leave the process, therefore, the entry predicate also becomes the logical condition to activate a hardware device. In the later chapters we will see that for a microprogrammed computer entry predicates will be instantiated to become predicates about the entry addresses of its microprogram. We existentially quantify the \( y \) to leave it to be determined by construction.

Let us take an example. Suppose in a construction process we have a sub-specification:

\[
\forall x. \exists y. E_i x \Rightarrow \text{receiver } y = \text{ADD (sender}_1 x, \text{sender}_2 x) \land E_o y. \quad (1)
\]

This requires us, in the time interval \( < x, y > \) and under the condition \( E_i \), to implement an add function and thereafter to ensure the condition \( E_o \). To construct (1) following its structure naturally we introduce two new entry predicates \( E_a \) and \( E_b \) and three new ports \( \text{input}_1, \text{input}_2 \) and \( \text{output} \).

\[
(\forall x. \exists y. E_i x \Rightarrow \text{input}_1 y = \text{sender}_1 x \land \text{input}_2 y = \text{sender}_2 x \land E_a y) \quad (2.1)
\]

\[
\land
\]

\[
(\forall x. \exists y. E_a x \Rightarrow \text{output } y = \text{ADD (input}_1 x, \text{input}_2 x) \land E_b y) \quad (2.2)
\]

\[
\land
\]

\[
(\forall x. \exists y. E_b x \Rightarrow \text{receiver } y = \text{output } x \land E_o y) \quad (2.3)
\]

Then, introduce two kinds of device, Register and ALU:

\[
\text{Register } (c, i, o) \equiv \forall t. o(t + 1) = (c t \Rightarrow i t \mid o t)
\]

\[
\text{ALU } (c, i_1, i_2, o) \equiv \forall t. o(t + 1) = (c t \Rightarrow \text{ADD (i}_1 t, i_2 t) \mid i_1 t)
\]

we can further get an abstract microprogram

\[
(\forall t. E_i t \Rightarrow c_{s1} t \land c_{s2} t \land E_a (t + 1)) \quad (3.1)
\]

\[
\land
\]
(\forall t. E_a \ t \Rightarrow c_a \ t \land E_b \ (t + 1))
\land
(\forall t. E_b \ t \Rightarrow c_r \ t \land E_o \ (t + 1))

and a datapath

Register \((c_{s1}, \text{sender}_1, i_1)\) \land Register \((c_{s2}, \text{sender}_2, i_2)\) \land
ALU \((c_a, i_1, i_2, o)\) \land Register \((c_r, o, \text{receiver})\)

We can further instantiate the entry predicates of the abstract microprogram to get part of a concrete microprogram.

\[(\forall t. \text{address} \ t = 12 \Rightarrow c_{s1} \ t \land c_{s2} \ t \land \text{address} \ (t + 1) = 13)\]
\land
\[(\forall t. \text{address} \ t = 13 \Rightarrow c_a \ t \land \text{address} \ (t + 1) = 14)\]
\land
\[(\forall t. \text{address} \ t = 14 \Rightarrow c_r \ t \land \text{address} \ (t + 1) = 15)\]

Finally, we get an implementation as the figure below.

![Diagram of microprogram](image)

We take the specification (1) for our example because a similar form is generally used in the derivation process, and the form is representative.

From this example we can comprehend that the main means of construction is to introduce new entry predicates and new ports. We have an assumption:

**Assumption 3.1.1. [existential variable as new one]:**

The new entry predicates and new signals (as ports) introduced in construction process are always existential variables.
Existential variables give us more freedom for further construction until at the last step, to get a concrete implementation, we instantiate these variables.
Part II

Formal Derivation of a Class of Computer
This part is the main content of this thesis. In it we discuss a logic-based process of specification, construction, and verification for a class of computers, from behavioral specification at machine instruction level to structural specification (as implementation) at register transfer level. The process includes five stages:

1. refining time,
2. selecting global structure,
3. decomposing to term transition,
4. decomposing to functional unit and connection,
5. introducing clock.

In conclusion, we present and discuss a concept of proof-based computer architecture.
Chapter 4

Behavior Specification Scheme

In this chapter we define the generic form of behavioral specification for a class of computers, at machine instruction level. This form is the starting point of the whole derivation. We describe three aspects: syntax, semantics, and constraints. An example of a behavioral specification scheme for Gordon’s computer can be found in part 3, the case study.

4.1 Syntax

In this section we present the syntax of behavioral specification schemes for a class of computers.

To describe a particular computer at machine instruction level we use state transitions as the semantics of its machine instructions. To describe a class of computers we use abstract syntax to describe a general structure of the state transitions. This is an abstract structure because its terminal symbols are uninterpreted, however, the relationship of the symbols is clearly defined. This is a scheme for a class of computers because the construction and verification of any concrete computer specified by instantiating the scheme will follow the construction and verification of the scheme.
Chapter 4. Behavior Specification Scheme

Specification Scheme:  

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>( spsc )</td>
<td>specification scheme</td>
</tr>
<tr>
<td>( sb )</td>
<td>specification body</td>
</tr>
<tr>
<td>( os )</td>
<td>output state</td>
</tr>
<tr>
<td>( sos_i )</td>
<td>the ( i^{th} ) sub output state</td>
</tr>
<tr>
<td>( iss )</td>
<td>input state structure</td>
</tr>
<tr>
<td>( A )</td>
<td>assignment</td>
</tr>
<tr>
<td>( P )</td>
<td>quantifier free formula</td>
</tr>
<tr>
<td>( tm )</td>
<td>term</td>
</tr>
<tr>
<td>( v )</td>
<td>variable</td>
</tr>
<tr>
<td>( sg u )</td>
<td>state signal</td>
</tr>
<tr>
<td>( f (tm, \ldots, tm) )</td>
<td>function</td>
</tr>
<tr>
<td>( \circ )</td>
<td>binary relation</td>
</tr>
<tr>
<td>( u )</td>
<td>time point</td>
</tr>
<tr>
<td>( p )</td>
<td>predicate</td>
</tr>
<tr>
<td>( c )</td>
<td>constant</td>
</tr>
</tbody>
</table>

Abbreviation:

\( spsc \) : specification scheme

\( sb \) : specification body

\( os \) : output state

\( sos_i \) : the \( i^{th} \) sub output state

\( iss \) : input state structure

\( is \) : input state

\( sis_i \) : the \( i^{th} \) sub input state

\( A \) : assignment

\( P \) : quantifier free formula

\( tm \) : term

\( v \) : variable

\( sg \) : state signal

\( f \) : function

\( \circ \) : binary relation

\( u \) : time point

\( p \) : predicate

\( c \) : constant

It is somewhat interesting but less practical that we could define and discuss a more generic and abstract computer. It is a tradeoff between abstract structure
and practical computer that we select the behavior specification scheme above. The important point is that such a scheme must be constructible. In fact, the work of the thesis can be understood as a proof that this quite generic behavior specification scheme is constructible. And we will construct it in this thesis.

4.2 Semantics

A behavioral specification scheme, of a computer, describes transitions to an output state $os$ from different input states $is$; the transition can be taken at any time point $u$. Each transition corresponds to a machine instruction and expresses the semantics of the instruction.

The output and input states have the same number $n$ of sub-output states $sos_i$ and sub-input $sis_i$. The sub-output state $sos_i$ has a simple structure: it is the value of a signal $sg$ at a time point $u + 1$. By the time point $u + 1$ we mean a moment at which a new state of the computer has been produced, namely, the execution of an instruction has been completed. This also means the 1 of $u + 1$ is a time unit for the execution of a machine instruction.

In contrast, a sub-input state $sis_i$ has a more complicated structure. In fact, it may be any term $tm$. A term $tm$ can be a constant $c$, a variable $v$ or the value of a signal $sg$ at a time point $u$. Moreover, a term $tm$ can be also constructed by applying a function $f$ to terms, so that we can obtain arbitrarily complex terms through recursion. The time point $u$ is the moment at which a new state has been established and an instruction starts its state transition from the state.

All input states are organized to form an input state structure $iss$. The $iss$ can recursively lead to any complicated structure. Its base case is just an input state $is$. Its step cases are constructed through the let in structure “let $A$ in $iss$” and the condition structure “$P \Rightarrow iss \downarrow iss$”. In the let in structure we have an assignment $A ::= v = tm$ which declares a simple variable $v$ to replace a complicated term
tm used in iss. By the condition structure “$P \Rightarrow iss_1 \downarrow iss_2$” we mean that if $P$ is true then $iss_1$ else $iss_2$. Here the predicate part $P$ is a quantifier-free formula whose base case is an atomic predicate $p\, (tm, \cdots, \, tm)$ and, through step cases, $P$ can be any complex predicate formula in the logic connectives $\neg, \land, \lor, \Rightarrow$, and $\iff$.

Throughout the thesis we call a signal in the behavior specification of a computer a state signal of the computer. Instead of $P \Rightarrow R \mid S$ we write the condition structure as $P \Rightarrow R \downarrow S$, namely, we use “$\downarrow$” to replace “$\mid$”. This is to avoid confusion with the “$|$” of BNF.

### 4.3 Constraint

To make the construction for let-in structure feasible, and to get an effective implementation, we have constraints for a let-in structure, let $v = tm$ in $S$, in the behavior specification scheme.

**Constraint 4.3.1. [let-in structure]:**

- The variable $v$ must have two or more appearances in its sub-structure $S$. Otherwise the let-in structure will be unnecessary.

- All variables of let-in structures are distinct from each other and each variable is only used once. So, a redundant structure such as

  let $v = 5$ in let $v = 9$ in $S$

  can be excluded.

- The variable $v$ only appears in the sub-structure $S$, which we call well-ordered. By the constraint, recursive assignments, for example below, can be avoided

  let $u = v$ in let $v = u$ in $S$
as \( v = u \) is not well-ordered.

Based on the constraints we have a theorem about the structure of sequential assignment let-in structure, which follows. Before discussing the theorem we introduce three definitions.

Definition 4.3.1. [sequential assignment let-in structure]:

A sequential assignment let-in structure is formed as

\[
\text{let } v_1 = tm_1 \text{ in let } v_2 = tm_2 \text{ in } \cdots \text{ let } v_n = tm_n \text{ in } S
\]

Definition 4.3.2. [independent assignments]:

A group of assignments formed as \( v_j = tm_j \) \((1 \leq j \leq k)\) is called independent if in the set of variables of the group any two variables are different from each other and no variable appears in any \( tm_j \).

We can understand that the assignments in such an independent group can be executed in parallel as they are completely independent from each other.

Definition 4.3.3. [independent group sequence let-in structure]:

An independent group sequence let-in structure is formed as

\[
G_1 \ G_2 \ \cdots \ G_m \ S
\]

Here, each \( G_i \) \((1 \leq i \leq m)\) is an independent assignment group and \( S \) is a substructure of let-in structure.

Theorem 4.3.1. [independent group sequence let-in structure]:

Any let-in structure formed under the constraints 4.3.1 can be certainly grouped to get an independent group sequence let-in structure.

Proof:

We sketch an algorithm to sort out the groups. The basic step of the algorithm is, along the assignment sequence of given let-in structure from left to right, to find the sequential separating points as the boundary of groups. Each of such separating points is located in the left of the first assignment \( v = tm \) whose \( tm \) contain one or more variables which appear as the variables in the assignments
located in the left of the separating point. If there is not such a separating point then all assignments are grouped to one group.
Chapter 5

Refining Time

In this chapter we discuss two steps for time refinement: making time computable and changing time layer.

We introduce some new ideas about time for hardware construction, however to make the ideas easy to understand, in this chapter we basically follow the approach of verification. In the chapter 13.1, after reader has sufficient understanding of our derivation model, we will present and discuss new concepts: how to understand and how to construct time layers in a formal hardware design process.

5.1 Time in Construction

The construction process of a hardware device also constructs the timing of the hardware device. Based on the derivation model we can have a concise and clear concept about time construction.

5.1.1 Constructing Time

Construction raises a concept of time which differs from the one of verification.
Chapter 5.  Refining Time

For verification both implementation and specification are given the task of verification is to prove that the implementation satisfies the specification. For construction only specification is given the task of construction is to construct an implementation which satisfies the specification.

This leads to a difference in the treatment of time, for verification and construction. Verification makes a temporal abstraction from a given low level time of implementation to the given high level time of specification. Construction makes a temporal refinement from a given high level time of specification to the unknown low level time of implementation.

Verification:
[ low_level.time, temporal_abstraction, high_level.time ]

Construction:
[ high_level.time, temporal_refinement, low_level.time ]

Let us emphasize that the low level time of implementation in construction is constructed, not given.

5.1.2 Time Based on Derivation Model

Intuitively, for a hardware device we take the execution periods of its components as the time unit and take the synchronisation of the execution process of two components as time point. Separating a device to its components is relative, namely we can have different coarse level components, therefore we have different time layers.

Based on the derivation model the basic means of construction is to introduce a new middle entry predicate to decompose a derivation model to two new models. We take the derivation model as a foundation to measure time. For derivation model

\[ \forall x. \exists y. E_i x \Rightarrow R \left( i^x, 0^y \right) \land E_o y \]
we take entry predicate to measure \textit{time point}, e.g. by making \( E_i x \) be true the \( x \) is taken as a time point. And we take the specification entity \( R (\bar{i} x, \bar{o} y) \) to measure \textit{time interval} \(< x, y >\) as to implement the \( R \) consumes a certain time. Therefore by construction when we get a specification, mainly through decomposition, we use all the derivation models in the specification to determine the time points of the specification and all these time points establish the time layer of the specification. In the time layer we take the execution time of each derivation model as a time unit, namely we take the time unit as individual. Of course, this is only relative to the time layer of the specification.

\section{Making Time Computable}

We start from the most abstract form of the behavior specification scheme. In this form we only give the specification body \( sb \) by \( S (u, v u + 1) \), namely, we do not consider the internal structure of \( sb \).

\textbf{Specification Scheme 0:} \( \forall u : \text{time. } S (u, u + 1) \)

The specification merely is a timed predicate \( S \) having two successive time point arguments \( u \) and \( u + 1 \). The time is a set of natural number for the time point of specification of a computer at machine instruction level. We use \( u + 1 \) merely to express a successive time point which follows the time point \( u \) and it is better if we write \textquotedblleft succ \( u \textquotedblright\) instead of \textquotedblleft \( u + 1 \textquotedblright\) to emphasize that only successive relation is required by our specification. The specification tells us that a computer should work well at any time. By construction 1 we make the time point of the scheme 1 become computable.

\textbf{Construction 1:}

5.2.1. \( \Psi_1 (\forall u. S (u, u + 1)) \equiv E 0 \land \forall u. E u \Rightarrow S (u, u + 1) \land E (u + 1) \)

In the construction we introduce an entry predicate \( E \) to make an inductively computable time point. This is a loop: starting from 0 and goes along 0, 1, 2, \ldots.
We let the $E$ be a existential variable to leave it to further constructions.

**Specification Scheme 1:** $E \ 0 \land \forall u. \ E \ u \Rightarrow S (u, u + 1) \land E (u + 1)$

Even at the such an abstract level as the specification scheme 1 we have obtained, by construction, a clear, but abstract, structure: that a computer should certainly have a loop structure for its time. Having $E \ 0$ as start point and repeatedly making $E$ be true to get any further time point.

The totalness and correctness of construction are obvious.

### 5.3 Changing Time Layer

It is natural that a certain level of specification is always described in a certain time layer, and that different level specifications are described in the different time layers.

For example, the behavioral specification of a computer at machine instruction level is described in the machine instruction time layer which defines a time unit corresponding to the complete execution of a machine instruction. In contrast, the structural specification of the computer at register transfer level is described in the microinstruction time layer which defines a time unit as the interval representing the execution of a microinstruction.

To construct implementation from specification we must change time layer. Furthermore as time underlies hardware specification, the change of time layer will influence all things which are set up based on time.

In contrast to verification in this stage of construction what we know is only the high time layer, machine instruction time layer; we do not yet know what low time layer is. What we establish in this construction step is the relationship of two time layers. Namely, the low time layer must be obtained by construction. To this
end we introduce the temporal abstraction function [50] to set up the relationship between two, arbitrary, successive time layers.

**Definition 5.3.1. [temporal abstraction function]:**

\[ \vdash \text{abs} \; \text{mark} \; 0 = \varepsilon \; t. \; \text{mark} \; t \land (\forall t'. \; t' < t \Rightarrow \neg \text{mark} \; t') \]

\[ \vdash \text{abs} \; \text{mark} \; (u + 1) = \]

\[ \varepsilon \; t. \]

\[ \text{mark} \; t \land \]

\[ (\text{abs} \; \text{mark} \; u) < t \land \]

\[ (\forall t'. \; (\text{abs} \; \text{mark} \; u) < t' < t \Rightarrow \neg \text{mark} \; t') \]

Here, “abs” is a function of type: \((\text{num} \rightarrow \text{bool}) \rightarrow \text{num} \rightarrow \text{num}\), “num” is the set of natural numbers as the type of discrete time points and “mark” is a function of type: \(\text{num} \rightarrow \text{bool}\) making (abs \(\text{mark}\)) a function of type: \(\text{num} \rightarrow \text{num}\). \(\varepsilon\) is the symbol used in the HOL system [33] for Hilbert’s epsilon operator which may be read as ‘the \(t\) such that ...’. As pp. 271 - 272 [57] if \(P[x]\) is a predicate involving a variable \(x\) of type \(ty\) then \(\varepsilon \; x. \; P[x]\) denotes some value of type \(ty\) such that \(P\) is true of that value. If there is no such value (i.e. \(P[v]\) is false for each value \(v\) of type \(ty\)) then \(\varepsilon \; x. \; P[x]\) denote some fixed but arbitrarily chosen value of type \(ty\).

\(\text{mark}\) is a boolean valued signal in the low time layer. “abs \(\text{mark} \; u\)” denotes the point of low time layer at which \(\text{mark}\) is true for the \(u\)th time. Therefore we can take “abs \(\text{mark}\)” as a function from the time point in high time layer to the time point in low time layer.

What the temporal abstraction function establishes is that a time point in the high layer corresponds to a time point in the low layer and one time unit in the high layer can correspond to a time interval in the low layer, which can include one or more time units in the low time layer.
5.3.1 Construction 2 and Specification Scheme 2

We use the construction 2 to construct the specification scheme 2 from the specification scheme 1.

Construction 2:

5.3.1.1. $\Psi_2 \left( E \; 0 \land \forall u. \; E \; u \Rightarrow S \; (u, u + 1) \land E \; (u + 1) \right) \equiv$

$E' \; 0 \land \Psi_{2a} \left( \forall u. \; E \; u \Rightarrow S \; (u, u + 1) \land E \; (u + 1) \right)$

5.3.1.2. $\Psi_{2a} \left( \forall u. \; E \; u \Rightarrow S \; (u, u + 1) \land E \; (u + 1) \right) \equiv$

$\forall x. \exists y. \; E' \; x \Rightarrow \Psi_{2b} \left( S \; (u, u + 1) \right) \land E' \; y$

5.3.1.3. $\Psi_{2b} \left( S \; (u, u + 1) \right) \equiv S' \; (x, y) \land (\forall t. \; x < t < y \Rightarrow \neg E \; t) \land x < y$

By construction 2 we get the scheme 2 at low time layer from the scheme 1 at high time layer. We know that all signals, functions and predicate in specification are established based on the time. Therefore when we change time layer we must, for each component in specification scheme 2, reestablish its new form in the new time layer. For this in the scheme 2 we use $S'$ to note that it is got by replacing each component $c$ in $S$ of the scheme 1, signal, function and predicate, by $c'$. Here, we write $c$ as a component in the high time layer and $c'$ as a component in the low time layer, which corresponds to $c$. For example, $E'$ which is in the low time layer corresponds to $E$ which is in the high time layer.

To guarantee that such a change leads to a correct construction we need to introduce some assumptions which are based on the temporal abstraction function.
Assumption 5.3.1.1. [correspondence through signal value]:

For each signal \( s \) in the \( S \) of the scheme 1 we assume that we can construct a signal \( s' \) in the low time layer such that

\[
    s \ u = s' \ (\text{abs } E' \ u)
\]

\( \square \)

Note: selecting \( E' \) as the \textit{mark} of temporal abstract function is not fortuitous. We will discuss this in the chapter 13.1.

We can have such an assumption because that we have an observation for the behavior specification:

Observation 5.3.1.1.

In the behavior specification a time point \( u \) always appears as an argument of a signal and vice versa a signal \( sg \) always appears in application to a time point. Namely, they always appear together with the form \( sg \ u \). \( \square \)

However, by the assumption 5.3.1.1 we only get the \( s' \) having definition at some low layer time points that can correspond to the high layer time points. To establish these \( s' \) are total for all time points in low time layer we need the second assumption.

Assumption 5.3.1.2. [signal is total for low time layer]:

We extend all signal \( s' \) of assumption 5.3.1.1 to have definition at all low time points and leave the concrete form of the definition to be established in the further construction process (if the definition is not required by the further construction process then we have a trivial case that we can simply assume that they have an arbitrary value of the same type as those having defined time point). \( \square \)

We need to further consider other high level structure such as function and predicate to make clear their significant in specification scheme 2. For this we have the third assumption:

Assumption 5.3.1.3. [extending higher level structures]:

We extend all higher level structures, we say higher level structure to distinguish it (functions or predicates) from signals (the function of time), in specific-
atation scheme 1 to have definitions in scheme 2 for their arguments which include time points, the extended signals by the assumption 5.3.1.2 and the higher level structures which have been extended by this assumption. Similar to the assumption 5.3.1.2, we leave the concrete form of these definitions to further construction process or simply have a trivial definition if they are irrelevant to construction. □

**Notice 5.3.1.1. [omitting]:**

Only in the construction 2 we use prime "\(\gamma\)" to distinguish the same component but in two different time layers. After the construction 2 we will omit the prime. □

**Specification Scheme 2:**

\[
E \ 0 \land \\
\forall x. \exists y. \ E \ x \implies S \ (x, y) \land \\
(\forall t. \ x < t < y \implies \neg E \ t) \land \\
x < y \land \\
E \ y
\]

The construction 2 and the specification scheme 2 show that

1. The further construction from the specification scheme 2 can be in a new finer time layer;

2. The time point \(x\) and \(y\) in the scheme 2 corresponds to the time point \(u\) and \(u + 1\) in the scheme 1.

3. the \(y\) is existentially quantified such that it is, following arbitrarily given time point \(x\), constructible and we constrain \(x < y\) to make sure that \(y\) follows \(x\) as that \(u + 1\) is the successor of \(u\);

4. the scheme 2 describes such a constraint that a new introduced finer time point \(t\) should correspond to an entry predicate differing from \(E\). This is
done by the predicate:
\[ \forall t. \ x < t < y \Rightarrow \neg E \ t. \]

Convenience 5.3.1.1. [omitting]:

In the further constructions from the scheme 2, for a concise writing form, we will omit the constraint \( x < y \) as it will certainly be satisfied by at least a introduction of two time point \( a \) and \( b \) between \( x \) and \( y \) having relation \( a < b \) or we have \( x < y \) directly. We can also omit the constraint \( (\forall t. \ x < t < y \Rightarrow \neg E \ t) \) as all further introduced new entry predicates are certainly different from \( E \) therefore the constrain is satisfied automatically. \( \square \)

5.3.2 Totalness and Correctness of Construction 2’ and 2

In this section if necessary we will use subscripts \( u \) and \( t \) to distinguish the time point in the high and low layer. The totalness of the construction 2 is obvious.

Theorem 5.3.2.1. [correctness of construction 2]:

For any \( s \in S_1 \) if \( \vdash \Psi_2 (s) \) then \( \vdash s \).

Proof:
The statement of the theorem is equal to that

if
\[
\vdash E' \ 0 \land \\
\forall x. \exists y. \ E' \ x \Rightarrow S' \ (x,y) \land \\
(\forall t. \ x < t < y \Rightarrow \neg E' \ t) \land x < y \land \\
E' \ y
\]
then
\[
\vdash E \ 0 \land \\
\forall u. \ E \ u \Rightarrow S \ (u,u+1) \land \\
E \ (u+1).
\]
Chapter 5. Refining Time

We have two steps to prove:

(1). By assumption 5.3.1.1 we have

\[ \vdash E \ 0_u = E' \ (\text{abs} \ E' \ 0_u). \]

By the definition of abs and \( E' \) we can suppose \( \text{abs} \ E' \ 0_u = 0_t \),
therefore, we have \( \vdash E \ 0_u = E' \ 0_t \). Omitting subscript we have \( \vdash E \ 0 = E' \ 0 \).
As \( E' \ 0 \) has been given we have \( \vdash E \ 0 \).

(2). Given \( E \ u \), here, \( u \) is an arbitrary time point in the high time layer.

By the assumption 5.3.1.1 we have

\[ \vdash E \ u = E' \ (\text{abs} \ E' \ u). \]

Suppose \( t_a = \text{abs} \ E' \ u \) we have

\[ \vdash E' \ t_a. \]

From

\[ \vdash E' \ 0 \land \]
\[ \forall x. \ \exists y. \ E' \ x \Rightarrow S' \ (x, y) \land \]
\[ (\forall t. \ x < t < y \Rightarrow \neg E' \ t) \land x < y \land \]
\[ E' \ y \]

we can get

\[ \vdash \exists y. \ E' \ t_a \Rightarrow S' \ (t_a, y) \land \]
\[ (\forall t. \ t_a < t < y \Rightarrow \neg E' \ t) \land t_a < y \land \]
\[ E' \ y. \]

Then taking \( t_b \) as the selected existential variable \( y \) of \( \exists y \), we have

\[ \vdash E' \ t_a \Rightarrow S' \ (t_a, t_b) \land \]
\[ (\forall t. \ t_a < t < t_b \Rightarrow \neg E' \ t) \land t_a < t_b \land \]
\[ E' \ t_b. \]

As we have \( \vdash E' \ t_a \) we have

\[ \vdash S' \ (t_a, t_b) \land \]
\[(\forall t. t_a < t < t_b \Rightarrow \neg E' t) \land t_a < t_b \land E' t_b.\]

By the definition 5.3.1 we have
\[\vdash \text{abs } E' (u + 1) =\]
\[\varepsilon t_b.\]
\[E' t_b \land\]
\[(\text{abs } E' u) < t_b \land\]
\[(\forall t. \text{abs } E' u) < t < t_b \Rightarrow \neg E' t).\]

As we have supposed \[\vdash t_a = \text{abs } E' u\] we have
\[\vdash \text{abs } E' (u + 1) =\]
\[\varepsilon t_b.\]
\[E' t_b \land\]
\[t_a < t_b \land\]
\[(\forall t. t_a < t < t_b \Rightarrow \neg E' t),\]

therefore, we can get
\[\vdash \text{abs } E' (u + 1) = t_b.\]

By the assumption 5.3.1.1 we have
\[\vdash E (u + 1) = E' (\text{abs } E' (u + 1)).\]

Hence we have
\[\vdash E (u + 1) = E' t_b.\]

We have
\[\vdash E' t_b.\]

So we can get
\[\vdash E (u + 1).\]

By the assumption 5.3.1.1, 2 and 3, and from \[\vdash S' (t_a, t_b)\] we get
\[\vdash S (u, u + 1).\]

This is that from a given \[E u\] of an arbitrary \[u\] we can get \[S (u, u + 1)\] and \[E (u + 1)\].
By above (1) and (2) we get
\[ \vdash E 0 \land \forall u. \ E \ u \Rightarrow S \ (u, u + 1) \land E \ (u + 1). \]

5.3.3 Discussion: Time Refinement and Predicate $E$

In Gordon’s computer [50] a special predicate “ready” is introduced as the mark of the temporal abstraction function. However in this section we have proved that the “ready” is redundant and can be replaced by the entry predicate $E$. This is a interesting example as it illustrates some differences between derivation and verification: derivation is an active construction and proof for an unknown object whereas verification is a passive proof for a known object. So, derivation has more freedom to create a new thing.

It is a compromise that we basically take the viewpoint of verification to deal with time layer change. It is to view the high and low two time layers as two complete different layers and their relation is the corresponding of time points. After reader becomes be familiar with the derivation model we will describe a new concept: low time layer includes high time layer and their relation is construction. The description for time layer change in this chapter is quite wordy. However, by the new concept all these are unnecessary.
Chapter 6

Selecting Global Structure

Construction as formal design includes two aspects: selection and decomposition. By selection we choose a desired one from many possible structures at a certain specification level. By decomposition we divide a structure at the high specification level to the several structures at the low specification level. It is important to note that a selection at the earlier stage of a design has a big influence on the later stages of the design. And a basic fact is that the structure of a specification often implies the structure of the implementation of the specification:

- A let-in structure “let A in S” implies the selection that first is to execute A and then is to execute S.

- A condition structure $P \Rightarrow S_1 \downarrow S_2$ means a necessary situation that first is to execute P then is to execute $S_1$ if the execution of P gets boolean value true or to execute $S_2$ if gets false.

By the selecting global structure we mainly do two things: selecting parallel/sequential execution and selecting common block. However, before this we have a technical step: setting state transition.
6.1 Setting State Transition

The specification scheme 2, like as the behavior specification scheme, has the output state and the input states of computer written separately. To get an individual state transition for each machine instruction we need to distribute output state to each input state across the assignment of let in structure and the condition predicate of condition structure. For example, from

\[
\text{output\_state} = \text{let } A_1 \text{ in} \\
\quad \text{let } A_2 \text{ in} \\
\quad P_1 \Rightarrow \text{input\_state}_1 \downarrow \\
\quad (P_2 \Rightarrow \text{input\_state}_2 \downarrow \text{input\_state}_3)
\]

transform to

\[
\text{let } A_1 \text{ in} \\
\quad \text{let } A_2 \text{ in} \\
\quad P_1 \Rightarrow (\text{output\_state} = \text{input\_state}_1) \downarrow \\
\quad (P_2 \Rightarrow (\text{output\_state} = \text{input\_state}_2) \downarrow (\text{output\_state} = \text{input\_state}_3))
\]

Each \(\text{output\_state} = \text{input\_state}_i\) \((1 \leq i \leq 3)\) is a state transition for a machine instruction.

6.1.1 Construction 3 and Specification 2' and 3

In this step we give more details to the \(S(x, y)\) of the scheme 2 and we rewrite 2 to 2'.

**Specification Scheme 2':**

\[ E \ 0 \land \forall x. \exists y. \ E \ x \Rightarrow \ sb \land \ E \ y \]
**Chapter 6. Selecting Global Structure**

**Specification body** $sb$:

\[
sb ::= os = iss \\
iss ::= is \\
| \text{let } A \text{ in } iss \\
| P \Rightarrow iss \downarrow iss
\]

**Abbreviation**

- $sb$ : specification body
- $iss$ : input state structure
- $os$ : output state
- $is$ : input state
- $A$ : assignment
- $P$ : predicate

For this construction we need not the further detail of the output state $os$, input state $is$, assignment $A$ and predicate $P$.

The construction 3 only change the specification body $sb$ of the scheme $2'$.

**Construction 3:**

6.1.1.1. $\Psi_3 (os = is) \equiv os = is$

6.1.1.2. $\Psi_3 (os = (\text{let } A \text{ in } iss)) \equiv \text{let } A \text{ in } \Psi_3 (os = iss)$

6.1.1.3. $\Psi_3 (os = (P \Rightarrow \alpha_1 \downarrow \alpha_2)) \equiv P \Rightarrow \Psi_3 (os = \alpha_1 \downarrow \Psi_3 (os = \alpha_2))$

In the construction equation 6.1.1 we use variables $\alpha_1$ and $\alpha_2$ to represent two different input state structure $iss$. Generally we have

**Definition 6.1.1.1. [variable in construction]:**

In a construction equation $\Psi$ we can use variable to represent any part of the argument of the $\Psi$. Variable is written by Greek letter. In the match of construction (the definition 2.2.1) a variable can always match the concrete part of an object to be constructed.

Usually, we use variable to represent a complex part of an object and use different variables to distinguish different parts of an object as these parts sometimes can be written as the same in BNF.

Use the construction 3 to the scheme $2'$ we get
Chapter 6. Selecting Global Structure

Specification Scheme 3:

\[ E \ 0 \land \forall x. \ \exists y. \ E \ x \Rightarrow st \ s \land E \ y \]

State transition structure \(st\):  

**Abbreviation:**

\[
\begin{align*}
sts & ::= \ st \\
& | \text{let } A \text{ in } sts \\
& | P \Rightarrow st \downarrow st \\
st & ::= os = is \\
& | \text{st : state transition structure} \\
& | \text{sts : state transition structure} \\
\end{align*}
\]

6.1.2 Totalness and Correctness of Construction 3

**Theorem 6.1.2.1.** [totalness of construction 3]:

\[ \forall s : S_2. \ \Phi_s (s) \in S_3 \]

**Proof:** Use structural induction on the structure of the input state structure \(iss\) of the scheme 2'.

**Lemma 6.1.2.1.** [distributing \(let\ . in\) over \(\Rightarrow\):]

If \(\vdash P \Rightarrow Q\) then \(\vdash let\ A\ in\ P \Rightarrow let\ A\ in\ Q\)

**Proof:** Supposing \(A \equiv (v = tm)\), by the constraint 4.3.1 (see page 22) when we write “let \(A\ in\ P\)" we certainly have the \(v\) as a variable in \(P\). Therefore from \(\vdash P \Rightarrow Q\) we can get \(\vdash P[tm/v] \Rightarrow Q[tm/v]\).

Then we get \(\vdash let\ A\ in\ P \Rightarrow let\ A\ in\ Q\).

Here and elsewhere, we use substitution defined as below.

**Definition 6.1.2.1.** [substitution]:

We write \(\alpha[y/x]\) to denote the result of replacing each \(x\) in \(\alpha\) by \(y\). More precise definition of substitution, for example about avoiding the capture of a free variable, follows [64] Page 26-27.

**Lemma 6.1.2.2.** [distributing condition over \(\Rightarrow\):]

If \(\vdash Q \Rightarrow Q'\) and \(\vdash R \Rightarrow R'\) then \(\vdash (P \Rightarrow Q \downarrow R) \Rightarrow (P \Rightarrow Q' \downarrow R')\)

**Proof:**
From $\vdash Q \Rightarrow Q'$ we have $\vdash (P \Rightarrow Q) \Rightarrow (P \Rightarrow Q')$
and
from $\vdash R \Rightarrow R'$ we have $\vdash (\neg P \Rightarrow R) \Rightarrow (\neg P \Rightarrow R')$.

Therefore we have
$\vdash (P \Rightarrow Q) \land (\neg P \Rightarrow R) \Rightarrow (P \Rightarrow Q') \land (\neg P \Rightarrow R')$.

Finally we get
$\vdash (P \Rightarrow Q \downarrow R) \Rightarrow (P \Rightarrow Q' \downarrow R')$. \hfill $\Box$

**Lemma 6.1.2.3. [Correctness of $\Psi_3$]:**

$\vdash \forall sb : S_{2'}. \Psi_3 (sb) \Rightarrow sb$

**Proof:**

Use structural induction on the iss of sb of the scheme $2'$.

**Base case:** iss $::= is$. It is obvious.

**Step case:** There are two cases: s1 and s2.

**Case s1:** iss $::= \text{let } A \text{ in } iss'$.

By the induction hypothesis we have

$\vdash \Psi_3 (os = iss') \Rightarrow os = iss'$.

Then by lemma 6.1.2.1 we have

$\vdash \text{let } A \text{ in } \Psi_3 (os = iss') \Rightarrow \text{let } A \text{ in } (os = iss')$.

As there is no variable in os (by the behavior specification scheme) we have

$\vdash \text{let } A \text{ in } \Psi_3 (os = iss') \Rightarrow os = \text{let } A \text{ in } iss'$.

By $\Psi_3$ we have

$\Psi_3 (os = (\text{let } A \text{ in } iss')) = \text{let } A \text{ in } \Psi_3 (os = iss')$,

therefore we get

$\vdash \Psi_3 (os = (\text{let } A \text{ in } iss')) \Rightarrow os = \text{let } A \text{ in } iss'$

**Case s2:** iss $::= P \Rightarrow iss' \downarrow iss''$.

Similar with the above case 1 but use the lemma 6.1.2.2. \hfill $\Box$

**Theorem 6.1.2.2. [Correctness of $\Psi_3$]:**

$\vdash \forall s : S_{2'}. \Psi_3 (s) \Rightarrow s$. 
**Proof:** By the fact that only \( sb \) of \( S_2 \) is changed and use the lemma 6.1.2.3. \( \square \)

### 6.2 Selecting Common Sub-State Transition

Using let in sentence we can pull out common structure in a term. This will lead to a corresponding implementation structure. However, for the construction of computer and as a complement, we can also select some common sub-structure of state transitions and the selection will also lead to its corresponding implementation structure. For example, for

\[
\text{State}_\text{Trans}_1 = \text{Sub}_\text{ST}_1 \land [\text{Sub}_\text{ST}_a] \land [\text{Sub}_\text{ST}_b]
\]

\[
\text{State}_\text{Trans}_2 = \text{Sub}_\text{ST}_2 \land [\text{Sub}_\text{ST}_a] \land [\text{Sub}_\text{ST}_b]
\]

\[
\text{State}_\text{Trans}_3 = \text{Sub}_\text{ST}_3 \land [\text{Sub}_\text{ST}_b]
\]

we select sub-state transition \( \text{Sub}_\text{ST}_a \) as the common block of the state transition 1, \( \text{State}_\text{Trans}_1 \), and 2, \( \text{State}_\text{Trans}_2 \), and \( \text{Sub}_\text{ST}_b \) as the common block of the state transition 1, 2 and 3. We bracket them to mark our selection. This will finally lead to the microprogram structure of an implementation formed as below (Here, we write \( sst \) for \( \text{State}_\text{ST} \))

![Diagram](image)

**Definition 6.2.1. [common block]:**

In a state transition \( st \) some of its sub-state transitions, if they are shared by two or more state transitions, can be selected as common block. We use brackets “[” and “]” to mark selected block, for example \([cb]\), to express that \( cb \) is selected as a common block. \( \square \)
Introducing common blocks in a specification can make the corresponding implementation with less hardware resources but, possibly, taking more computation time.

We can extend let.in structure to include common block. However, for the construction of computer we use the concept of common block to express a selection for the execution order, which is as the picture above shown:

**Assumption 6.2.1. [common block in implementation]:**

A common block as sub-state transition is always to be put to the rear part of the execution process of the state transition.

The brackets of common block are purely syntactic and should not influence the logic meaning of specification.

**Assumption 6.2.2. [bracket non-relating to logic meaning]:**

We extend specification scheme to including the syntactic form of common block and suppose that for any common block \([\alpha]\) to have \(\vdash [\alpha] \Leftrightarrow \alpha\).

### 6.2.1 Construction 4 and Specification Scheme 3’ and 4

For the construction 4 we rewrite the scheme 3 to the scheme 3’ by rewriting the state transition \(st\) of the scheme 3 to the conjunction of its sub-state transitions. The rewriting is according to the behavior specification scheme.

**Specification Scheme 3’:**

\[ E \ 0 \land \forall x. \exists y. \ E \ x \Rightarrow \ sts \land E \ y \]

**State transition structure \(sts\):**

Abbreviation:

\[
sts ::= \ st \\
\ | \ let \ A \ in \ sts \\
\ | \ P \Rightarrow \ sts \downarrow \ sts \\
st ::= \ sst \\
\ | \ sst \land \ st
\]
Chapter 6. Selecting Global Structure

\[ sst ::= sos_i = sis_i \]

\[ sst \text{: sub-state transition} \]

\[ sos_i \text{: the } i^{th} \text{ sub output state} \]

\[ sis_i \text{: the } i^{th} \text{ sub input state} \]

\[ (1 \leq i \leq n) \]

Construction 4 only selects whether or not a sub-state transition \( sst \) as a common block.

**Construction 4:**

6.2.1.1. \( \Psi_4 (st) \equiv \mathcal{A}_4 (\Psi_{4a} (st)) \)

6.2.1.2. \( \Psi_{4a} (sst) \equiv \Psi_{4b} (sst) \)

6.2.1.3. \( \Psi_{4a} (sst \land st) \equiv \Psi_{4b} (sst) \land \Psi_{4a} (st) \)

6.2.1.4. \( \Psi_{4b} (sst) \equiv sst \lor [sst] \)

In the construction equation 6.2.1.4 we introduce symbol “\( \lor \)” which means “or”. The construction equation says that by construction \( \Psi_{4b} \) \( sst \) can be transformed to \( sst \) or \([sst]\). Generally, we have

**Definition 6.2.1.1. [nondeterministic construction]:**

An object \( x \) can be constructed by \( \Psi \) to get \( y_1 \) or \( y_2 \) or \( \ldots \) \( y_n \). We call this nondeterministic construction and write:

\[ \Psi (x) \equiv y_1 \lor y_2 \lor \cdots \lor y_n \]

In the construction 4 we introduce an abstract algorithm \( \mathcal{A}_4 \) to sort the common blocks in a state transition. Generally we have

**Definition 6.2.1.2. [abstract algorithm]:**

Abstract algorithm is a construction which is borrowed from from other area of computer science, therefore we only require its specification. For an object \( x \), through an abstract algorithm \( \mathcal{A} \), is constructed to get \( y \) we write

\[ \mathcal{A} (x) \equiv y. \]

We introduce abstract algorithm as a means to interface our derivation to other methods, especially to the optimization algorithm in the high level synthesis field.
Abstract Algorithm 6.2.1.1. [$A_4$: sorting common blocks]:

$A_4$ manipulates $\Psi_{4a}(st)$.

1. All $sst$ which are not selected as common blocks are put together to get sub-state transition set $ssts$ as the private part $pp$ of the state transition.

2. All $[sst]$ that are shared by the same state transitions will be put together to get a common sub-state transition set $[ssts]$.

3. Then, all $[ssts]$ are sorted as below to get the common part $cp$ of scheme 5:

   - All the $[ssts]$ in a state transition are sorted from left to right in the order of increasing number of shared state transitions $st$. For example, suppose a state transition $st$ includes three common blocks $cb_1$, $cb_2$ and $cb_3$, where $cb_1$ is shared by 5 state transitions, $cb_2$ is shared by 2 state transitions, and $cb_3$ is shared by 3 state transitions. Then we have $A_5 (cb_1 \land cb_2 \land cb_3) = cb_2 \land cb_3 \land cb_1$. As the private part $pp$ of a state transition is always “shared” by itself only the $pp$ always is put the most left. For the example above we get $st = pp \land cb_2 \land cb_3 \land cb_1$.

4. For each state transition $st_i$ check the sequences of sub-state transition set $ssts$ in the $st_i$ from left to right such that for arbitrary two state transition, for example $st_1$ and $st_2$, if they have the same common block then all other $ssts$ on the right (if any) of the common block must be the same and have the same order.

   - Where this is not satisfied change some common block back to the private part of $st$ to satisfy the requirement above.

Such checking and changing is to guarantee a tree form structure. For example, the case

$$st_1 = pp_1 \land [ssts_a] \land [ssts_b]$$
\[ st_2 = pp_2 \land [sst_{s_a}] \]
\[ st_3 = pp_3 \land [sst_{s_b}] \]

cannot lead to a tree form implementation structure and we need to change to either

\[ st_1 = (pp_1 \land sst_{s_a}) \land [sst_{s_b}] \]
\[ st_2 = (pp_2 \land sst_{s_a}) \]
\[ st_3 = pp_3 \land [sst_{s_b}] \]

in which case \( sst_{s_a} \) is moved back to the private parts, or

\[ st_1 = (pp_1 \land sst_{s_b}) \land [sst_{s_a}] \]
\[ st_2 = pp_2 \land [sst_{s_a}] \]
\[ st_3 = (pp_3 \land sst_{s_b}) \]

in which case \( sst_{s_b} \) is moved back to the private parts. \( \Box \)

After the construction 4 we get the specification scheme 4.

**Specification Scheme 4:**

\[ E \ 0 \land \forall x. \exists y. E \ x \Rightarrow sst \land E \ y \]

**State transition structure** \( sst \):  **Abbreviation:**

\[ sst ::= \]
\[ | \text{let } A \text{ in } sst \]
\[ | P \Rightarrow sst \downarrow sst \]
\[ st ::= pp \land cp \]
\[ pp ::= T \]
\[ | sst \]
\[ cp ::= T \]
\[ | [sst] \land cp \]
\[ sst ::= sst \]
\[ | sst \land sst \]

\( T \) : boolean value true
\( sst \) : sub-state transition
In the scheme 4 a state transition $st$ includes two parts: the private part $pp$ and the common part $cp$. However, each part can be empty which is expressed by the boolean constant $T$.

The totalness and correctness of the construction 4 is obvious.

### 6.3 Selecting Parallel Term Transition

We can put two function units ADD and SUB into the same time interval to execute if they are implemented by two hardware devices. We call this as the parallel execution. The parallelism is at function unit level which is traditionally called as the register transfer level.

In the construction process we find that we can introduce parallelism at a higher level which we call the term transition level in the earlier stage of design.

**Definition 6.3.1. [term transition]:**

A term transition is formed as $sg \ y = tm$. Here, $sg$ is a signal and $tm$ is a term which is defined in the behavior specification scheme, however here, is extended to include predicate $p$ and predicate connective $\neg$, $\land$, $\lor$, $\rightarrow$, and $\leftrightarrow$ as function $f$. ☐

For the construction of computer there are two cases where we can select whether or not to take the parallel term transition. For example, for a `let` in structure

let $v_1 = tm_1$ in let $v_2 = tm_2$ in $S$,

suppose that it satisfies the constraint 4.3.1 (page 22) and their two assignments are independent from each other. We can construct two term transitions for the two assignments

$$s_1 \ y = tm_1 \quad (1)$$

$$s_2 \ y = tm_2 \quad (2)$$
and replace $S$ by $S \ [s_1 \ y/v_1, \ s_2 \ y/v_2]$. 

Then we have two selections:

(a). parallel execution: in the same time interval to execute both (1) and (2).

(b). sequential execution: taking two time intervals: first to execute (1) and then to execute (2)

For the case (a) we write:

\[ \{_1 \text{let } v_1 = tm_1 \text{ in let } v_2 = tm_2 \text{ in } \{_2 \ S\}\} \]  \hfill (11)

and for the case (b) we write:

\[ \{_3 \text{let } v_1 = tm_1 \text{ in } \{_4 \text{let } v_2 = tm_2 \text{ in } \{_5 \ S\}\}\} \]  \hfill (12)

We have

**Assumption 6.3.1. [parallel selection for let in structure]:**

The assignments between two successive braces "{" are selected for parallel execution. \hfill \Box

For example, in (11) both $v_1 = tm_1$ and $v_2 = tm_2$ are put into two successive braces $\{_1$ and $\{_2$. And in (12) only one $v_1 = tm_1$ is put between $\{_3$ and $\{_4$ and another $v_2 = tm_2$ is put between $\{_4$ and $\{_5$. Note, here we put subscript to braces "{" for conveniently distinguishing them.

However we have a constraint for the selection.

**Constraint 6.3.1. [parallel selection for let in structure]:**

Only the assignments in the same group of independent assignment (the definition 4.3.2, page 23) can be selected for parallel execution. \hfill \Box

We have the constraint as only under it a parallel execution is feasible or, at least, better parallel.

Similarly, for a condition structure

$P_1 \Rightarrow \ S_1 \downarrow (P_2 \Rightarrow \ S_2 \downarrow \ S_3)$
we introduce two term transitions for the executions of two predicates $P_1$ and $P_2$

\[ b_1, y = P_1 \]  
\[ b_2, y = P_2 \]  

Here $b_1$ is a boolean valued signal and (3) expresses a process to calculate predicate $P_1$ such that at $y$, $b_1$ has the same boolean value as of the $P_1$.

Similarly, we have selection for that parallel and sequential executions for (3) and (4). After the execution of (3) and (4) the execution of the condition structure becomes to execute

\[ b_1, x \Rightarrow S_1 \downarrow (b_2, x \Rightarrow S_2 \downarrow S_3) \]

For the parallel one we write the condition structure

\[ \{1P_1 \Rightarrow \{2S_1\} \downarrow (P_2 \Rightarrow \{3S_2\} \downarrow \{4S_3\})\} \]  

(p1)

and for the sequential one

\[ \{5P_1 \Rightarrow \{6S_1\} \downarrow \{7P_2 \Rightarrow \{8S_2\} \downarrow \{9S_3\}\}\} \]  

(p2)

We also have

**Assumption 6.3.2. [parallel selection for condition structure]:**

The predicates between the same brace “{“ and a group of successive braces “{“ are selected for parallel execution. □

For example, in (p1) $P_1$ and $P_2$ are put into a pair of $\{1$ and a group of $\{2$ and $\{3$. And in (p2) $P_1$ is put into a pair of $\{5$ and $\{6$ and $P_2$ is put into another pair of $\{7$ and $\{8$ separately.

The selection for parallelism in the earlier stage of design has a bigger influence on the structure of implementation than the one at function unit level. This is because the term transition is implemented by a sequence of function unit and the selection at function unit level is one in the inside of a term transition. Therefore, they are high and low two level selections. The high one influences the the whole structure of microprogram of computer and the low one only influences the local
structure of microprogram (linear part). In the chapter 8 we will see more details about the low level.

We need a selection for two possibilities of parallel one or sequential one. This means that it is not case that a parallel is always better than the sequential one. For example for
\[
\{1P_1 \Rightarrow \{2S_1\} \downarrow (P_2 \Rightarrow \{3S_2\} \downarrow \{4S_3\})
\]
\[
\{5P_1 \Rightarrow \{6S_1\} \downarrow \{7P_2 \Rightarrow \{8S_2\} \downarrow \{9S_3\}\}\}
\]
\[(c1)\]
\[(d1)\]
Let us suppose that the execution of (3) above \((b_1, y = P_1)\) takes 2 time units, the one of (4) above \((b_2, y = P_2)\) takes 100 time unit and the execution of \(S_1\) takes 3 time units and we have an application where \(P_1\) is more often calculated to be true. Then in the case the sequential selection \((d1)\) will lead to a faster execution.

The each sub-state transition \(sst\) is naturally a term transition. For the execution of the state transition \(st\), traditionally, all the \(sst\) in the private part \(pp\) of a \(st\) are selected for parallel execution and so are all the \(sst\) in a common block of a \(st\). However, it is also natural the \(pp\) and common blocks, if all they are taken as an execution unit, are selected for sequential execution because the common block will be shared by two or more state transitions. For example if a \(st = pp \land cb_1 \land cb_2\) then the execution of the \(st\) takes three time intervals: the first is for \(pp\), the second is for \(cb_1\) and the third is for \(cb_2\).

The brace used for the selections for parallel or sequential executions are purely syntactic and should not influence the logic meaning of specification. We have similar assumption like as one for common block.

**Assumption 6.3.3.** [brace non-relating to logic meaning]:

We extend specification scheme to including braces for expressing selection for parallel or sequential executions and suppose that for any part formed as \(\{\alpha\}\) to have \(\vdash \{\alpha\} \Leftrightarrow \alpha\).
Chapter 6. Selecting Global Structure

6.3.1 Construction 5 and Specification Scheme 4' and 5

For the construction 5 we only need a shorted form of the specification scheme 4 as the state transition \( st \) of the scheme 4 can be taken as a terminal.

**Specification Scheme 4':**

\[ E 0 \land \forall x. \exists y. E x \Rightarrow st \times E y \]

**State transition structure \( st \):**

**Abbreviation:**

\[
\begin{align*}
st & ::= \quad st \\
 & \quad | \text{let } A \text{ in } st \\
 & \quad | P \Rightarrow st \downarrow st
\end{align*}
\]

For the construction 5 and the scheme 5 we take a convenience.

**Convenience 6.3.1.1. [omitting the most outside braces]:**

The most outside a pair braces "{" and "}" of the state transition structure \( st \) are omitted.

Only the \( st \) of the scheme 4' is dealt with by the construction 5.

**Construction 5:**

6.3.1.1. \( \Psi_5 (st) \equiv st \)

6.3.1.2. \( \Psi_5 (\text{let } A \text{ in } st) \equiv \text{let } A \text{ in } \Psi_l(st) \)

6.3.1.3. \( \Psi_5 (P \Rightarrow \alpha_1 \downarrow \alpha_2) \equiv P \Rightarrow \Psi_p(\alpha_1) \downarrow \Psi_p(\alpha_2) \)

6.3.1.4. \( \Psi_l(st) \equiv \{st\} \)

6.3.1.5. \( \Psi_l(\text{let } A \text{ in } st) \equiv \quad \text{let } A \text{ in } \Psi_l(st) \\
 & \quad | \{\text{let } A \text{ in } \Psi_l(st)\} \)

6.3.1.6. \( \Psi_l(P \Rightarrow \beta_1 \downarrow \beta_2) \equiv \{P \Rightarrow \Psi_p(\beta_1) \downarrow \Psi_p(\beta_2)\} \)

6.3.1.7. \( \Psi_p(st) \equiv \{st\} \)

6.3.1.8. \( \Psi_p(\text{let } A \text{ in } st) \equiv \{\text{let } A \text{ in } \Psi_l(st)\} \)

6.3.1.9. \( \Psi_p(P \Rightarrow \gamma_1 \downarrow \gamma_2) \equiv \quad P \Rightarrow \Psi_p(\gamma_1) \downarrow \Psi_p(\gamma_2) \\
 & \quad | \{P \Rightarrow \Psi_p(\gamma_1) \downarrow \Psi_p(\gamma_2)\} \)
In the construction 5 the equations 6.3.1.1, 2 and 3 deal with the \textit{sts}. The equations 6.3.1.4, 5 and 6 deal with \textit{let.in} structure and through the equation 6.3.1.5 we have selection that whether or not to take a parallel execution. Similarly the equation 6.3.1.7, 8 and 9 deal with condition structure and the equation 6.3.1 is for parallel selection.

\textbf{Specification Scheme 5:}

\[ E 0 \land \forall x. \exists y. E x \Rightarrow sts \land E y \]

\begin{itemize}
  \item \textbf{State transition structure} \textit{sts}: \textbf{Abbreviation:}
  \begin{verbatim}
  ststs := st
  \mid let A in ls
  \mid P \Rightarrow cs \downarrow cs
  
  ls := let A in ls
  \mid \{ ststs \}
  
  cs := P \Rightarrow cs \downarrow cs
  \mid \{ ststs \}
  \end{verbatim}
\end{itemize}

The totalness and correctness of the construction 5 are obvious.
Chapter 7

Decomposing to Term Transition

Decomposition, by which a complex object is divided into its simple components, is the main means of construction. In this chapter we discuss how to decompose a specification at the scheme 5 level to a specification whose basic component is the term transition (the definition 6.3.1 in page 47).

We take term transition as a stage of the construction of computer because:

- The term transition is a middle level structure of specification of computer. A sub-state transition by itself is a term transition. The assignment $v = tm$ of a let.in structure is implemented by a term transition $s.y = tm$. The predicate $P$ of a condition structure is implemented by a term transition $b.y = P$. The term transition nearly is the unique component of a specification at the term transition level, except the branch structure. These will be clearer in the specification scheme 6.

- When taking term transition as a basic component of the specification of computer, namely at this level we do not further consider the inside structure of term transition, we can have some arbitrary selections for the parallel and sequential executions to organize all the term transitions in a specification of computer. This is as the discussions of preceding chapter.
The organization of term transitions will form a high level structure of a computer. The structure will directly correspond to an abstract microprogram of the implementation of computer, which is formed as a closed tree:

- Its each arc corresponds to a term transition or a group of parallel term transitions and the arc is implemented through the control of a corresponding leaner microprogram.
- Its each node entries an arc and corresponds to the entry address of a linear microprogram which the arc corresponds to.
- Its cross node that an arc is connected to two or more arcs corresponds to a branch structure: under different conditions an arc is connected to different arcs.
- The tree is closed as its every leaf arc will certainly go back to the root of the tree. The root is the global entry point of the whole microprogram of computer.

Let us give two examples to help reader to understand the description above and what a decomposition is. We only describe what two pictures before and after the decomposition are, and leave the decomposition process to the construction where we give a generic and exact description for the construction process.

**Example 7.1**

The example is a piece of specification picked from the specification of Gordon’s computer.

Specification 1:

(\text{let } v = \text{Fetch}_{13} \ (memory \ x) \ (pc \ x) \text{ in}
\ (\text{Val}_3 \ (\text{Opcode} \ v) = 0) \Rightarrow \text{state\_transition}_4 \ | \ \text{state\_transition}_5
\ ) \ [E_i, \ E_o]

Here,
Chapter 7. Decomposing to Term Transition

\[\text{state-transition}_4 \equiv (pc \ y = pc \ x \ \land \ \text{idle} \ y = T)\]
\[\text{state-transition}_5 \equiv (pc \ y = \text{Cut}_{13} \ v \ \land \ \text{idle} \ y = F)\]

\(pc\) (program counter) and \(idle\) are the state signals of the computer.
Fetch_{13}, Opcode, Val_3 and Cut_{13} are constant functions.
\(v\) is an assignment variable. \(T\) and \(F\) are boolean constants.

By decomposition we get

Specification 2:

\[(v \ y = \text{Fetch}_{13} (\text{memory} \ x) (pc \ x) \ \land \ pc \ y = pc \ x \ \land \ \text{idle} \ y = \text{idle} \ x) \ [E_i, \ E_1]\] (1)
\[\land\]
\[(\forall x. \ E_1 \ x \Rightarrow (\text{Val}_3 (\text{Opcode} (v \ x)) = 0 \Rightarrow E_2 \ x \downarrow E_3 \ x))\] (2)
\[\land\]
\[\text{state-transition}_4 \ [E_2, \ E_o]\] (3)
\[\land\]
\[\text{state-transition}_{5,1} \ [E_3, \ E_o]\] (4)

Here,

\[\text{state-transition}_{5,1} \equiv (pc \ y = \text{Cut}_{13} (v \ x) \ \land \ \text{idle} \ y = F)\]

Thus specification 1 is decomposed into the four parts: (1), (2), (3) and (4). The decomposition follows the natural structure of the specification 1: (1) corresponds to its let_in sentence; (2) corresponds to its conditional sentence; (3) and (4) correspond to its two state transitions. Each part becomes an independent sub-specification which has the uniform form of the derivation model: (1), (3) and (4) follow the derivation model 3.1.1 and (2) follows the branch structure model 7.1 as below. If we use the definitions of models to unfold the specifications 1 and 2 then it is obvious that specification 2 satisfies specification 1, namely,

\[\vdash \text{Specification 2} \Rightarrow \text{Specification 1}\]

In the (1) above we introduce \(pc \ y = pc \ x\) and \(idle \ y = idle \ x\). They are used to keep both \(pc\) and \(idle\) unchanged in the time intervals of (1).
Branch Structure Model 7.1:

Syntax: \[ E_i, E_x \]  
\[ E_x ::= E_o \mid b \ (E_x, E_x) \]

Semantics: \[ S_m ([E_i, E_x]) = \forall x. \ E_i \ x \Rightarrow S_m (E_x) \]
\[ S_m (E_o) = E_o \ x \]
\[ S_m (b (E_{x1}, E_{x2})) = (b \ x \Rightarrow S_m (E_{x1}) \downarrow S_m (E_{x2})) \]

Here, \( b \) is a boolean valued signal;
\( E_i \) is an entry predicate;
\( E_o \) is a leaving predicate.
\( E_x \) is called leaving predicate structure.

For instance, we write

\[ [E_2, b_1 (E_3, b_2 (E_4, E_5))] \]

to express

\[ \forall x. \ E_2 \ x \Rightarrow (b_1 \ x \Rightarrow E_3 \ x \downarrow (b_2 \ x \Rightarrow E_4 \ x \downarrow E_5 \ x)) \]

For easier to write a branch structure we can use the convenient form that we can write \( b \ x \) to replace \( b \), even write a formula to replace \( b \) if the formula is considered to be executable at a given implementation level.

Example 7.2
The example is a simple and abstract but somewhat complete one to give us a general picture about what is after decomposition.

Specification 1:

\[ \forall x. \ \exists y. \ E \ x \Rightarrow \text{sts} \land E \ y \]
\[ \text{sts} \equiv \{ \ \text{let } v_1 = tm_1 \ \text{in} \]
\[ \text{let } v_2 = tm_2 \ \text{in} \]
\[ \{ \ P_1 \Rightarrow \{ pp_1 \land [cb] \} \downarrow \]
\[ \{ \ P_2 \Rightarrow \{ pp_2 \land [cb] \} \downarrow \{ pp_3 \} \} \]
In the specification the two assignments \( v_1 = tm_1 \) and \( v_2 = tm_2 \) of the let.in structure has been selected for parallel execution and so do the two predicates \( P_1 \) and \( P_2 \) of the condition structure. There are three state transitions: \( pp_1 \land [cb] \), \( pp_2 \land [cb] \) and \( pp_3 \). \([cb]\) marks that the \( cb \) is selected as a common block.

Using the derivation model we rewrite the specification to

**Specification 1':**

\[
\{ \text{let } v_1 = tm_1 \text{ in} \\
\text{let } v_2 = tm_2 \text{ in} \\
\{ P_1 \Rightarrow \{ pp_1 \land [cb] \} \downarrow \\
\{ P_2 \Rightarrow \{ pp_2 \land [cb] \} \downarrow \{ pp_3 \} \} \}
\] \[
\} \ [E, E]
\]

By decomposition from the specification 1' we get:

**Specification 2:**

\[
(s_1 \ y = tm_1 \land s_2 \ y = tm_2 \land K_1) \ [E, E_1] \quad (1)
\]
\[
\land
\]
\[
(b_1 \ y = P_1 \land b_2 \ y = P_2 \land K_2) \ [E_1 E_2] \quad (2)
\]
\[
\land
\]
\[
[E_2, b_1 \ (E_3, b_2 \ (E_4, E_5))] \quad (3)
\]
\[
\land
\]
\[
(pp_1 [s_1 \ x/v_1, s_2 \ x/v_2] \land K_3) \ [E_3, E_6] \quad (4)
\]
\[
\land
\]
\[
(pp_2 [s_1 \ x/v_1, s_2 \ x/v_2] \land K_4) \ [E_4, E_6] \quad (5)
\]
\[
\land
\]
\[
(cb [s_1 \ x/v_1, s_2 \ x/v_2] \land K_5) \ [E_6, E] \quad (6)
\]
\[
\land
\]
\[(pp_3 [s_1 x/v_1, s_2 x/v_2]) [E_5, E]\] (7)

Here,
\[
K_1 = P_1^i \land P_2^i \land pp_1^i \land pp_2^i \land pp_3^i \land cb^i \\
K_2 = pp_1^i \land pp_2^i \land pp_3^i \land cb^i \\
K_3 = K_4 = cb^i \\
K_5 = pp_1^o \land pp_2^o
\]

Here and elsewhere, we use the abbreviation \(S^i\) and \(S^o\):

**Definition 7.1.** [abbreviation for input and output signal protection]:
\[
S^i \equiv (r_1 y = r_1 x) \land \cdots \land (r_n y = r_n x)
\]
Here, \(r_i\) \((1 \leq i \leq n)\) are all the input signals in \(S\).

Similarly,
\[
S^o \equiv (s_1 y = s_1 x) \land \cdots \land (s_m y = s_m x)
\]
Here, \(s_i\) \((1 \leq i \leq m)\) are all the output signals in \(S\). We also take the boolean constant \(T\) as a special \(S^i\) or \(S^o\). \(\square\)

And following the branch model 7.1
\[
[E_2, \ b_1 (E_3, \ b_2 (E_4, \ E_5))]
\]
\[
= \forall x. \ E_2 x \Rightarrow (b_1 x \Rightarrow E_3 x \downarrow (b_2 x \Rightarrow E_4 x \downarrow E_5 x))
\]

In the specification 2

1. (1) is a process to calculate \(tm_1\) and \(tm_2\) parallel and then assign the results to the signals \(s_1\) and \(s_2\) respectively at some time point \(y\).
   Meanwhile in the time interval of calculation all the input signals in the \(P_1, P_2, pp_1, pp_2, pp_3\) and \(cb\) are kept unchanged.

2. (2) is a process to calculate \(P_1\) and \(P_2\) in parallel manner and then assign the boolean values to the signals \(b_1\) and \(b_2\) respectively at some time point \(y\).
   Meanwhile in the time interval of calculation all the input signals in the \(pp_1, pp_2, pp_3\) and \(cb\) are kept unchanged.
3. (3) is a branch structure which starts from $E_2$ at a time point $x$ if $b_1$ at $x$ is true then makes $E_3$ at $x$ be true, otherwise if $b_2$ at $x$ is true then makes $E_4$ at $x$ be true, otherwise makes $E_5$ at $x$ be true.

4. (4) is a process to calculate $pp_1\ [s_1 \ x/v_1, \ s_2 \ x/v_2]$ (for short we write it $pp'_1$ in the figure 7.1.) The $v_1$ in $pp_1$ has been replaced by $s_1 \ x$ through which the result of the calculation of $tm_1$ is passed to, and similarly for the $v_2$. Meanwhile in the time interval of the calculation all the input signals in the $cb$ are kept unchanged.

5. (5) is similar to (4).

6. (6) is to calculate $cb\ [s_1 \ x/v_1, \ s_2 \ x/v_2]$.

Meanwhile in the time interval of calculation all the output signals in the $pp_1$ and $pp_2$ are kept unchanged.

7. (7) is a process to calculate $pp_3\ [s_1 \ x/v_1, \ s_2 \ x/v_2]$.

The above is as the figure 7.1.

From the two examples we can imagine that the decomposition is a process to introduce new entry predicates, for example the $E_i$ ($1 \leq i \leq 6$) in the specification 2 of the example 2, to divide a specification to a set of term transitions along the natural structure of the specification: let in structure, condition structure and state transition.
The flowchart shows us a high level structure of the computer whose specification is the specification 1 of the example 2. Each frame-box corresponds to a group of parallel term transitions.

### 7.1 Construction 6, Abstract Algorithm 6 and Specification Scheme $5'$ and 6

The construction will deal with the decomposition to term transition level. We take the description of state transition $st$ of the specification scheme 4 to the specification scheme 5 to get the specification scheme $5'$ which shows not only the
parallel selections for let.in structure and condition structure but also the selection for the common block of state transition.

**Specification Scheme 5'**:  
\[ E \ 0 \land (sts) [E, E] \]

**State transition structure** \( sts \): **Abbreviation:**  
\[
sts ::= st  
| let A in ls  
| P ⇒ cs ↓ cs  
\]

\[
l s ::= let A in ls  
| \{sts\}  
\]

\[
c s ::= P ⇒ cs ↓ cs  
| \{sts\}  
\]

\[
st ::= pp \land cp  
| ssts  
\]

\[
pp ::= T  
| ssts  
\]

\[
cp ::= T  
| [ssts] \land cp  
\]

The construction 6 manipulates the scheme 5' to produce the specification scheme 6. It deals with \((sts) [E, E]\), however, in the construction \(Ψ_6\), instead of \([E, E]\), we write \([E_i, E_o]\) because the \(E_i\) and \(E_o\) can be used to represent any entry predicate. This is also because that the \(Ψ_6\) can be recursively used to any level sub-structure of a \(sts\).

**Construction 6:**

7.1.1.  \[ Ψ_6 (st [E_i, E_o]) ≡ Ψ_6s (((T) \land st) [E_i, E_o]) \]

7.1.2.  \[ Ψ_6 ((let A in \alpha) [E_i, E_o]) \equiv (Ψ_{61} (let A in \alpha)) [E_i, E_m] \land \\
Ψ_6 ((Ψ_{62} (let A in \alpha, [])) [E_m, E_o]) \]
7.1.3.  \( \Psi_6 \left( (P \Rightarrow \alpha_1 \downarrow \alpha_2) \left[ E_i, \ E_o \right] \right) \equiv \left( \Psi_{6p1} \left( P \Rightarrow \alpha_1 \downarrow \alpha_2 \right) \right) \left[ E_i, \ E_m \right] \wedge \left[ E_m, \ \Psi_{6p2} \left( P \Rightarrow \alpha_1 \downarrow \alpha_2 \right) \right] \wedge \Psi_{6p3} \left( P \Rightarrow \alpha_1 \downarrow \alpha_2, \ E_o \right) \\

7.1.4.  \Psi_{6s} \left( \left( (K) \wedge \alpha \wedge [\beta] \right) \left[ E_i, \ E_o \right] \right) \equiv \Psi_{6s} \left( \left( (\beta^d \wedge K) \wedge \alpha \right) \left[ E_i, \ E_m \right] \wedge \left( (\alpha^o \wedge K) \wedge \beta \right) \left[ E_m, \ E_o \right] \right) \\

Here, \( E_m = E_{(\beta, E_o)} \). 

7.1.5.  \Psi_{6s} \left( \left( (K) \wedge \gamma \right) \left[ E_i, \ E_o \right] \right) \equiv \left( K \wedge \gamma' \right) \left[ E_i, \ E_o \right] \\
Here, if \( \gamma = [\alpha] \) then \( \gamma' = \alpha \) otherwise \( \gamma' = \gamma \) 

7.1.6.  \Psi_{6l1} \left( \text{let } v = tm \text{ in } \alpha \right) \equiv s_v \ y = tm \wedge \Psi_{6l1} \left( \alpha \right) 

7.1.7.  \Psi_{6l1} \left( \{ \beta \} \right) \equiv \beta^d 

7.1.8.  \Psi_{6l2} \left( \text{let } v = tm \text{ in } \alpha, \ L \right) \equiv \Psi_{6l2} \left( \alpha, \ [s_v \ x/v] :: L \right) 

7.1.9.  \Psi_{6l2} \left( \{ \beta \}, \ L \right) \equiv \beta \ L 

7.1.10.  \Psi_{6p1} \left( P \Rightarrow \alpha_1 \downarrow \alpha_2 \right) \equiv \beta_p \ y = P \wedge \Psi_{6p1} \left( \alpha_1 \right) \wedge \Psi_{6p1} \left( \alpha_2 \right) 

7.1.11.  \Psi_{6p1} \left( \{ \beta \} \right) \equiv \beta^d 

7.1.12.  \Psi_{6p2} \left( P \Rightarrow \alpha_1 \downarrow \alpha_2 \right) \equiv \beta_p \left( \Psi_{6p2} \left( \alpha_1 \right), \ \Psi_{6p2} \left( \alpha_2 \right) \right) 

7.1.13.  \Psi_{6p2} \left( \{ \beta \} \right) \equiv E_{\beta} 

7.1.14.  \Psi_{6p3} \left( P \Rightarrow \alpha_1 \downarrow \alpha_2, \ E_o \right) \equiv \Psi_{6p3} \left( \alpha_1, \ E_o \right) \wedge \Psi_{6p3} \left( \alpha_2, \ E_o \right) 

7.1.15.  \Psi_{6p3} \left( \{ \beta \}, \ E_o \right) \equiv \Psi_6 \left( \{ \beta \} \left[ E_{\beta}, \ E_o \right] \right) 

Let us explain the construction 6.

The equation 7.1.1, 2 and 3 distinguish the three cases: state transition “st”, let in structure “let A in α” and condition structure “P ⇒ α1 ↓ α2”, and pass them to the constructions \( \Psi_{6s} \), \( \Psi_{6li} \) (1 ≤ i ≤ 2) and \( \Psi_{6pi} \) (1 ≤ i ≤ 3) respectively to deal with further.

**Construction of State Transition:**

The construction \( \Psi_{6s} \) deals with state transition which has form:

\[
pp \wedge [cb_1] \wedge [cb_2] \wedge \cdots \wedge [cb_n]
\] (11)
Here, $pp$ is the private part and others are the sorted common blocks.

However, the $(11)$ will be treated along direction from right to left, namely along the reverse of the sequence $(11)$:

$[cb_n], [cb_{n-1}], \ldots, [cb_1], pp$

(This is for correctly introducing new entry predicate to get a tree-shape implementation (see page 42))

Therefore in the construction equation 7.1.4 when we write

$\alpha \land [\beta]$

where,

$[\beta]$ corresponds to the $[cb_n]$ of $(11)$,

$\alpha$ corresponds to the $pp \land [cb_1] \land [cb_2] \land \cdots \land [cb_{n-1}]$ of $(11)$.

(The match follows logic programming Prolog: we use the bracket “[“ and “]” to identify the last common block)

In the $\Psi_6$ we also introduce variable $K$ to accommodate all signal protections (the definition 7.1 in page 58). To distinguish the $K$ from the state transition $\alpha \land [\beta]$ of the construction equation 7.1.4 and from the sub-state transition $\gamma$ of the construction equation 7.1.5 we bracketed it as $(K)$. The $K$ is initiated by the boolean constant $T$ in the construction equation 7.1.1

In the construction equation 7.1.4 we introduce a new entry predicate $E_m$ whose detailed form is $E_{(\beta, E_o)}$ where the subscript $(\beta, E_o)$ means that $E_{(\beta, E_o)}$ is determined uniquely following $\beta$ and $E_o$. Through the $E_m$ we break down $\alpha \land [\beta]$ to two parts $\alpha$ and $[\beta]$. This is that the original process (derivation model)

$((K) \land \alpha \land \beta) [E_i, E_o]$

is divided to two new processes:

$((\beta' \land K) \land \alpha) [E_i, E_m]$  \hspace{1cm} (12)

$((\alpha' \land K) \land \beta) [E_m, E_o]$  \hspace{1cm} (13)
In the (l3) by $\alpha^0$ all output signals in $\alpha$ are protected and in the (l2) by $\beta^0$ all input signals in $\beta$ are protected. All these protections are added to $K$.

The construction equation 7.1.5 deals with the most left sub-state transition $\gamma$. In this case the brackets of $K$ is taken off as where they have become unnecessary.

In the construction equation 7.1.4 and 5 all the brackets "[" and "]" of common blocks are also taken off as their tasks have completed.

Example 7.1.1:

\[
\Psi_{65} ((pp \land [cb1] \land [cb2]) [E_i, E_o]) = \\
\Psi_{65} (((cb_2^i) \land pp \land [cb_1]) [E_i, E_b]) \\
\{ \text{ Here, } E_b \equiv E_{(cb_2,E_o)} \} \\
\land \\
((pp \land cb_1)^o \land cb_2) [E_b, E_o] \quad (e1)
\]

- By the construction equation 7.1.4

\[
(e1) = \\
\Psi_{65} (((cb_1^i \land cb_2^i) \land pp) [E_i, E_a]) \\
\{ \text{ Here, } E_a \equiv E_{(cb_1,E_o)} \} \\
\land \\
((pp^o \land cb_2) \land cb_1) [E_a, E_b] \quad (e3)
\]

- By the construction equation 7.1.4

\[
(e3) = \\
((cb_1^i \land cb_2^i) \land pp) [E_i, E_a] \quad (e5)
\]

- By the construction equation 7.1.5

Therefore we get

\[
\Psi_{65} ((pp \land [cb1] \land [cb2]) [E_i, E_o]) = \\
((cb_1^i \land cb_2^i) \land pp) [E_i, E_a] \\
\land \\
((pp^o \land cb_2) \land cb_1) [E_a, E_b] \quad (e4)
\]
Chapter 7. Decomposing to Term Transition

\[(...(pp \land cb_1)^o \land cb_2)[E_b, E_o]\]  
(e2)

We have

\[(pp \land cb_1)^o = pp^o \land cb_1^o\]

Finally, we can sort up the result above to get:

\[\Psi_{6a} ((pp \land \lbrack cb_1 \land \lbrack cb_2 \lbrack [E_i, E_o]) =\]

\[(pp \land cb_1^i \land cb_2^i) [E_i, E_a]\]  
(e5')

\[\land\]

\[(pp^o \land cb_1 \land cb_2^i) [E_a, E_b]\]  
(e4')

\[\land\]

\[(pp^o \land cb_1^o \land cb_2) [E_b, E_o]\]  
(e2')

\[\square\]

A picture below can help reader to understand our result.

\[
\begin{array}{cccc}
E_i & \rightarrow & E_a & \rightarrow & E_b & \rightarrow & E_o \\
pp & & pp^o & & pp^o & & \\
\quad cb_1^i & & \quad cb_1 & & \quad cb_1^o & & \\
\quad cb_2^i & & \quad cb_2^i & & \quad cb_2 & & \\
\end{array}
\]

From the picture we can see that the original model having logic boundary 
\([E_i, E_o]\) is divided to three pieces by introducing two middle entry predicates \(E_a\) 
and \(E_b\). In the model having logic boundary \([E_i, E_a]\), \(pp\) is implemented and 
meanwhile the input signals of \(cb_1\) and \(cb_2\) are protected. In the model having 
logic boundary \([E_a, E_b]\), \(cb_1\) is implemented and meanwhile the output signals of 
\(pp\) and the input signals of \(cb_2\) are protected. In the model having logic boundary 
\([E_b, E_o]\), \(cb_2\) is implemented and meanwhile the output signals of \(pp\) and \(cb_1\) are 
protected.

**Construction of Let_In Structure:**

To deal with a let_in structure the construction 6 uses two constructions \(\Psi_{6l1}\) 
and \(\Psi_{6l2}\) and each one of the two constructions takes the same copy of the let_in
structure but for different proposals. The $\Psi_{61}$ is used to produce a group of term transitions for the assignments selected for parallel execution. The $\Psi_{62}$ deals with the sub-structure of let in structure. The sub-structure is bracketed by two braces “{” and “}”. Moreover, the $\Psi_{62}$ replaces each variable $v$ in the sub-structure by a corresponding signal $s_v$ applied to a time point $x$. ($v$ as variable is used for description, however, $s_v$ as signal in $s_v \ x$ is used for implementation). Two constructions $\Psi_{61}$ and $\Psi_{62}$ are put into two successive time intervals and this is expressed through the $[E_i, E_m]$ and $[E_m, E_o]$, namely a new entry predicate $E_m$ is introduced to divide the $[E_i, E_o]$ of the let in structure.

In the construction equation 7.1.6 each assignment formed as

$$v = tm$$

in a group of assignments selected for parallel execution will lead to a term transition

$$s_v \ y = tm$$

and all these term transitions are conjuncted in construction equation 7.1.6 and then is put into the same derivation model (process) as shown in the construction equation 7.1.2.

In the construction equation 7.1.7 when a bracketed sub-structure $\{\beta\}$ is met, it means that the treatment for parallel assignment of a let in structure have been completed. In this case an additional work is to protect all the input signals in $\beta$.

In the construction equation 7.1.8 for the variable $v$ of each assignment “$v = tm$” we constructs a substitution “$s_v \ x/v” and append the substitution to a list $L$ which is initiated by a empty list $[\ ]$ in the construction equation 7.1.2. We use operation “:\:” for append operation $[12]$.

Let us use examples to help reader:

$$[a, b, c] :: [d, e] = [a, b, c, d, e]$$

$$[a] :: [b, c] = [a, b, c]$$

$$[\ ] :: [a, b] = [a, b]$$
\[ [a, b] :: [ ] = [a, b] \]

In the construction equation 7.1.9 the substitution list \( L \) is used to the substructure \( \beta \) and we write \( \beta L \) to express that each variable formed as \( v \) in \( \beta \) has been replaced by a corresponding signal value \( s_v x \). The brackets of \( \{ \beta \} \) are taken off as their task have completed.

**Example 7.1.2:**

\[
\Psi_6 ((\text{let } v_1 = tm_1 \text{ in let } v_2 = tm_2 \text{ in } \{S\}) [E_i, E_o]) = \\
(\Psi_{6\ell_1} (\text{let } v_1 = tm_1 \text{ in let } v_2 = tm_2 \text{ in } \{S\}) [E_i, E_m]) \wedge \\
\Psi_6 (\Psi_{6\ell_2} (\text{let } v_1 = tm_1 \text{ in let } v_2 = tm_2 \text{ in } \{S\}, [\ ] [E_m, E_o]) \\
\text{– By the construction equation 7.1.2}
\]

\[
\Psi_{6\ell_1} (\text{let } v_1 = tm_1 \text{ in let } v_2 = tm_2 \text{ in } \{S\}) = \\
s_1 y = tm_1 \wedge \\
\Psi_{6\ell_1} (\text{let } v_2 = tm_2 \text{ in } \{S\}) \quad \text{(13)}
\]

\( \text{– By the construction equation 7.1.6} \)

\[
(14) = \\
s_2 y = tm_2 \wedge \\
\Psi_{6\ell_1} (\{S\}) \quad \text{(16)}
\]

\( \text{– By the construction equation 7.1.6} \)

\[
(16) = \\
S^t \quad \text{(17)}
\]

\( \text{– By the construction equation 7.1.7} \)

Therefore we get

\[
(11) = \\
(s_1 y = tm_1 \wedge s_2 y = tm_2 \wedge S^t) [E_i, E_m] \quad \text{(18)}
\]

\[
\Psi_{6\ell_2} (\text{let } v_1 = tm_1 \text{ in let } v_2 = tm_2 \text{ in } \{S\}, [\ ])
\]
Chapter 7. Decomposing to Term Transition

\[= \Psi_{q2} \text{ (let } v_2 = tm_2 \text{ in } \{S\}, [s_1 x/v_1]) \quad \text{− By the construction equation 7.1.8}\]

\[= \Psi_{q2} \text{ (} \{S\} [s_2 x/v_2, s_1 x/v_1] \text{) } \quad \text{− By the construction equation 7.1.8}\]

\[= S [s_2 x/v_2, s_1 x/v_1] \quad \text{− By the construction equation 7.1.9}\]

Therefore,

\[(12) = \Psi_6 (\{S [s_2 x/v_2, s_1 x/v_1]\} [E_m, E_o])\]

Finally, we get

\[\Psi_6 \text{ (let } v_1 = tm_1 \text{ in } \text{let } v_2 = tm_2 \text{ in } \{S\}) [E_i, E_o] = \]

\[(v_1 y = tm_1 \land v_2 y = tm_2 \land S^i) [E_i, E_m] \]

\[\land \]

\[\Psi_6 ((S [s_2 x/v_2, s_1 x/v_1]) [E_m, E_o]) \quad \square\]

A picture below can help reader to understand our result.

\[
\begin{array}{|c|}
\hline
& E_i \\
\hline
v_1 y = tm_1 \land v_2 y = tm_2 \land S^i & E_m \\
\hline
S [v_2 x/v_2, v_1 x/v_1] & E_o \\
\hline
\end{array}
\]

Note, in the second frame box of the picture above we write \( S \) to mark that the box represents

\[\Psi_6 ((S [s_2 x/v_2, s_1 x/v_1]) [E_m, E_o]).\]

It will be similar in the example 7.1.3 for the construction of condition structure.

**Construction of Condition Structure:**

To deal with a condition structure the construction 6 uses the three constructions \( \Psi_{6p_1}, \Psi_{6p_2} \) and \( \Psi_{6p_3} \). Each one of the three constructions takes the same copy of the condition structure but does different thing. The \( \Psi_{6p_1} \) produces the term transitions for all the predicates which are selected for the parallel execution. The \( \Psi_{6p_2} \) produces a branch structure. And the \( \Psi_{6p_3} \) introduces the new derivation models for every sub-structure which is bracketed and passes them to the con-
struction $\Psi_6$. Differing from the constructions $\Psi_{6l1}$ and $\Psi_{6l2}$ for let in structure where only a new entry predicate $E_m$ is introduced to divide the let in structure, the construction for condition structure introduces a branch structure to divide the condition structure and it produces a group of term transition and several (two or two more) sub-structures to be further constructed.

In the construction equation 7.1.10 each predicate $P$ in the group of predicates selected for parallel execution leads to constructing a corresponding term transition

$$b_p y = P$$

Here, the subscript $p$ of $b_p$ represent the $b_p$ is uniquely corresponding to the predicate $P$. And all these term transitions are conjuncted in the construction equation 7.1.10 and then are put into the same model of $[E_i, E_m]$ in the construction equation 7.1.3.

In the construction equation 7.1.11 when a bracketed sub-structure $\{\beta\}$ is met, it means that the treatment for the parallel predicates has been completed so we can take off the brackets of the sub-structure and write $\beta^4$ to express that all the input signals in the $\beta$ are protected.

In the construction equation 7.1.12 for a condition structure we construct a branch structure by taking the boolean valued signal formed as $b_p$ constructed in the construction equation 7.1.10. The construction is recursive along the structure of the condition structure until meeting a bracketed sub-structure $\{\beta\}$.

The sub-structure is dealt with by the construction equation 7.1.13 which introduces a new entry predicate $E_\beta$. The subscript $\beta$ marks that the $E_\beta$ is unique for the sub-structure $\beta$. The $E_\beta$ is the leaving predicate of the branch structure and is also the entry predicate for the derivation model (process) of $\beta$.

In the construction equation 7.1.14 and 15 $\Psi_{6p3}$ recursively passes the predicates of the condition structure until meeting a bracketed sub-structure $\{\beta\}$ where
it introduce a new model (β) \([E_β, E_o]\) and then the new model is passed to \(Ψ_6\) to deal with.

**Example 7.1.3.**

\[
Ψ_6 (P \Rightarrow \{S_1\} \downarrow (Q \Rightarrow \{S_2\} \downarrow \{S_3\}) [E_i, E_m]) =
\]

\((Ψ_{6p1} (P \Rightarrow \{S_1\} \downarrow (Q \Rightarrow \{S_2\} \downarrow \{S_3\}))) [E_i, E_m]\)  \hspace{1cm} \text{(p1)}

\(\wedge\)

\[E_m, Ψ_{6p2} (P \Rightarrow \{S_1\} \downarrow (Q \Rightarrow \{S_2\} \downarrow \{S_3\})]\)  \hspace{1cm} \text{(p2)}

\(\wedge\)

\(Ψ_{6p3} (P \Rightarrow \{S_1\} \downarrow (Q \Rightarrow \{S_2\} \downarrow \{S_3\}), E_o)\)  \hspace{1cm} \text{(p3)}  

- By the construction equation 7.1.3

\[
Ψ_{6p1} (P \Rightarrow \{S_1\} \downarrow (Q \Rightarrow \{S_2\} \downarrow \{S_3\}))
\]

\[= (b_p \ y = P) \wedge Ψ_{6p1} (\{S_1\}) \wedge Ψ_{6p1} (Q \Rightarrow \{S_2\} \downarrow \{S_3\})\]

- By the construction equation 7.1.10

\[= (b_p \ y = P) \wedge S_1^t \wedge (b_q \ y = Q) \wedge Ψ_{6p1} (\{S_2\}) \wedge Ψ_{6p1} (\{S_3\})\]

- By the construction equation 7.1.11 and 10

\[= (b_p \ y = P) \wedge S_1^t \wedge S_2^t \wedge S_3^t\]

- By the construction equation 7.1.11

\[= (b_p \ y = P) \wedge (b_q \ y = Q) \wedge S_1^t \wedge S_2^t \wedge S_3^t\]

Therefore,

\(\text{(p1)} = (b_p \ y = P \wedge b_q \ y = Q \wedge S_1^t \wedge S_2^t \wedge S_3^t) [E_i, E_m].\)

\[
Ψ_{6p2} (P \Rightarrow \{S_1\} \downarrow (Q \Rightarrow \{S_2\} \downarrow \{S_3\}))
\]

\[= b_p (Ψ_{6p2} (\{S_1\}), Ψ_{6p2} (Q \Rightarrow \{S_2\} \downarrow \{S_3\}))\]

- By the construction equation 7.1.12

\[= b_p (E_{S_1}, b_q (Ψ_{6p2} (\{S_2\}), Ψ_{6p2} (\{S_3\})))\]

- By the construction equation 7.1.13 and 12

\[= b_p (E_{S_1}, b_q (E_{S_2}, E_{S_3}))\]

- By the construction equation 7.1.13

Therefore, we have
(p2) = \[ E_m, b_p (E_{S_1}, b_q (E_{S_2}, E_{S_3})) \].

(p3)
= \psi_{6p3} (\{S_1\}, E_o) \land \psi_{6p3} (Q \Rightarrow \{S_2\} \downarrow \{S_3\}, E_o)

- By the construction equation 7.1.14

= \psi_6 ((S_1) [E_{S_1}, E_o]) \land \psi_{6p3} (\{S_2\}, E_o) \land \psi_{6p3} (\{S_3\}, E_o)

- By the construction equation 7.1.15 and 14

= \psi_6 ((S_1) [E_{S_1}, E_o]) \land \psi_6 ((S_2) [E_{S_2}, E_o]) \land \psi_6 ((S_3) [E_{S_3}, E_o])

- By the construction equation 7.1.15

A picture below can help reader to understand our result.

<table>
<thead>
<tr>
<th>E_i</th>
</tr>
</thead>
<tbody>
<tr>
<td>b_p y = P \land b_q y = Q \land S_1^i \land S_2^i \land S_3^i</td>
</tr>
</tbody>
</table>

In the example 7.2 (page 58) we can see that the output signal protection \( K_3 \) of the common block “cb” includes both “pp_1^p” and “pp_2^p” as the state transition “pp_1 \land \lfloor cb \rfloor” and “pp_2 \land \lfloor cb \rfloor” share the “cb”. Generally speaking, a common block should protect all the output signals which comes from the state transitions sharing the common block (of course, these signals are ones in the private part or other common blocks which are at the left of the common block). However, the construction 6 only can separately construct the protection for each state transition, therefore after the construction 6 we need an abstract algorithm to combine all the output signal protections of a common block.
Abstract Algorithm 7.1.1. [A6: merging signal protection in common block]:

For the specification scheme which is produced by the construction 6 all the output signal protection of each common block are put together (through conjunction) as the output signal protection of the common block. □

For example, from $K_1^o \land [cb]$ and $K_2^o \land [cb]$ by the algorithm to get $K_1^o \land K_2^o \land [cb]$.

Specification Scheme 6:

$E \ 0 \land ttms$

**Term transition model set ttms:** Abbreviation:

$$ttms ::= ttm \land ttms$$

$$ttm ::= (tt) \ [E_i, E_o] \land brs$$

$$tt ::= tt \land tts$$

$$tt ::= s \ y = tm$$

$$brs ::= [E_i, E_x]$$

$$E_x ::= E_o$$

$$bs \ (E_x, E_x)$$

$E_i$ : entry predicate

$E_o$ : leaving predicate

$tm$ : term

$s$ : signal

$bs$ : boolean valued signal

The specification scheme 6 includes two kinds of models: the term transition model and branch structure model. Note, signal protection such as $s \ y = s \ x$ is included in the term transition $tt$. 
Theorem 7.1.1. [logic boundary of term transition model]:

In the scheme 6, there are not two term transition models which have the same entry predicate. Moreover, only the common block can lead its corresponding term transition models having the same leaving predicate.

Proof:

By the construction 6 a new entry predicates is always introduced to divide a state transition structure $sts$ of the scheme $5'$, therefore, the new one always becomes a middle one between the given entry and leaving predicates of the $sts$. Moreover, the new introduced predicate is always taken as the leaving predicate of a model and meanwhile as the entry predicate of another model.

The term transition models introduced by decomposing the let in structure $ls$ and condition structure $cs$ always have different entry and different leaving predicates.

The term transition models introduced by decomposing the state transition $st$, by the construction 6 and the abstract algorithm $A_6$, can be the same, therefore they are viewed and treated as the same model, or having the different entry predicate and the same leaving predicate or having both entry and leaving predicates different.

Therefore in any case we have term transition models of the scheme 6 as the theorem described.

Corollary 7.1.1. [uniqueness of entry time point and time interval]:

In the scheme 6 there are not two term transition models having the same entry time point, therefore, any two term transition models always are executed in the two different time intervals.

Proof:

The time point of a model is decided by the corresponding entry predicate of the model. And further by the lemma 7.1.1 at any time point there is only one entry predicate which can be made to be true.
7.2 Totalness and Correctness of Construction 6

Theorem 7.2.1. [totalness of construction 6]:
\[ \forall s : S_{5'}, A_6 (\Psi_6 (s)) \in S_6 \]

Proof: Use structural induction on the structure of the state transition structure
\( st_s \) of the specification scheme 5' and aware that the base case of \( st_s \) is state
transition \( st \) and step case includes the let.in structure and the condition structure.
In all cases the term transition models are produced by the construction 6 and
the abstract algorithm 6, but the only exception is the branch structure models
which is produced in the construction of the condition structure. \( \square \)

Lemma 7.2.1. [correctness of \( \Psi_{6s} \) for \( st \]):
\[ \forall st : S_{5'}, \Psi_{6s} (((K) \land st) [E_i, E_o]) \Rightarrow (K \land st) [E_i, E_o] \]

Proof:
Let us use induction on the number of sub-state transition set \( sst_s \) of the \( st \).
By the scheme 5' we know that a \( sst_s \) can be a private part pp or a common block.
Base case: There is only a sub-state transition set \( \gamma \).
In this case by the construction equation 7.1.5 we have
\[ \Psi_{6s} (((K) \land \gamma) [E_i, E_o]) = (K \land \gamma') [E_i, E_o] \]
where, if \( \gamma = [\alpha] \) then \( \gamma' = \alpha \) otherwise \( \gamma' = \gamma \)
Namely, the brackets are taken off as the brackets are only used to mark
for common block.
Therefore we can get
\[ \Psi_{6s} (((K) \land \gamma) [E_i, E_o]) \Rightarrow (K \land \gamma) [E_i, E_o] \]

Step case: Suppose \( st = \alpha \land [\beta] \)
where, \( \alpha \) contains one or more conjuncted \( sst_s \) and \( \beta \) is a \( sst_s \). By the construction
equation 7.1.4 we have
\[ \Psi_{6s} \left( (((K \land \alpha) \land [\beta]) \left[ E_i, \ E_o \right]) \right) \overset{\Delta}{=} \Psi_{6s} \left( (((\beta^d \land K) \land \alpha) \left[ E_i, \ E_m \right]) \right) \land \left( (\alpha^o \land K) \land \beta \right) \left[ E_m, \ E_o \right] \]

By induction hypothesis we have
\[ \Psi_{6s} \left( (((\beta^d \land K) \land \alpha) \left[ E_i, \ E_m \right]) \right) \Rightarrow \left( (\beta^d \land K) \land \alpha \right) \left[ E_i, \ E_m \right] \]

Hence our proof is reduced to
\[ \left( (\beta^d \land K) \land \alpha \right) \left[ E_i, \ E_m \right] \land \left( (\alpha^o \land K) \land \beta \right) \left[ E_m, \ E_o \right] \Rightarrow \left( K \land \alpha \land [\beta] \right) \left[ E_i, \ E_o \right] \]

This is obviously tenable. \[ \square \]

The specification for let.in structure in the scheme 5' is described as
\[
\begin{align*}
sts & ::= \text{let } A \text{ in } ls \\
ls & ::= \{sts\} \\
& | \text{let } A \text{ in } ls
\end{align*}
\]

By structural induction to prove the correctness of the construction for the let.in structure of (11) we can take the hypothesis that the constructions for the (12) and (13) are correct. However, in the proof of the lemma below we need not to distinguish the two case but only consider them as \( ls \).

**Lemma 7.2.2.** [correctness of construction for let.in structure]:
\[ \Psi_6 \left( \left( \text{let } v = tm \text{ in } ls \right) \left[ E_i, \ E_o \right] \right) \Rightarrow \left( \text{let } v = tm \text{ in } ls \right) \left[ E_i, \ E_o \right] \]
if
\[ \Psi_6 \left( ls \left[ E_x, \ E_y \right] \right) \Rightarrow ls \left[ E_x, \ E_y \right] \]

Note: all entry predicates in the lemma can be arbitrarily given because the
structural induction merely concerns the structure of $sts$. For example, here we use $[E_x, E_y]$.

**Proof:**

$$
\Psi_6 \left( \langle \text{let } v = \text{tm in } ls \rangle \ [E_i, E_o] \right) \\
= \Psi_6 \left( \langle ls \ [tm/v] \rangle \ [E_i, E_o] \right) \\
\Rightarrow \langle ls \ [tm/v] \rangle \ [E_i, E_o] - \text{By the hypothesis of the lemma} \\
= \langle \text{let } v = \text{tm in } ls \rangle \ [E_i, E_o] - \text{By the definition of let.in}
$$

Therefore we get

$$
\Psi_6 \left( \langle \text{let } v = \text{tm in } ls \rangle \ [E_i, E_o] \right) \Rightarrow \langle \text{let } v = \text{tm in } ls \rangle \ [E_i, E_o] 
$$

The specification for condition structure in the scheme $5'$ is described as

\[
sts ::= P \Rightarrow cs \downarrow cs \\
| P \Rightarrow cs \downarrow cs
\]

(p1)

\[
| \{sts\}
\]

(p2)

By structural induction to prove the correctness of the construction for the condition structure of (p1) we can take the hypothesis that the constructions for the (p2) and (p3) are correct.

**Lemma 7.2.3. [correctness of construction for condition structure]:**

$$
\Psi_6 \left( \langle P \Rightarrow R \downarrow S \rangle \ [E_i, E_o] \right) \Rightarrow \langle P \Rightarrow R \downarrow S \rangle \ [E_i, E_o] \\
\text{if}
$$

$$
\Psi_6 \left( R \ [E_x, E_y] \right) \Rightarrow R \ [E_x, E_y] \\
\text{and}
$$

$$
\Psi_6 \left( S \ [E_u, E_v] \right) \Rightarrow S \ [E_u, E_v]
$$

**Proof:**

For each of $R$ and $S$ we should discuss two cases according to (p2) and (p3), therefore there are four combinations. Here, for short we only discuss two combinations (the other two are similar):

1. Both $R$ and $S$ are of case (p2).
2. $R$ is of case (p2) and $S$ is of case (p3)
In (1) and (2) below let us follow such a proof structure:

Suppose \( \Psi_6 \left( (P \Rightarrow R \downarrow S) \left[ E_i, E_o \right] \right) \)

To get \( (P \Rightarrow R \downarrow S) \left[ E_i, E_o \right] \)

(1). In this case by the construction 6 we have

\[
\Psi_6 \left( (P \Rightarrow \{S_1\} \downarrow \{S_2\}) \left[ E_i, E_o \right] \right) \\
\overset{\text{(L3.1)}}{=} \left( b_p \ y = P \land S_1^t \land S_2^t \right) \left[ E_i, E_m \right] \\
\land \\
\left[ E_m, b_p \ (E_1, E_2) \right] \overset{\text{(L3.2)}}{=} \\
\land \\
\Psi_6 \left( (S_1) \left[ E_1, E_o \right] \right) \\
\land \\
\Psi_6 \left( (S_2) \left[ E_2, E_o \right] \right)
\]

By the hypothesis of the lemma we have

\[
\Psi_6 \left( (S_1) \left[ E_1, E_o \right] \right) \Rightarrow (S_1) \left[ E_1, E_o \right] \\
\text{and} \\
\Psi_6 \left( (S_2) \left[ E_2, E_o \right] \right) \Rightarrow (S_2) \left[ E_2, E_o \right]
\]

Therefore we can get

\[
(L3.1) \land (L3.2) \land (S_1) \left[ E_1, E_o \right] \land (S_2) \left[ E_2, E_o \right] \overset{\text{(L3.3)}}{=}
\]

Given \( E_t \ t_1 \) for an arbitrary time point \( t_1 \) and suppose that \( P \) is true at \( t_1 \), by

(L3.1) we get a time point \( t_2 \) such that

\[
b_p \ t_2 = P \text{ and } E_m \ t_2
\]

hold, therefore \( b_p \ t_2 \) gets true, and \( S_1^t \) and \( S_2^t \) hold for \( t_1 \) and \( t_2 \), namely, all the

input signals in \( S_1 \) and \( S_2 \) are protected for two time points \( t_1 \) and \( t_2 \)

By (L3.2) and \( E_m \ t_2 \) and \( b_p \ t_2 \) we get \( E_1 \) to be true at \( t_2 \). Then by \( (S_1) \left[ E_1, E_o \right] \)

we have a time point \( t_3 \) such that \( S_1 \) holds for \( t_2 \) and \( t_3 \) and \( E_o \) is true at \( t_3 \). By

\( S_1^t \) we further have \( S_1 \) holds for \( t_1 \) and \( t_3 \).
Therefore given $E_i t_1$ of the arbitrary $t_1$ and suppose $P$ to be true at $t_1$ we get a $t_3$ such that $S_1$ holds for $t_1$ and $t_3$ and $E_o$ holds at $t_3$.

It is similar for the case where we start from another assumption that $P$ to be false to get $S_2$.

Therefore, we get

\[(P \Rightarrow S_1 \downarrow S_2) \ [E_i, E_o]\]

(2). In this case by the construction 6 we have

\[
\Psi_6 ((P \Rightarrow \{S\} \downarrow (Q \Rightarrow \alpha \downarrow \beta)) \ [E_i, E_o])
= (b_p \ y = P \land b_q \ y = Q \land S^I \land \ b_1 \ y = P_1 \land \cdots \land b_k \ y = P_k \land S^I_1 \land \cdots \land S^I_n) \ [E_i, E_m]
\]

\[
\land
\]

\[
[E_m, b_p (E_s, b_q (E_\alpha, E_\beta))]
\]

\[
\land
\]

\[
\Psi_6 ((S) \ [E_s, E_o])
\]

\[
\land
\]

\[
\Psi_6 ((S_1) \ [E_1, E_o])
\]

\[
\land \cdots \land
\]

\[
\Psi_6 ((S_n) \ [E_n, E_o])
\]

Here, we suppose $\alpha$ and $\beta$ contain $k$ predicates $P_1, \cdots, P_k$ and and $n$ bracketed sub-structures $\{S_1\}, \cdots, \{S_n\}$. Further, for (L3.6) by the branch structure model 7.1 (page 56) we have

\[
[E_m, b_p (E_s, b_q (E_\alpha, E_\beta))]
\]

\[
= \forall x. E_m x \Rightarrow (b_p x \Rightarrow E_s x) \downarrow (b_q x \Rightarrow S_m (E_\alpha) \downarrow S_m (E_\beta))
\]

\[
= \forall x. E_m x \Rightarrow (b_p x \Rightarrow E_s x) \land (\neg b_p x \Rightarrow (b_q x \Rightarrow S_m (E_\alpha) \downarrow S_m (E_\beta)))
\]  

(L3.7)

Suppose given $E_i t_1$ for an arbitrary time point $t_1$ and suppose $P$ has false value at $t_1$. From (L3.5) we can have a time point $t_2$ such that $E_m t_2$ is true and $b_1 t_2$ is false.
Then by (L3.7) for any time point \( x \) and suppose \( b_p x \) false we can get

\[
\forall x. \ E_m \ x \Rightarrow (b_q x \Rightarrow S_m (E_\alpha) \downarrow S_m (E_\beta))
\]

This is

\[
[E_m, b_q (E_\alpha, E_\beta)]
\]  \hspace{1cm} \text{(L3.8)}

From (L3.5) we can get

\[
(b_q y = Q \land b_1 y = P_1 \land \cdots \land b_k y = P_k \land S_1^d \land \cdots \land S_n^d) \ [E_i, E_m]
\]  \hspace{1cm} \text{(L3.9)}

And from (L3.4) we have

\[
\Psi_6 ((S_1) [E_1, E_o]) \land \cdots \land \Psi_6 ((S_n) [E_n, E_o])
\]  \hspace{1cm} \text{(L3.10)}

Hence from (L3.4) we can get

\[
(L3.9) \land (L3.8) \land (L3.10)
\]

By the construction 6 this is just

\[
\Psi_6 ((Q \Rightarrow \alpha \downarrow \beta) [E_i, E_o])
\]

Then by the hypothesis of the lemma we have

\[
(Q \Rightarrow \alpha \downarrow \beta) [E_i, E_o]
\]  \hspace{1cm} \text{(L3.11)}

Therefore we have that

from a given \( E_i t_1 \) of the arbitrary \( t_1 \) and by (L3.11)

we can get a time point \( t_3 \) such that

\[
Q \Rightarrow \alpha \downarrow \beta
\]

holds for \( t_1 \) and \( t_3 \) and

\[
E_o t_3
\]

holds.

As (L3.11) is got from (L3.4) therefore we get that

(A). suppose given \( E_i t_1 \) of the arbitrary \( t_1 \) and suppose \( P \) is false at \( t_1 \)

from (L3.4) we can get a time point \( t_3 \) such that

\[
Q \Rightarrow \alpha \downarrow \beta
\]
holds for \( t_1 \) and \( t_3 \) and
\[
E_0 \ t_3
\]
holds.

It is similar to the preceding (1) of this lemma that we have that (B). suppose given \( E_i \ t_1 \) of the arbitrary \( t_1 \) and suppose \( P \) is true at \( t_1 \) from (L3.4) we can get a time point \( t_3 \) such that
\[
S
\]
holds for \( t_1 \) and \( t_3 \) and
\[
E_0 \ t_3
\]
holds.

From the two conclusions (A) and (B) above we can finally get
\[
(P \Rightarrow \{S\} \downarrow (Q \Rightarrow \alpha \downarrow \beta)) \ [E_i, \ E_0]. \quad \Box
\]

**Theorem 7.2.2. [correctness of construction 6]:**

\[ \forall \text{sts} : \mathcal{S}_{\text{st}}. \ \mathcal{A}_0 (\Psi_6 (\text{sts} [E_i, \ E_o])) \Rightarrow (\text{sts} [E_i, \ E_o]) \]

**Proof:** Let us use structural induction on the structure of \( \text{sts} \).

**Base case:** \( \text{sts} = \text{st} \). In the case
\[
\Psi_6 (\text{st} [E_i, \ E_o])
\]
\[= \Psi_6 (((T) \land \text{st}) [E_i, \ E_o]) \quad \text{– By the construction equation 7.1.1}
\]
\[\Rightarrow ((T) \land \text{st}) [E_i, \ E_o] \quad \text{– By the lemma 7.2.1}
\]
\[\Leftrightarrow (\text{st}) [E_i, \ E_o] \quad \text{– as } (T \land \text{st}) \Leftrightarrow \text{st}
\]

**Step case:** There are further two case (1) and (2).

(1). \( \text{sts} = \text{let A in ls} \)

In the case taking the induction hypothesis:
\[
\Psi_6 (\text{ls} [E_x, \ E_y]) \Rightarrow \text{ls} [E_x, \ E_y]
\]
and using the lemma 7.2.2 we get:
\[ \Psi_6 \left( (\text{let } A \text{ in } l) \ [E_i, E_o] \right) \Rightarrow (\text{let } A \text{ in } l) \ [E_i, E_o] \]

(2). \( sts = P \Rightarrow R \downarrow S \)

In this case taking the induction hypothesis:

\[ \Psi_6 \left( R \ [E_x, E_y] \right) \Rightarrow R \ [E_x, E_y] \]

and

\[ \Psi_6 \left( S \ [E_u, E_v] \right) \Rightarrow S \ [E_u, E_v] \]

and using the lemma 7.2.3 we get:

\[ \Psi_6 \left( (P \Rightarrow R \downarrow S) \ [E_i, E_o] \right) \Rightarrow (P \Rightarrow R \downarrow S) \ [E_i, E_o] \]

By the base and step cases above we get:

\[ \Psi_6 \left( (sts) \ [E_i, E_o] \right) \Rightarrow (sts) \ [E_i, E_o] \]

By the abstract algorithm 6 \( A_6 \) we have

\[ A_6 \left( \Psi_6 \left( (sts) \ [E_i, E_o] \right) \right) \Rightarrow \Psi_6 \left( (sts) \ [E_i, E_o] \right) \]

Finally, we get

\[ A_6 \left( \Psi_6 \left( (sts) \ [E_i, E_o] \right) \right) \Rightarrow (sts) \ [E_i, E_o] \] \( \square \)
Chapter 8

Decomposing to Function Unit and Connection

In this chapter we present a generic process to decompose a term transition to function unit and connection. By function unit and connection we mean the elementary specification pieces which can be directly implemented by hardware devices at register transfer level which is characterized by using register and bus to “glue” up function units. We will demonstrate how to use derivation model to express and treat the decomposition process.

We have an assumption that each function in the term transition is implementable at register transfer level. If not then we further assume that through data refinement it is always to be able to transformed to such functions.

Let us use an example below to illustrate the basic decomposition process.

**Specification 1:**
\[ \forall x. \exists y. E_i x \Rightarrow s_1 y = f (g (r_1 x)) \land s_2 y = h (r_2 x) \land E_o y \]

For short we rewrite it:
\[ (s_1 y = f (g (r_1 x)) \land s_2 y = h (r_2 x)) [E_i, E_o] \]

We start decomposition with two sub-specifications below:
\[ s_1 y = f (g (r_1 x)) [E_i, E_o] \quad \text{(a1)} \]

82
Chapter 8. Decomposing to Function Unit and Connection

\[ s_2 y = h (r_2 x) [E_i, E_o] \]  
(a2)

Decomposing to function unit and connection level:

\[(i_1 y = r_1 x) [E_i, E_1] \land (o_1 y = g (i_1 x)) [E_1, E_2] \land (i_2 y = o_1 x) [E_2, E_3] \land \]
\[(o_2 y = f (i_2 x)) [E_3, E_4] \land (s_1 y = o_2 x) [E_4, E_o] \]  
(b1)

\[(i_3 y = r_2 x) [E_i, E_a] \land (o_3 y = h (i_3 x)) [E_a, E_b] \land \]
\[(s_2 y = o_3 x) [E_b, E_o] \]  
(b2)

Here we write “c” to represent the specification body is a connection.

We select a device to implement the function \( f \). And we select another device to implement both the function units \( g \) and \( h \), therefore we unified their inputs and outputs to be the same ones: \( i \) and \( o \).

\[(i y = r_1 x) [E_i, E_1] \land (o y = g (i x)) [E_1, E_2] \land (i_2 y = o x) [E_2, E_3] \land \]
\[(o_2 y = f (i_2 x)) [E_3, E_4] \land (s_1 y = o_2 x) [E_4, E_o] \]  
(c1)

\[(i y = r_2 x) [E_i, E_a] \land (o y = h (i x)) [E_a, E_b] \land \]
\[(s_2 y = o x) [E_b, E_o] \]  
(c2)

We can put \( f \) and \( h \) into a model as they are selected to implement by two different devices.
We divide the model
\[(i \ y = r_2 \ x) \ [E_i, \ E_a]\]
into three models which match the models
\[(i \ y = r_1 \ x) \ [E_i, \ E_1] \land (o \ y = g \ (i \ x)) \ [E_1, \ E_2] \land (i_2 \ y = o \ x) \ [E_2, \ E_3]\]
Then compose each pair of corresponding models to get:
\[(i \ y = r_1 \ x \land r_2 \ y = r_2 \ x) \ [E_i, \ E_1]\] (d1)
\[\land\]
\[(o \ y = g \ (i \ x) \land r_2 \ y = r_2 \ x) \ [E_1, \ E_2]\] (d2)
\[\land\]
\[(i_2 \ y = o \ x \land i \ y = r_2 \ x) \ [E_2, \ E_3]\] (d3)
\[\land\]
\[(o_2 \ y = f \ (i_2 \ x) \land o \ y = h \ (i \ x)) \ [E_3, \ E_4]\] (d4)
\[\land\]
\[(s_1 \ y = o_2 \ x \land s_2 \ y = o \ x) \ [E_4, \ E_o]\] (d5)

8.1 Construction 7 and Specification Scheme 6' and 7

8.1.1 Specification 6'

As the extension of the scheme 6 in the scheme 6' we give the details of term \(tm\) which will be dealt with by construction 7.

Specification Scheme 6':
\( E \ 0 \land \ttms \)

**Term transition model set \( \ttms \):** Abbreviation:

\[
\ttms ::= \ttm \\
\quad \mid \ttm \land \ttms
\]

\[
\ttm ::= (\ttt) [E_i, E_o] \\
\quad \mid \bbrs \\
\ttt ::= \tt \\
\quad \mid \tt \land \ttt
\]

\[
\tt ::= s \ y = \tm \\
\tm ::= c \\
\quad \mid s \ x \\
\quad \mid f (\tm, \cdots, \tm)
\]

\( \ttm \) : term transition model

\( \ttt \) : term transition set

\( \tt \) : term transition

\( \tm \) : term

\( \ttt \) : term transition set

\( \ttm \) : term transition model

\( \bbrs \) : branch structure model

Note, (1). in the scheme the function \( f \) has covered the predicate and predicate connective. (2). A different point for the term \( \tm \) of the scheme from one of the behavior specification scheme is that it does not contain variable as which have been eliminated in the construction 6.

### 8.1.2 Linear Chain

Linear chain is the entity which the construction 7 mainly deals with. In this section let us define what a linear chain is. First we introduce three primary definitions.

**Definition 8.1.2.1. [\( f_u \) : function unit]:**

A function unit \( f_u \) is a specification body formed as:

\( o \ y \overset{a}{=} f (i_1 \ x, \cdots, i_n \ x) \)

Here, “a” represents that a device has been allocated to implement the function unit, therefore the ports of the unit, \( o, i_1, \cdots, i_n \), are the same as the ports of the
device. We call $o$ the output (port) of $fu$ and $i_1, \cdots, i_n$ the inputs (port) of $fu$.

\[ \square \]

Note: here and elsewhere, we often use port instead of signal to emphasize the implementation aspect of a specification.

A corollary from the definition 8.1.2.1 is that if two function unit are implemented by the same device then their corresponding ports are the same.

**Definition 8.1.2.2. [cu : connection unit]:**

A connection unit $cu$ is a specification body formed as:

\[ s \ y = r \ x \]

Here, $s$ and $r$ are different signals and we call $s$ and $r$ output and input (port) of $cu$ respectively. We also take the boolean constant $T$ as a special connection unit.

\[ \square \]

**Definition 8.1.2.3. [du : delay unit]:**

A delay unit $du$ is a specification body formed as:

\[ s \ y = s \ x \]

Here, $s$ is a signal.

\[ \square \]

**Definition 8.1.2.4. [fl: function link]:**

A function link $fl$ is a conjunction of one or more function units and zero or more delay units.

\[ \square \]

**Definition 8.1.2.5. [cl: connection link]:**

A connection link $cl$ is a conjunction of one or more connection units and zero or more delay units.

\[ \square \]

**Definition 8.1.2.6. [sl: special link]:**

A special link $sl$ is a conjunction of one or more delay units. We assume that it can be used as function link or as connection link.

\[ \square \]

**Definition 8.1.2.7. [tfl: tolerant function link]:**

A function link is tolerant if in the link there are not two function units to be
implemented by the same device. This means that there are not any two ports in
the link to be the same.

\textbf{Definition 8.1.2.8. [tcl: tolerant connection link]}:

A connection link is tolerant if in the link there are not two connection units
having the same output.

\textbf{Definition 8.1.2.9. [linear chain]}:

A linear chain is formed as

\[(tcl_1) [E_1, E_2] \land (tfl_2) [E_2, E_3] \land (tcl_3) [E_3, E_4] \land \cdots \land \]
\[(tcl_{n-3}) [E_{n-3}, E_{n-2}] \land (tfl_{n-2}) [E_{n-2}, E_{n-1}] \land (tcl_{n-1}) [E_{n-1}, E_n].\]

Here, \(n\) is an odd number. When \(n = 1\) we define a linear chain is

\[(tcl) [E_1, E_2].\]

Intuitively, a linear chain is a sequence of derivation models along an entry pre-
dicate sequence \(E_1, E_2, \cdots, E_n\) and the specification bodies of these models are
alternate of tolerant connection links and tolerant function links.

For a linear chain or the part of a linear chain form as

\[\alpha_1 [E_1, E_2] \land \alpha_2 [E_2, E_3] \land \cdots \land \alpha_{n-1} [E_{n-1}, E_n]\]

we call \(E_1\) and \(E_n\) as its head node and tail node respectively; \(\alpha_1 [E_1, E_2]\) and
\(\alpha_{n-1} [E_{n-1}, E_n]\) as its head link and tail links respectively.

An example of linear chain can be taken from the preceding example: (page 84)

\[(i \ y = r_1 \ x \land r_2 \ y = r_2 \ x) [E_i, E_1]\]  \hspace{1cm} (d1)
\land
\[(o \ y = g \ (i \ x) \land r_2 \ y = r_2 \ x) [E_1, E_2]\]  \hspace{1cm} (d2)
\land
\[(i_2 \ y = o \ x \land i \ y = r_2 \ x) [E_2, E_3]\]  \hspace{1cm} (d3)
\land
\[(o_2 \ y = f \ (i_2 \ x) \land o \ y = h \ (i \ x)) [E_3, E_4]\]  \hspace{1cm} (d4)
\begin{align*}
\land \\
(s_1 y = o_2 x \land s_2 y = o x) [E_4, E_0] \quad (d5)
\end{align*}

Here, (d1), (d3) and (d5) are tolerant connection link models, (d2) and (d4) are tolerant function link models.

### 8.1.3 Construction 7

Construction 7 deals with each term transition model of the scheme $6'$ to transform it to a linear chain.

**Definition 8.1.3.1. [abbreviation for model sequence]:**

We write $\Gamma_{(i,j)}^\alpha$ as abbreviation for

$$\alpha_i [E_i^\alpha, E_{i+1}^\alpha] \land \alpha_{i+1} [E_{i+1}^\alpha, E_{i+2}^\alpha] \land \cdots \land \alpha_j [E_j^\alpha, E_{j+1}^\alpha]$$

Note: we write $E_i^\alpha$ to emphasize that the entry predicate belongs to a model whose specification body is $\alpha$ (with subscript if necessary). \hfill \Box

For example, we write $\Gamma_{(3,5)}^\beta$ for $\beta_3 [E_3, E_4] \land \beta_4 [E_4, E_5] \land \beta_5 [E_5, E_6]$

**Definition 8.1.3.2. [abbreviation for fu and cu]:**

We write $\alpha_f^\alpha$ for a function unit and $\alpha_c^\alpha$ for a connection unit. \hfill \Box

**Definition 8.1.3.3. [merging two function links]:**

Two function links are merge-able if putting them together through conjunction we can get a tolerant function link. \hfill \Box

**Definition 8.1.3.4. [merging two connection links]:**

Two connection links are merge-able if putting them together through conjunction we can get a tolerant connection link. \hfill \Box
Construction 7:

8.1.3.1. \( \Psi_7 ((tts) [E_i, E_o]) \equiv \Psi_{7m} (\Psi_{7s} ((tts) [E_i, E_o])) \)

8.1.3.2. \( \Psi_{7s} ((tt) [E_i, E_o]) \equiv \Psi_{7l} ((tt) [E_i, E_o]) \)

8.1.3.3. \( \Psi_{7s} ((tt \land tts) [E_i, E_o]) \equiv \Psi_{7l} ((tt) [E_i, E_o]), \Psi_{7s} ((tts) [E_i, E_o]) \)

8.1.3.4. \( \Psi_{7l} ((s y = r x) [E_i, E_o]) \equiv (s y = r x) [E_i, E_o] \)

Here, \( r \) is a signal.

8.1.3.5. \( \Psi_{7l} ((s y = c) [E_i, E_o]) \)
\( \equiv (s y = o x) [E_i, E_o] \land (o y \overset{a}{=} c) [E_a, E_o] \land \top [E_o, E_o] \)

Here and elsewhere, when the "\( = \)" of a function unit is replaced by "\( \overset{a}{=} \)”, this represents that a device has been allocated to implement the unit.

In this case we have an assumption:

Assumption 8.1.3.1. [unifying ports]:

For the function units assigned to the same device to implement their ports (inputs and outputs) are unified to be the same. (However, still to be kept as existent variables). \( \square \)

8.1.3.6. \( \Psi_{7l} ((s y = f (tm_1, \ldots, tm_n) [E_i, E_o]) \equiv \)
\( \Psi_7 ((i_1 y = tm_1 \land \cdots \land i_n y = tm_n) [E_i, E_o]) \land \)
\( (o y \overset{a}{=} f (i_1 x, \cdots, i_n x)) [E_a, E_b] \land \)
\( (s y = o x) [E_b, E_o] \)

Starting from \( \Psi_{7m} \) we have an assumption that for each construction all its arguments, except specially stated, are the linear chains or the part of a linear chain, which have the same head node and tail node. For example, in the construction 8.1.3.12 for \( \Psi_{7c} (\Gamma_1, \Gamma_2) \) we have that \( \Gamma_1 \) and \( \Gamma_2 \) have the same head and tail nodes.
8.1.3.7. \( \Psi_{7m} (\alpha) \equiv \alpha \)

8.1.3.8. \( \Psi_{7m} (\alpha_1, \alpha_2) \equiv \Psi_{7f} (\alpha_1, \alpha_2) \)

8.1.3.9. \( \Psi_{7m} (\alpha_1, \alpha_2, \alpha_3, \cdots, \alpha_n) \equiv \Psi_{7m} (\Psi_{7f} (\alpha_1, \alpha_2), \alpha_3, \cdots, \alpha_n) \)

For constructions \( \Psi_{7f}, \Psi_{7c} \) and \( \Psi_{7e1} \) below we assume their two arguments to be commutative for example, we have

\[ \Psi_{7f} (\alpha, \beta) = \Psi_{7f} (\beta, \alpha) \]

And for \( \Psi_{7c} \) and \( \Psi_{7e2} \) below we assume their first two arguments to be commutative, for example, we have

\[ \Psi_{7c} (\alpha, \beta, \gamma) = \Psi_{7c} (\beta, \alpha, \gamma) \]

8.1.3.10. \( \Psi_{7f} (\Gamma_1, \Gamma_2) \equiv \Psi_{7c} (\Gamma_1, \Gamma_2) \)

For construction equation 8.1.3.10 we suppose that there are not two function units from \( \Gamma_1 \) and \( \Gamma_2 \) respectively being merge-able.

8.1.3.11. \( \Psi_{7f} (\Gamma_{(0,i-1)}^\alpha \land \alpha_i^\rho [E_i^\alpha, E_{i+1}^\alpha] \land \Gamma_{(i+1,m)}^\alpha, \Gamma_{(0,j-1)}^\beta \land \beta_j^\rho [E_j^\beta, E_{j+1}^\beta] \land \Gamma_{(j+1,n)}^\beta) \)

\( \equiv \Psi_{7f} (\Gamma_{(0,i-1)}^\alpha, \Gamma_{(0,j-1)}^\beta) \land \\
(\alpha_i^\rho \land \beta_j^\rho) [E_x, E_y] \land \\
\Psi_{7f} (\Gamma_{(i+1,m)}^\alpha, \Gamma_{(j+1,n)}^\beta) \)

For construction equation 8.1.3.11 we suppose that there are at least a pair of function units from \( \Gamma^\alpha \) and \( \Gamma^\beta \) respectively being merge-able. And we arbitrarily select a pair:

\( \alpha_i^\rho [E_i^\alpha, E_{i+1}^\alpha] \) and \( \beta_j^\rho [E_j^\beta, E_{j+1}^\beta] \)

to be merged as the shown in the construction equation.

Note: here, we introduce \( E_x \) and \( E_y \) and have \( E_i^\alpha = E_j^\beta = E_x \)
and \( E_{i+1}^\alpha = E_{j+1}^\beta = E_y \)
8.1.3.12. $\Psi_{7c} (\Gamma_1, \Gamma_2) \triangleq \Psi_{7c1} (\Gamma_1, \Gamma_2)$

For construction equation 8.1.3.12 we suppose that there are not two connection units from $\Gamma_1$ and $\Gamma_2$ respectively being merge-able.

8.1.3.13. $\Psi_{7c} \ (\Gamma_{(0,i,-1)}^\alpha \land \alpha_i^C \ [E_i^\alpha, E_{i+1}^\alpha] \land \Gamma_{(i+1,m)}^\alpha, \ 
\Gamma_{(0,j,-1)}^\beta \land \beta_j^C \ [E_j^\beta, E_{j+1}^\beta] \land \Gamma_{(j+1,n)}^\beta) \ 
\triangleq \Psi_{7c2} (\Gamma_1^L, \Gamma_2^L, \gamma) \land \n(\alpha_i^C \land \beta_j^C) [E_x, E_y] \land \n\Psi_{7c'} (\Gamma_1^R, \Gamma_2^R, \gamma')$.

For construction equation 8.1.3.13 we suppose that there are at least a pair of connection units from $\Gamma^\alpha$ and $\Gamma^\beta$ respectively being merge-able. And we select the pair which first appears in the two chains along direction from left to right: $\alpha_i^C [E_i^\alpha, E_{i+1}^\alpha]$ and $\beta_j^C [E_j^\beta, E_{j+1}^\beta]$ to be merged as the shown in the construction equation.

In the equation we write $\Psi_{7c2} (\Gamma_1^L, \Gamma_2^L, \gamma)$ to express three possible cases:

1. $\Psi_{7c2} (\Gamma_{(0,i,-1)}^\alpha, \Gamma_{(0,j,-1)}^\beta, \gamma)$, in this case both $\Gamma_{(0,i,-1)}^\alpha$ and $\Gamma_{(0,j,-1)}^\beta$ are not empty and $\gamma$ is useless.

2. $\Psi_{7c2} (\Gamma_{(0,i,-1)}^\alpha, \Gamma_{(0,j,-1)}^\beta, \gamma)$, in this case $\Gamma_{(0,j,-1)}^\beta$ is empty (boolean constant T) and $\gamma$ is $\beta_j^i$.

3. $\Psi_{7c2} (\Gamma_{(0,j,-1)}^\beta, \Gamma_{(0,i,-1)}^\alpha, \gamma)$, in this case $\Gamma_{(0,i,-1)}^\alpha$ is empty and $\gamma$ is $\alpha_i^i$. 
Similarly, we write $\Psi_{7c'} (\Gamma_1^r, \Gamma_2^r, \gamma')$ to express three possible cases:

1. $\Psi_{7c'} (\Gamma_{(i+1,m)}^\alpha, \Gamma_{(j+1,n)}^\beta, \gamma')$, in this case both $\Gamma_{(i+1,m)}^\alpha$ and $\Gamma_{(j+1,n)}^\beta$ are not empty and $\gamma'$ is useless.

2. $\Psi_{7c'} (\Gamma_{(i+1,m)}^\alpha, \Gamma_{(j+1,n)}^\beta, \gamma')$, in this case $\Gamma_{(j+1,n)}^\beta$ is empty and $\gamma'$ is $\alpha_i^\beta$.

3. $\Psi_{7c2} (\Gamma_{(j+1,n)}^\beta, \Gamma_{(i+1,m)}^\alpha, \gamma')$, in this case $\Gamma_{(i+1,m)}^\alpha$ is empty and $\gamma'$ is $\alpha_i^\alpha$.

Note: here, we introduce $E_x$ and $E_y$ and have $E_i^\alpha = E_j^\beta = E_x$ and $E_{i+1}^\alpha = E_{j+1}^\beta = E_y$.

8.1.3.14. $\Psi_{7c'} (\Gamma_1, \Gamma_2, \gamma) \equiv \Psi_{7c2} (\Gamma_1, \Gamma_2, \gamma')$

For the construction equation 8.1.3.14 we assume that there are not two connection units from $\Gamma_1$ and $\Gamma_2$ respectively being mergeable. This includes that case that $\Gamma_2$ is empty linear chain, i.e. $\Gamma_2 = T$ and the case that both $\Gamma_1$ and $\Gamma_2$ are empty linear chain.

8.1.3.15. $\Psi_{7c'} (\Gamma_{(0,i-1)}^\alpha \wedge \alpha_i^c \ [E_i^\alpha, E_{i+1}^\alpha] \wedge \Gamma_{(i+1,m)}^\alpha, \Gamma_{(0,j-1)}^\beta \wedge \beta_j^c \ [E_j^\beta, E_{j+1}^\beta] \wedge \Gamma_{(j+1,n)}^\beta, \gamma')$

$\equiv \Psi_{7c1} (\Gamma_{(0,i-1)}^\alpha, \Gamma_{(0,j-1)}^\beta) \wedge$

$(\alpha_i^c \wedge \beta_j^c) \ [E_x, E_y] \wedge$

$\Psi_{7c'} (\Gamma_1^r, \Gamma_2^r, \gamma')$
Chapter 8. Decomposing to Function Unit and Connection

For construction equation 8.1.3.15 we suppose that there are at least a pair of connection units from \( \Gamma^\alpha \) and \( \Gamma^\beta \) respectively being merge-able. And we select the pair which first appears in the two chains along direction from left to right:

\[
\alpha^\alpha_i \left[ E^\alpha_i, E^\alpha_{i+1} \right] \text{ and } \beta^\beta_j \left[ E^\beta_j, E^\beta_{j+1} \right]
\]

to be merged as shown in the construction equation.

Similar to the construction equation 8.1.3.13, we write \( \Psi_{7c} \left( \Gamma^\alpha_1, \Gamma^\alpha_2, \gamma' \right) \) to express three possible cases:

1. \( \Psi_{7c} \left( \Gamma^\alpha_{(i+1,m)}, \Gamma^\beta_{(j+1,n)}, \gamma' \right) \), in this case both \( \Gamma^\alpha_{(i+1,m)} \) and \( \Gamma^\beta_{(j+1,n)} \) are not empty and \( \gamma' \) is useless.

2. \( \Psi_{7c} \left( \Gamma^\alpha_{(i+1,m)}, \Gamma^\beta_{(j+1,n)}, \gamma' \right) \), in this case \( \Gamma^\beta_{(j+1,n)} \) is empty and \( \gamma' \) is \( \beta^\beta_j \).

3. \( \Psi_{7c} \left( \Gamma^\beta_{(j+1,n)}, \Gamma^\alpha_{(i+1,m)}, \gamma' \right) \), in this case \( \Gamma^\alpha_{(i+1,m)} \) is empty and \( \gamma' \) is \( \alpha^\alpha_i \).

Note: here, we introduce \( E_x \) and \( E_y \) and have \( E^\alpha_i = E^\beta_j = E_x \) and \( E^\alpha_{i+1} = E^\beta_{j+1} = E_y \).

8.1.3.16. \( \Psi_{7e_1} \left( \Gamma^\alpha_{(a,b)}, \Gamma^\beta_{(c,d)} \right) \triangleq \Psi_{7d} \left( \Gamma^\alpha_{(a,b)}, \Gamma^\beta_{(c,d)} \right) \land \left( \alpha^\beta_a \land \beta^\beta_d \right) \left[ E_x, E_y \right] \land \left( \alpha^\beta_a \land \beta^\beta_d \right) \left[ E_x, E_y \right] \land \left( \alpha^\beta_a \land \beta^\beta_d \right) \left[ E_x, E_y \right] \land

In the construction equation 8.1.3.16 we introduce \( E_w \) and have \( E_w = E^\alpha_a \), introduce \( E_x \) and use it to replace \( E^\alpha_{b+1} \), introduce \( E_y \) and use it to replace \( E^\beta_d \), and introduce \( E_z \) and have \( E_z = E^\beta_d \). If there are the same entry predicates between \( \Gamma^\alpha_{(a,b)} \) and \( \Gamma^\beta_{(c,d)} \) in two \( \Psi_{7d} \) then change them to make them different from each other.

For \( \beta^\beta_c \) and \( \alpha^\beta_b \), if you forget its meaning, please check the definition 7.1 (page 58).
8.1.3.17. $\Psi_{\tau e 2} (T, T, \gamma) \models T$

8.1.3.18. $\Psi_{\tau e 2} (\Gamma_{(a,b)}^\alpha, T, \beta_c^i) \models \Psi_{\tau d} (\Gamma_{(a,b)}^\alpha, \beta_c^i [E_x, E_y])$

8.1.3.19. $\Psi_{\tau e 2} (\Gamma_{(a,b)}^\alpha, \Gamma_{(c,d)}^\beta, \gamma)
\models \Psi_{\tau d} (\Gamma_{(a,b)}^\alpha, \beta_c^i [E_x, E_y]) \land \Psi_{\tau d} (\Gamma_{(c,d)}^\beta, \alpha_b^o [E_y, E_z])$

In the construction $\Psi_{\tau e 2}$ we use boolean constant $T$ to express an empty part chain. In the construction equation 8.1.3.18 we introduce $E_x$ and $E_y$ and have $E_x = E_a^\alpha$ and $E_y = E_{b+1}^\alpha$. In the construction equation 8.1.3.19 we introduce $E_x$, $E_y$ and $E_z$ and have $E_x = E_a^\alpha$, $E_y = E_{b+1}^\alpha = E_c^\beta$ and $E_z = E_{d+1}^\beta$.

8.1.3.20. $\Psi_{\tau d} (\alpha [E_1, E_2], \beta^{i/o} [E_1, E_2]) \models (\alpha \land \beta^{i/o}) [E_1, E_2]$

8.1.3.21. $\Psi_{\tau d} (\alpha_1 [E_1, E_2] \land \Gamma_{(2, n)}^\alpha, \beta^{i/o} [E_1, E_n])
\models (\alpha_1 \land \beta^{i/o}) [E_1, E_2] \land \Psi_{\tau d} (\Gamma_{(2, n)}^\alpha, \beta^{i/o} [E_2, E_n])$

In the construction $\Psi_{\tau d}$ we write $\beta^{i/o}$ to mean is may be $\beta^i$ or $\beta^o$.

Let us explain the construction $7$.

$\Psi_7$ is a construction which transforms a term transition model formed as $(tt s) [E_i, E_o]$ to a linear chain. The work is done through two construction $\Psi_{7s}$ and $\Psi_{7m}$.

$\Psi_{7s}$ is a trivial operation. It separates the term transitions $tt$ in a term transition set $tt s$ through using comma “,” to replace logic conjunctive “$\land$” and then passes the $tt$ with the same logic boundary $[E_i, E_o]$ of the $tt s$ to the construction $\Psi_{7l}$. The work of $\Psi_{7s}$ is done through the construction equations 8.1.3.2 and 3.

**Example 8.1.3.1.** [for understanding $\Psi_{7s}$]:

$\Psi_{7s} ((s_1 y = f (g (r_1 x)) \land s_2 y = h (r_2 x)) [E_i, E_o])$

$= \Psi_{7l} (s_1 y = f (g (r_1 x)) [E_i, E_o]), \Psi_{7l} (s_2 y = h (r_2 x) [E_i, E_o])$ \[\square\]
\( \Psi_{\ell} \) is the main framework of the construction 7. It transforms a term transition model \((tt) [E_i, E_o]\) to a linear chain. Construction equation 8.1.3.4 deals with a base case: connection unit model. It does not change any thing. Construction equation 8.1.3.5 deals with another base case: constant \(tt\). It allocates a device to implement the constant term transition and meanwhile adds two connection links among them the \(T [E_o, E_o]\) is a special one. Construction equation 8.1.3.6 deals with the step case. For a general \(s y = f (tm, \cdots, tm) [E_i, E_o]\) it first recursively calls \(\Psi_7\) to treat all the argument terms of the function \(f\)

\[
\Psi_7 ((i_1 y = tm_1 \land \cdots \land i_n y = tm_n) [E_i, E_a])
\]

and get a linear chain. Then it produces a function unit and a connection units and makes the three parts form a linear chain.

Example 8.1.3.2. [for understanding \(\Psi_{\ell}\)]:

\[
\Psi_{\ell} ((s y = f (r_1 x, r_2 x)) [E_i, E_o]) \\
= \Psi_7 ((i_1 y = r_1 x \land i_2 y = r_2 x) [E_i, E_a]) \land \\
( o y \equiv f (i_1 x, i_2 x)) [E_a, E_b] \land \\
( s y = o x) [E_b, E_o] \quad \text{By construction equation 8.1.3.6}
\]

\[
= (i_1 y = r_1 x \land i_2 y = r_2 x) [E_i, E_a] \land \\
( o y \equiv f (i_1 x, i_2 x)) [E_a, E_b] \land \\
( s y = o x) [E_b, E_o]
\]

Note: later in this construction 7 we will see how to get

\[
(i_1 y = r_1 x \land i_2 y = r_2 x) [E_i, E_a]
\]

from

\[
\Psi_7 ((i_1 y = r_1 x \land i_2 y = r_2 x) [E_i, E_a])
\]

following a general construction process. But here, it can be intuitively accepted as it contains only two connection units and it is obvious that we can put them into a model to implement.

\(\Psi_{\ell m}\) merge linear chains to get a linear chain. These chains have the same head
and tail nodes (see page 87): \( E_i \) and \( E_o \). If there is only a chain to be merged then
the chain has been the merged one. This is as the construction equation 8.1.3.7.
If there are two chains to be merged then, as the construction equation 8.1.3.8 shown,
the merging is passed to \( \Psi_f \) to do. If there are more than two chains need
to be merged then, as the construction equation 8.1.3.9 shown, this is recursively
done through calling \( \Psi_f \).

**Example 8.1.3.3. [for understanding \( \Psi_{7m} \):]**

\( \Psi_{7m} \) can be understood through the process from (c1) and (c2) to (d1), (d2),
..., (d5) in the example in page 83 and page 84.

\( \Psi_{7f} \) merge two linear chains to get a linear chain. There are two cases: base
one and step one. The base case is that there is not any two function links coming
from the two chains respectively being merge-able. In this case the two chains are
passed to \( \Psi_{7c} \) to treat. This is as the construction equation 8.1.3.10. The step
case is that there is at least a pair of function units coming from the two chains
respectively being merge-able. In this case, as the construction equation 8.1.3.11
shown, we can arbitrarily select a pair of merge-able function units to merge them
to get a model

\[
(\alpha_i^F \land \beta_j^F) [E_x, E_y]
\]

and leave four sub-chains

\[
\Gamma_{(0,i-1)}^\alpha, \Gamma_{(0,j-1)}^\beta, \Gamma_{(i+1,m)}^\alpha \text{ and } \Gamma_{(j+1,n)}^\beta
\]

to be recursively dealt with by \( \Psi_{7f} \) itself:

\[
\Psi_{7f} (\Gamma_{(0,i-1)}^\alpha, \Gamma_{(0,j-1)}^\beta) \text{ and } \Psi_{7f} (\Gamma_{(i+1,m)}^\alpha, \Gamma_{(j+1,n)}^\beta)
\]

We like to point that the two chains \( \Gamma_1 \) and \( \Gamma_2 \) in the construction equation 8.1.3.10
certainly are non-empty by the structure of linear chain.

**Example 8.1.3.4. [for understanding \( \Psi_{7f} \):]**

The example is the same as the example 8.1.3.3. Two places should be pointed
out to help reader to understand.
1. Two function links

\((\alpha_2 y = f (i_2 x)) [E_3, E_4]\)

\((\alpha y = h (i x)) [E_a, E_b]\)

are merged to get a tolerant function link model as (d4)

2. Other links are merged to get two sub-chains:

   (1). (d1) \& (d2) \& (d3)

   (2). (d5)

Construction \(\Psi_{\tau e}\) deals with two linear chains which are passed from \(\Psi_{\tau f}\), therefore, have not function links merge-able. If the two chains further have not two connection links merge-able then they are passed \(\Psi_{\tau e1}\) to treat. This is as the construction equation 8.1.3.12 shown. Otherwise, as the construction equation 8.1.3.13 shown, we find the pair of merge-able connection links which is the first one in the two chains along the direction from left to right:

\[\alpha^C_i [E^\alpha_i, E^\alpha_{i+1}] \text{ and } \beta^C_j [E^\beta_j, E^\beta_{j+1}]\]

to merge them to get \((\alpha^C_i \& \beta^C_j) [E_x, E_y]\)

We pass the two left parts of the two linear chains to the construction \(\Psi_{\tau e2}\) and the two right parts of the two linear chains to the construction \(\Psi_{\tau e'}\). However, it is possible that the one left part or two left parts are empty chains, in this case we write boolean constant T to express it.

For the case of an empty chain we use \(\gamma\) for the input signal protection (see 58) of the corresponding merged connection link. And as the \(\Psi_{\tau e2}\) is commutative we can swap its first two arguments to make sure that in the case of there are empty chains the second argument certainly is the empty one (This is for the construction \(\Psi_{\tau e2}\) easier to be treated). Similarly we deal with the two right part.

**Example 8.1.3.5. [for understanding \(\Psi_{\tau e}\):**

\[\Psi_{\tau e} ((s_1 y = r_1 x) [E_1, E_2] \& \alpha^F [E_2, E_3] \& (s_2 y = r_2 x) [E_3, E_4],\]
\[ (s_3 \, y = r_3 \, x) \, [E_1, \, E_4] \]
\[ = \Psi_{\tau e_2} (T, \, T, \, \gamma) \land \]
\[ (s_1 \, y = r_1 \, x \land s_3 \, y = r_3 \, x) \, [E_1, \, E_2] \land \]
\[ \Psi_{\tau c} (\alpha^\vee [E_2, \, E_3] \land (s_2 \, y = r_2 \, x) \, [E_3, \, E_4], \, T, \, s_3 \, y = s_3 \, x) \]

By the construction equation 8.1.3.13

\[ = T \land \]
\[ (s_1 \, y = r_1 \, x \land s_3 \, y = r_3 \, x) \, [E_1, \, E_2] \land \]
\[ (\alpha^\vee \land s_3 \, y = s_3 \, x) \, [E_2, \, E_3] \land (s_2 \, y = r_2 \, x \land s_3 \, y = s_3 \, x) \, [E_3, \, E_4]) \]

Note: the \( \Psi_{\tau e_2} \) and \( \Psi_{\tau c} \) later will make the construction, however here, we suppose that they are intuitively acceptable by reader.

\[ = (s_1 \, y = r_1 \, x \land s_3 \, y = r_3 \, x) \, [E_1, \, E_2] \land \]
\[ (\alpha^\vee \land s_3 \, y = s_3 \, x) \, [E_2, \, E_3] \land \]
\[ (s_2 \, y = r_2 \, x \land s_3 \, y = s_3 \, x) \, [E_3, \, E_4]) \]

This is a linear chain. \( \Box \)

Construction \( \Psi_{\tau c} \) is similar to \( \Psi_{\tau c} \). However, a difference is that the two left parts \( \Gamma_{(0, j-1)}^\alpha \) and \( \Gamma_{(0, j-1)}^\beta \) in the construction equation 8.1.3.15 certainly are non-empty. This is because that \( \Psi_{\tau c} \) is called by \( \Psi_{\tau c} \) and by the structure of linear chain between two connection links there is at least a function link.

**Example 8.1.3.6.** [for understanding \( \Psi_{\tau c} \)]:

\[ \Psi_{\tau c'} ((\alpha^\vee \land \beta_1^\vee) \, [E_1, \, E_2] \land (s_1 \, y = r_1 \, x) \, [E_2, \, E_3], \]
\[ (\alpha^\vee \land \beta_2^\vee) \, [E_1, \, E_2] \land (s_2 \, y = r_2 \, x) \, [E_2, \, E_3], \, \gamma) \]
\[ = \Psi_{\tau e_1} ((\alpha^\vee \land \beta_1^\vee) \, [E_1, \, E_2], \, (\alpha^\vee \land \beta_2^\vee) \, [E_1, \, E_2]) \land \]
\[ (s_1 \, y = r_1 \, x \land s_2 \, y = r_2 \, x) \, [E_2, \, E_3] \land \]
\[ \Psi_{\tau c'} (T, \, T, \, \gamma') \]

By construction equation 8.1.3.15

\[ = \Psi_{\tau e_1} ((\alpha^\vee \land \beta_1^\vee) \, [E_1, \, E_2], \, (\alpha^\vee \land \beta_2^\vee) \, [E_1, \, E_2]) \land \]
\[ (s_1 \, y = r_1 \, x \land s_2 \, y = r_2 \, x) \, [E_2, \, E_3] \]
Note: the $\Psi_{T_{E_2}}$ later will make that $\Psi_{T_{E_2}} (T, T', \gamma') = T$, however here, we suppose that they are intuitively acceptable by reader.

Construction $\Psi_{T_{E_1}}$ deals with two non-empty chains (if it is called by the construction equation 8.1.3.12) or non-empty parts of chains (if it is called by the construction equation 8.1.3.15) whose head and tail link are function link. And the two chain or two parts of chains have all links (function links and connection links) non-merge-able. In the case we have to put the two parts into two different time intervals and to introduce a new link to connect the two parts. The new link is a special one as it only contains delay units (see the definition 8.1.2.6, page 86).

**Example 8.1.3.7. [for understanding $\Psi_{T_{E_1}}$]:**

\[
\Psi_{T_{E_1}} (\alpha_1^f [E_1, E_2] \land \alpha_2^c [E_2, E_3] \land \alpha_3^f [E_3, E_4], \\
\beta_1^f [E_1, E_2] \land \beta_2^c [E_2, E_3] \land \beta_3^f [E_3, E_4]) \\
= \Psi_{T_{d}} (\alpha_1^f [E_1, E_a] \land \alpha_2^c [E_a, E_b] \land \alpha_3^f [E_b, E_c], \ (\beta_1^f)^{i} [E_1, E_c]) \land \\
\quad ((\alpha_3^c)^{o} \land (\beta_1^f)^{i}) [E_c, E_d] \land \\
\Psi_{T_{d}} (\beta_1^f [E_d, E_e] \land \beta_2^c [E_e, E_f] \land \beta_3^f [E_f, E_4], \ (\alpha_3^c)^{o} [E_d, E_4])
\]

Construction $\Psi_{T_{E_2}}$ deals with two part chains whose head and tail links are connection link and function link respectively (if $\Psi_{T_{E_2}}$ is called by the construction equation 8.1.3.13) or function link and connection link respectively (if $\Psi_{T_{E_2}}$ is called by the construction equation 8.1.3.14). The two part chains have not any their links merge-able. One or both of the two parts possibly are empty.

To deal with the empty part chain we introduce the third argument in $\Psi_{T_{E_2}}$ to record signal protections which is passed from the construction equation 8.1.3.13 or 14. And this is used for the construction equation 8.1.3.18. The construction equation 8.1.3.17 deals with the case of that two part chains are empty. In this case, obviously, the result is also an empty part chain and the third argument is useless. The construction equation 8.1.3.19 deals with the case of that two part chains are non-empty. This case is similar to $\Psi_{T_{E_1}}$ but needs not to introduce a special link. And in this case the third argument is too useless.
Example 8.1.3.8. [for understanding $\Psi_{7e_2}$]:

$$
\Psi_{7e_1} (\alpha_{1}^{E_1} [E_1, E_2] \land \alpha_{2}^{E_2} [E_2, E_3] \land \alpha_{3}^{E_3} [E_3, E_4] \land \alpha_{4}^{E_4} [E_4, E_5], \\
\beta_{1}^{E_e} [E_a, E_b] \land \beta_{2}^{E_e} [E_b, E_c], \gamma) \\
= \Psi_{7d} (\alpha_{1}^{E_1} [E_1, E_2] \land \alpha_{2}^{E_2} [E_2, E_3] \land \alpha_{3}^{E_3} [E_3, E_4] \land \alpha_{4}^{E_4} [E_4, E_5], (\beta_{1}^{E_e})^t [E_1, E_5]) \\
\land \\
\Psi_{7d} (\beta_{1}^{E_e} [E_5, E_b] \land \beta_{2}^{E_e} [E_b, E_c], (\alpha_{4}^{E_e})^t [E_5, E_c])
$$

Construction $\Psi_{7d}$ divide a delay unit and then distributes it to a chain or a part chain. To the chain the construction equation 8.1.3.20 deals with its base case and the construction equation 8.1.3.21 deals with its step case.

Example 8.1.3.9. [for understanding $\Psi_{7d}$]:

$$
\Psi_{7d} (\alpha_{1}^{E_1} [E_1, E_2] \land \alpha_{2}^{E_2} [E_2, E_3] \land \alpha_{3}^{E_3} [E_3, E_4], \beta_{1}^{t} [E_1, 4]) \\
= (\alpha_{1}^{E_1} \land \beta_{1}^{t}) [E_1, E_2] \land (\alpha_{2}^{E_2} \land \beta_{2}^{t}) [E_2, E_3] \land (\alpha_{3}^{E_3} \land \beta_{3}^{t}) [E_3, E_4]
$$

Finally, let us give a synthetic example. We reconstruction the example (page 82).

Example 8.1.3.10. [synthetic one]:

$$
\Psi_{7} ((s_1 y = f (g (r_1 x)) \land s_2 y = h (r_2 x)) [E_i, E_o]) \text{...............(1)} \\
= \Psi_{7m} (\Psi_{7s} ((s_1 y = f (g (r_1 x)) \land s_2 y = h (r_2 x)) [E_i, E_o])) \text{...............} \\
\text{by the construction equation 8.1.3.1} \\
= \Psi_{7m} (\Psi_{7l} ((s_1 y = f (g (r_1 x))) [E_i, E_o]), \Psi_{7l} ((s_2 y = h (r_2 x)) [E_i, E_o])) \text{...} \\
\text{by the construction equation 8.1.3.2 and 3} \\
\Psi_{7l} ((s_1 y = f (g (r_1 x))) [E_i, E_o]) \text{...............(2)} \\
= \Psi_{7} ((i_2 y = g (r_1 x)) [E_i, E_3]) \land \\
\text{(o_2 y = f (i_2 x)) [E_3, E_4] \land} \\
\text{(s_1 y = o_2 x) [E_4, E_o]. \text{by the construction equation 8.1.3.6}} \\
\Psi_{7} ((i_2 y = g (r_1 x)) [E_i, E_3]) \text{...............(3)} \\
= \Psi_{7m} (\Psi_{7s} ((i_2 y = g (r_1 x)) [E_i, E_3])) \text{...by the construction equation 8.1.3.1} \text{...} \\
= \Psi_{7m} (\Psi_{7l} ((i_2 y = g (r_1 x)) [E_i, E_3])) \text{...by the construction equation 8.1.3.2} \text{...}
\[ \Psi_{\eta_1} ((i_2 y = g (r_1 x)) [E_i, E_3]) \] ........................................... (4)
\[ = \Psi_7 ((i_1 y = r_1 x) [E_i, E_1]) \wedge \]
\[ (o_1 y = g (i_1 x)) [E_1, E_2] \wedge \]
\[ (i_2 y = o_1 x) [E_2, E_3] \text{..............by the construction equation 8.1.3.6} \]
\[ \Psi_7 (i_1 y = r_1 x) [E_i, E_1] \] ........................................... (5)
\[ = \Psi_{\eta_m} (\Psi_{\eta_s} ((i_1 y = r_1 x) [E_i, E_1])) \text{......by the construction equation 8.1.3.1} \]
\[ = \Psi_{\eta_m} (\Psi_{\eta_1} ((i_1 y = r_1 x) [E_i, E_1])) \text{......by the construction equation 8.1.3.2} \]
\[ = \Psi_{\eta_m} ((i_1 y = r_1 x) [E_i, E_1]) \text{......by the construction equation 8.1.3.4} \]
\[ = (i_1 y = r_1 x) [E_i, E_1] \text{...............by the construction equation 8.1.3.7} \]

Going back to (4), we have
\[ \Psi_{\eta_1} ((i_2 y = g (r_1 x)) [E_i, E_3]) \] ........................................... (4’)
\[ = (i_1 y = r_1 x) [E_i, E_1] \wedge \]
\[ (o_1 y = g (i_1 x)) [E_1, E_2] \wedge \]
\[ (i_2 y = o_1 x) [E_2, E_3] \]

Going back to (3), we get
\[ \Psi_7 ((i_2 y = g (r_1 x)) [E_i, E_3]) \] ........................................... (3’)
\[ = \Psi_{\eta_m} (\Psi_{\eta_1} ((i_2 y = g (r_1 x)) [E_i, E_3])) \]
\[ = \Psi_{\eta_m} ((i_1 y = r_1 x) [E_i, E_1] \wedge (o_1 y = g (i_1 x)) [E_1, E_2] \wedge \]
\[ (i_2 y = o_1 x) [E_2, E_3]) \]
\[ = (i_1 y = r_1 x) [E_i, E_1] \wedge \]
\[ (o_1 y = g (i_1 x)) [E_1, E_2] \wedge \]
\[ (i_2 y = o_1 x) [E_2, E_3] \text{..............by the construction equation 8.1.3.7} \]

Going back to (2) we get
\[ \Psi_{\eta_1} ((s_1 y = f (g (r_1 x))) [E_i, E_o]) \] ........................................... (2’)
\[ = (i_1 y = r_1 x) [E_i, E_1] \wedge \]
\[ (o_1 y = g (i_1 x)) [E_1, E_2] \wedge \]
\[ (i_2 y = o_1 x) [E_2, E_3] \wedge \]
(o_2 \ y = f \ (i_2 \ x)) \ [E_3, \ E_4] \land
(s_1 \ y = o_2 \ x) \ [E_4, \ E_o]

Similarly, we get

\[ \Psi_{71} (s_2 \ y = h \ (r_2 \ x)) \ [E_i, \ E_o] \] \hdashline (6)

\[ = (i_3 \ y = r_3 \ x) \ [E_i, \ E_a] \land \]
\[ (o_3 \ y = h \ (i_3 \ x)) \ [E_a, \ E_b] \land \]
\[ (s_2 \ y = o_3 \ x) \ [E_b, \ E_o] \]

Using (2') and (6) and going back to (1) we have

\[ \Psi_7 ((s_1 \ y = f \ (g \ (r_1 \ x)) \land s_2 \ y = h \ (r_2 \ x)) \ [E_i, \ E_o]) \] \hdashline (1')

\[ = \Psi_{7m} ((i_1 \ y = r_1 \ x) \ [E_i, \ E_i] \land \]
\[ (o_1 \ y = g \ (i_1 \ x)) \ [E_1, \ E_2] \land \]
\[ (i_2 \ y = o_1 \ x) \ [E_2, \ E_3] \land \]
\[ (o_2 \ y = f \ (i_2 \ x)) \ [E_3, \ E_4] \land \]
\[ (s_1 \ y = o_2 \ x) \ [E_4, \ E_o], \]
\[ (i_3 \ y = r_2 \ x) \ [E_i, \ E_a] \land \]
\[ (o_3 \ y = h \ (i_3 \ x)) \ [E_a, \ E_b] \land \]
\[ (s_2 \ y = o_3 \ x) \ [E_b, \ E_o] \]

As we select a device to implement both function \( g \) and \( h \) we use \( i \) and \( o \) to replace \( i_1, \ i_3 \) and \( o_1, \ o_3 \) respectively.

\[ \Psi_7 ((s_1 \ y = f \ (g \ (r_1 \ x)) \land s_2 \ y = h \ (r_2 \ x)) \ [E_i, \ E_o]) \] \hdashline (1'')

\[ = \Psi_{7m} ((i \ y = r_1 \ x) \ [E_i, \ E_i] \land \]
\[ (o \ y = g \ (i \ x)) \ [E_1, \ E_2] \land \]
\[ (i_2 \ y = o \ x) \ [E_2, \ E_3] \land \]
\[ (o_2 \ y = f \ (i_2 \ x)) \ [E_3, \ E_4] \land \]
\[ (s_1 \ y = o_2 \ x) \ [E_4, \ E_o], \]
\[ (i \ y = r_2 \ x) \ [E_i, \ E_a] \land \]
\[ (o \ y = h \ (i \ x)) \ [E_a, \ E_b] \land \]
\[ (s_2 \ y = o \ x) \ [E_b, \ E_o] \]

\[ = \Psi_{7f} ((i \ y = r_1 \ x) \ [E_i, \ E_i] \land \]
\[(o \ y = g \ (i \ x)) \ [E_1, \ E_2] \wedge \]
\[(i_2 \ y = o \ x) \ [E_2, \ E_3] \wedge \]
\[(o_2 \ y = f \ (i_2 \ x)) \ [E_3, \ E_4] \wedge \]
\[(s_1 \ y = o_2 \ x) \ [E_4, \ E_0], \]
\[(i \ y = r_2 \ x) \ [E_i, \ E_a], \]
\[(o \ y = h \ (i \ x)) \ [E_a, \ E_b] \wedge \]
\[(s_2 \ y = o \ x) \ [E_b, \ E_o]] \] ......... by the construction equation 8.1.3.8

\[= \Psi_{7f} \ ((i \ y = r_1 \ x) \ [E_i, \ E_1] \wedge (o \ y = g \ (i \ x)) \ [E_1, \ E_2] \wedge (i_2 \ y = o \ x) \ [E_2, \ E_3], \]
\[(i \ y = r_2 \ x) \ [E_i, \ E_3]) \wedge \]
\[(o_2 \ y = f \ (i_2 \ x) \wedge o \ y = h \ (i \ x)) \ [E_3, \ E_4] \wedge \]
\[\Psi_{7f} \ ((s_1 \ y = o_2 \ x) \ [E_4, \ E_0], \ (s_2 \ y = o \ x) \ [E_4, \ E_o]) \] .................

INNER by the construction equation 8.1.3.11

\[\Psi_{7f} \ ((i \ y = r_1 \ x) \ [E_i, \ E_1] \wedge (o \ y = g \ (i \ x)) \ [E_1, \ E_2] \wedge (i_2 \ y = o \ x) \ [E_2, \ E_3], \]
\[(i \ y = r_2 \ x) \ [E_i, \ E_3]) \].......................... (7)

\[= \Psi_{7c} \ ((i \ y = r_1 \ x) \ [E_i, \ E_1] \wedge (o \ y = g \ (i \ x)) \ [E_1, \ E_2] \wedge (i_2 \ y = o_1 \ x) \ [E_2, \ E_3], \]
\[(i \ y = r_2 \ x) \ [E_i, \ E_3]) \] .................... by the construction equation 8.1.3.10

\[= \Psi_{7e2} \ ((i \ y = r_1 \ x) \ [E_i, \ E_1] \wedge (o \ y = g \ (i \ x)) \ [E_1, \ E_2], \ T, \ r_2 \ y = r_2 \ x) \wedge \]
\[(i_2 \ y = o \ x \wedge i \ y = r_2 \ x) \ [E_2, \ E_3] \wedge \]
\[\Psi_{7e2} \ (T, \ T, \ \gamma) \] .................. by the construction equation 8.1.3.13

\[\Psi_{7e2} \ ((i \ y = r_1 \ x) \ [E_i, \ E_1] \wedge (o \ y = g \ (i \ x)) \ [E_1, \ E_2], \ T, \ r_2 \ y = r_2 \ x) \] ......... (8)

\[= \Psi_{7d} \ ((i \ y = r_1 \ x) \ [E_i, \ E_1] \wedge (o \ y = g \ (i \ x)) \ [E_1, \ E_2], \]
\[(r_2 \ y = r_2 \ x) \ [E_i, \ E_2]) \] ............ by the construction equation 8.1.3.18

\[= \ (i \ y = r_1 \ x \wedge r_2 \ y = r_2 \ x) \ [E_i, \ E_1] \wedge \]
\[\ (o \ y = g \ (i \ x) \wedge r_2 \ y = r_2 \ x) \ [E_1, \ E_2] \] .................

........................................ by the construction equation 8.1.3.21 and 20

\[\Psi_{7c'} \ (T, \ T, \ \gamma) \] .................................................. (9)

\[= \Psi_{7e2} \ (T, \ T, \ \gamma) \] .................... by the construction equation 8.1.3.14

\[= \ T \] ........................................ by the construction equation 8.1.3.17
Using (8) and (9) and going back to (7) we get

\[ \Psi_{7f} ((i \ y = r_1 \ x) \ [E_i, \ E_1] \land (o \ y = g (i \ x)) \ [E_1, \ E_2] \land (i_2 \ y = o \ x) \ [E_2, \ E_3], \]

\[ (i \ y = r_2 \ x) \ [E_i, \ E_3]) \] \hfill (7')

\[ = (i \ y = r_1 \ x \land r_2 \ y = r_2 \ x) \ [E_i, \ E_1] \land \]

\[ (o \ y = g (i \ x) \land r_2 \ y = r_2 \ x) \ [E_1, \ E_2] \land \]

\[ (i_2 \ y = o \ x \land i \ y = r_2 \ x) \ [E_2, \ E_3] \]

\[ \Psi_{7f} ((s_1 \ y = o_2 \ x) \ [E_4, \ E_0], \ (s_2 \ y = o \ x) \ [E_4, \ E_0]) \] \hfill (10)

\[ = \Psi_{7c} ((s_1 \ y = o_2 \ x) \ [E_4, \ E_0], \ (s_2 \ y = o \ x) \ [E_4, \ E_0]) \]

\[ \] \hfill by the construction equation 8.1.3.10

\[ = \Psi_{7c_2} (T, \ T, \ \gamma) \land \]

\[ (s_1 \ y = o_2 \ x \land s_2 \ y = o \ x) \ [E_1, \ E_0] \land \]

\[ \Psi_{7c'} (T, \ T, \ \gamma') \] \hfill by the construction equation 8.1.3.13

\[ = (s_1 \ y = o_2 \ x \land s_2 \ y = o \ x) \ [E_4, \ E_0] \]

\[ \] \hfill by the construction equation 8.1.3.17 and 14

Using (7') and (10) and going back to (1'') we get

\[ \Psi_7 ((s_1 \ y = f (g (r_1 \ x)) \land s_2 \ y = h (r_2 \ x)) \ [E_i, \ E_0]) \] \hfill (1'')

\[ = (i \ y = r_1 \ x \land r_2 \ y = r_2 \ x) \ [E_i, \ E_1] \land \]

\[ (o \ y = g (i \ x) \land r_2 \ y = r_2 \ x) \ [E_1, \ E_2] \land \]

\[ (i_2 \ y = o \ x \land i \ y = r_2 \ x) \ [E_2, \ E_3] \land \]

\[ (o_2 \ y = f (i_2 \ x) \land o \ y = h (i \ x)) \ [E_3, \ E_4] \land \]

\[ (s_1 \ y = o_2 \ x \land s_2 \ y = o \ x) \ [E_4, \ E_0] \]

\[ \square \]

8.1.4 Specification Scheme 7

The scheme 7 obtained through the construction 7 has three type structures: function link, connection link and branch structure.

Specification Scheme 7:

\[ E \ 0 \land fcms \]
Function unit and connection

**model set** \( fcms: \)

\[
fcms ::= \ fcm \mid \ fcm \land fcms
\]

\[
fcm ::= (fus) \ [E_i, E_o] \mid (cus) \ [E_i, E_o]
\]

\[
fus ::= \ f \mid \ f u \land fus
\]

\[
cus ::= \ cu \mid \ cu \land cus
\]

\[
f u ::= o \ y \overset{a}{\longrightarrow} f (i_1, \ldots, i_n, x) \mid o \ y \overset{a}{\longrightarrow} c \mid s \ y = s \ x
\]

\[
c u ::= s \ y = r \ x \mid s \ y = s \ x
\]

**Abbreviation:**

\( fcms: \) function unit and connection

\( fcm: \) model set

\( fus: \) function unit set

\( cus: \) connection unit set

\( fu: \) function unit

\( cu: \) connection unit

\( c u: \) delay unit

**Lemma 8.1.4.1.** [different logic boundary and time intervals]:

In the scheme 7 there are not two linear chains which have the same head node or the same time interval.

**Proof:**

By the theorem 7.1 (page 73), the corollary 7.1.1 (page 73) there is not two term transitions which have the same entry predicates or the same time interval. And by the construction 7 a linear chain keeps its head and tail nodes being the same as the entry and leaving predicates of the term transition model \( ttm \) from which the chain is constructed.

**Theorem 8.1.4.1.** [uniqueness of entry time point and time interval of model at link level]:

**Proof:**
By the scheme 7 a model at link level is one among function link, connection link or branch structure. By the lemma 8.1.4.1 we have had that any two linear chains have their head nodes different therefore their entry time points different and their time intervals different. By the construction 7 we further have that any two links have different entry time point and different time intervals. 

\section{Totalness and Correctness of Construction 7}

\textbf{Theorem 8.2.1.} [totalness of construction 7]:
\[\forall : S \in S_{\forall} \cdot \Psi_7 (s) \in S_{\forall}\]

\textbf{Proof:}

The constructions of the construction 7 can be mainly grouped into two classes: \(\Psi_{7l}\) for decomposition and \(\Psi_{7m}, \Psi_{7s}, \Psi_{7f}, \Psi_{7c}, \Psi_{7c'}, \Psi_{7e1}, \Psi_{7e2}\) and \(\Psi_{7d}\) for merging two models to get a model. Through \(\Psi_{7l}\) function unit and connection unit models are introduced and through other constructions we introduce delay unit model and merge the three type models to get function links and connection links. Branch structures are left unchanged by the construction 7. Therefore for every term transition model of the scheme 6' by the construction 7 we certainly get a linear chain which belongs to the scheme 7. 

\textbf{Lemma 8.2.1.} [correctness of \(\Psi_{7m}\)]:

If for each term transition \(tt\) we have a linear chain \(\Psi_{7l} ((tt) [E_i, E_o])\) and
\[\vdash \Psi_{7l} ((tt) [E_i, E_o]) \Rightarrow (tt) [E_i, E_o]\]

then we have
\[\vdash \Psi_{7m} (\Psi_{7l} ((tt_1) [E_i, E_o]), \cdots, \Psi_{7l} ((tt_n) [E_i, E_o]))\]
\[\Rightarrow (tt_1 \land \cdots \land tt_n) [E_i, E_o].\]

\textbf{Proof:}

The lemma assumes that each \(\Psi_{7l} ((tt_k) [E_i, E_o])\) is a linear chain. For \(\Psi_{7m}\) if there is one linear chain argument then, by the construction equation 8.1.3.7,
conclusion is obvious. For the case of two or more chains \( \Psi_{\tau_m} \) recursively reduces each two of them to \( \Psi_{\tau_f} \) to treat. Therefore the construction is reduced how to merge two linear chains to get a new linear chain.

In the construction 7 there is a construction sequence from \( \Psi_{\tau_f} \) to \( \Psi_{\tau_c}, \Psi_{\tau_c'}, \Psi_{\tau_{e1}}, \Psi_{\tau_{e2}} \) and \( \Psi_{\tau_d} \). In the construction sequence each function or connection unit in the two given linear chains is rearranged to form or into a function link or a connection link. In these constructions some entry predicates may be re-named or some new entry predicates can be introduced and some new delay units possibly are introduced. All these constructions finally lead to a new linear chain merged from the two given linear chains.

What are changed are only entry predicate. What are newly introduced are only delay units. All function units and connection units of the two given linear chains are kept unchanged. Therefore, we can get the conclusion of this lemma.

\[ \square \]

**Theorem 8.2.2.** [correctness of construction 7]:

\[ \Psi_7 ((tts) \ [E_i, E_o]) \text{ is a linear chain and} \]
\[ \models \forall \text{tts} : S_\theta. \Psi_7 ((tts) \ [E_i, E_o]) \Rightarrow (tts) \ [E_i, E_o] \]

**Proof:**

For an arbitrarily given term transition model \( (tts) \ [E_i, E_o] \) let us assume that its term transition set \( tts \) contains \( n \) \( (n \geq 1) \) term transitions \( tt \). By the construction \( \Psi_{\tau_s} \) and \( \Psi_{\tau_t} \) we have

\[ \Psi_7 ((tt_1 \land \cdots \land tt_n) \ [E_i, E_o]) \]
\[ = \Psi_{\tau_m} (\Psi_{\tau_1} ((tt_1 \ [E_i, E_o]), \cdots, \Psi_{\tau_n} ((tt_n \ [E_i, E_o]))) \]

For each \( \Psi_{\tau_t} ((tt_k \ [E_i, E_o]) \ (1 \leq k \leq n) \) we can prove it is a linear chain and have

\[ \models \Psi_{\tau_t} ((tt_k \ [E_i, E_o]) \Rightarrow (tt_k) \ [E_i, E_o] \]

By the structure of \( tt_k \) there are three cases:

(1). \( tt_k = (s \ y = r \ x) \): in this case by the construction equation 8.1.3.4 we have
\[ \Psi_{\eta} ((s\ y = r\ x) [E_i, E_o]) = (s\ y = r\ x) [E_i, E_o] \]

The \((s\ y = r\ x) [E_i, E_o]\) is a linear chain and obviously we have

\[ \vdash \Psi_{\eta} ((tt_k) [E_i, E_o]) \Rightarrow (tt_k) [E_i, E_o] \]

(2). \(tt_k = (s\ y = c)\): in this case by the construction equation 8.1.3.5 we have

\[ \Psi_{\eta} ((s\ y = c) [E_i, E_o]) \]
\[ = (s\ y = o\ x) [E_i, E_o] \land (o\ y = c) [E_a, E_o] \land T [E_o, E_o] \]

\("(s\ y = o\ x) [E_i, E_o] \land (o\ y = c) [E_a, E_o] \land T [E_o, E_o]"\) is a linear chain and obviously we have

\[ \vdash \Psi_{\eta} ((tt_k) [E_i, E_o]) \Rightarrow (tt_k) [E_i, E_o] \]

(3). \(tt_k = (s\ y = f\ (tm_1, \ldots, tm_m))\): in this case by the construction equation

8.1.3.6 we have

\[ \Psi_{\eta} ((s\ y = f\ (tm_1, \ldots, tm_m) [E_i, E_o]) \]
\[ = \Psi_{\eta} ((i_1\ y = tm_1 \land \cdots \land i_m\ y = tm_m) [E_i, E_o]) \land \\
(o\ y = f\ (i_1\ x, \ldots, i_m\ x)) [E_a, E_o] \land \\
(s\ y = o\ x) [E_b, E_o] \]

By the induction hypothesis we have that

\[ \Psi_{\eta} ((i_1\ y = tm_1 \land \cdots \land i_m\ y = tm_m) [E_i, E_o]) \]

is a linear chain. Hence,

\[ \Psi_{\eta} ((s\ y = f\ (tm_1, \ldots, tm_m) [E_i, E_o]) \]

also is a linear chain. By the induction hypothesis we also have

\[ \vdash \Psi_{\eta} ((i_1\ y = tm_1 \land \cdots \land i_m\ y = tm_m) [E_i, E_o]) \]
\[ \Rightarrow (i_1\ y = tm_1 \land \cdots \land i_m\ y = tm_m) [E_i, E_o] \]

Hence, we have

\[ \vdash \Psi_{\eta} ((s\ y = f\ (tm_1, \ldots, tm_m) [E_i, E_o]) \]
\[ \Rightarrow \\
(i_1\ y = tm_1 \land \cdots \land i_m\ y = tm_m) [E_i, E_o] \land \\
(o\ y = f\ (i_1\ x, \ldots, i_m\ x)) [E_a, E_b] \land \]
Chapter 8.  Decomposing to Function Unit and Connection

\[(s \; y = o \; x) \; [E_b, \; E_o]\]
\[\Rightarrow\]
\[(s \; y = f \; (tm_1, \cdots, tm_m) \; [E_i, \; E_o]\]

Therefore, in this case we still have
\[\vdash \Psi_{7i} ((tt_k) \; [E_i, \; E_o]) \Rightarrow (tt_k) \; [E_i, \; E_o]\]

By the lemma 8.2.1 we get
\[\vdash \Psi_{7m} (\Psi_{7i} ((tt_1) \; [E_i, \; E_o]), \cdots, \Psi_{7i} ((tt_n) \; [E_i, \; E_o]))\]
\[\Rightarrow (tt_1 \land \cdots \land tt_n) \; [E_i, \; E_o].\]

Finally, we get
\[\vdash \Psi_7 ((tt_1 \land \cdots \land tt_n) \; [E_i, \; E_o]) \Rightarrow (tt_1 \land \cdots \land tt_n) \; [E_i, \; E_o]\] \qed
Chapter 9

Introducing Register, Bus and Microinstruction

In this chapter we deal with the linear chain of the scheme 7. We discuss why and how to introduce bus and register. And when and where we need to introduce clock and microprogram. As usually we start with an example to help reader to catch the basic ideal. Let us continue the construction of the example of the last chapter (page 82). In the example we have got (page 84):

\[(i \ y = r_1 \ x \land r_2 \ y = r_2 \ x) \ [E_i, \ E_1] \] \hspace{1cm} (d1)

\[\land\]

\[(o \ y = g \ (i \ x) \land r_2 \ y = r_2 \ x) \ [E_1, \ E_2] \] \hspace{1cm} (d2)

\[\land\]

\[(i_2 \ y = o \ x \land i \ y = r_2 \ x) \ [E_2, \ E_3] \] \hspace{1cm} (d3)

\[\land\]

\[(o_2 \ y = f \ (i_2 \ x) \land o \ y = h \ (i \ x)) \ [E_3, \ E_4] \] \hspace{1cm} (d4)

\[\land\]

\[(s_1 \ y = o_2 \ x \land s_2 \ y = o \ x) \ [E_4, \ E_o] \] \hspace{1cm} (d5)
We can see that in the linear chain there are two connections:

\[ i \ y = r_1 \ x \quad \text{of} \quad (d1) \]
\[ i \ y = r_2 \ x \quad \text{of} \quad (d3) \]

that have the same output \( i \). For this we need to introduce a bus, therefore, we need to change them to

\[ i \ y = r_1 \ y \land r_1 \ y = r_1 \ x \]
\[ i \ y = r_2 \ y \land r_2 \ y = r_2 \ x \]

and introduce a constraint \( y > x \) to add to (d2) to guarantee that the two connections be separated. We get

\[
(i \ y = r_1 \ y \land r_1 \ y = r_1 \ x \land r_2 \ y = r_2 \ x) \ [E_i, E_1] \quad (e1)
\]
\[
\land
\]
\[
(o \ y = g \ (i \ x) \land r_2 \ y = r_2 \ x \land y > x) \ [E_1, E_2] \quad (e2)
\]
\[
\land
\]
\[
(i_2 \ y = o \ x \land i \ y = r_2 \ y \land r_2 \ y = r_2 \ x) \ [E_2, E_3] \quad (e3)
\]
\[
\land
\]
\[
(o_2 \ y = f \ (i_2 \ x) \land o \ y = h \ (i \ x)) \ [E_3, E_4] \quad (e4)
\]
\[
\land
\]
\[
(s_1 \ y = o_2 \ x \land s_2 \ y = o \ x) \ [E_4, E_o] \quad (e5)
\]

In the next step we eliminate unnecessary connections. For (e1) use \( x \) to replace \( y \) and \( E_i \) to replace \( E_1 \) to get (f1); for (e2) we use \( s \ y = r_2 \ x \) to replace \( r_2 \ y = r_2 \ x \), \( s_2 \) to replace \( o \) and \( E_i \) to replace \( E_1 \) to get (f2); for (e3) use \( x \) to replace \( y \), \( s \) to replace \( r_2 \), \( s_2 \) to replace \( o \) and \( i_2 \), and \( E_2 \) to replace \( E_3 \) to get (f3); for (e4) use \( s_1 \) to replace \( o_2 \), \( s_2 \) to replace \( o \) and \( i_2 \), and finally, \( E_2 \) and \( E_o \) to replace \( E_3 \) and \( E_4 \) respectively to get (f4); for (e5) use \( x \) to replace \( y \), \( s_1 \) to replace \( o_2 \), \( s_2 \) to replace
\[ o \text{ and } E_o \text{ to replace } E_4 \text{ to get (f5).} \]

\[
\begin{align*}
  (i \ x = r_1 \ x) \ [E_i, E_i] \quad &\quad \text{(f1)} \\
  \land \\
  (s_2 \ y = g (i \ x) \land s \ y = r_2 \ x \land y > x) \ [E_i, E_2] \quad &\quad \text{(f2)} \\
  \land \\
  (i \ x = s \ x) \ [E_2, E_2] \quad &\quad \text{(f3)} \\
  \land \\
  (s_1 \ y = f (s_2 \ x) \land s_2 \ y = h (i \ x)) \ [E_2, E_o] \quad &\quad \text{(f4)} \\
  \land \\
  (s_1 \ x = s_1 \ x \land s_2 \ x = s_2 \ x) \ [E_o, E_o] \quad &\quad \text{(f5)}
\end{align*}
\]

To suit the realistic device group of gate-bus for (f1) and (f3) we use

\[
i \ x = r'_1 \ x \land r'_1 \ x = r_1 \ x
\]
\[
i \ x = s' \ x \land s' \ x = s \ x
\]

to replace

\[
i \ x = r_1 \ x
\]
\[
i \ x = s \ x
\]

Then merge (f1) into (f2), (f3) into (f4). Finally, eliminate (f5) we get:

\[
\begin{align*}
  (s_2 \ y = g (i \ x) \land i \ x = r'_1 \ x \land r'_1 \ x = r_1 \ x \land s \ y = r_2 \ x \land y > x) \ [E_i, E_2] \quad &\quad \text{(g1)} \\
  \land \\
  (s_1 \ y = f (s_2 \ x) \land s_2 \ y = h (i \ x)) \land i \ x = s' \ x \land s' \ x = s \ x) \ [E_2, E_o] \quad &\quad \text{(g2)}
\end{align*}
\]
To get an implementation we need to introduce devices:

\[ D_{gh} (c_g, c_h, i, s_2) \equiv \forall t. s_2 (t + 1) = (c_g \ t \Rightarrow g \ (i \ t) \downarrow (c_h \ t \Rightarrow h \ (i \ t) \downarrow i \ t)) \]

\[ D_f (c_f, s_2, s_1) \equiv \forall t. s_1 (t + 1) = (c_f \ t \Rightarrow f \ (s_2 \ t) \downarrow s_2 \ t) \]

\[ \text{Reg} \ (c_r, r_2, s) \equiv \forall t. s (t + 1) = (c_r \ t \Rightarrow r_2 \ t \downarrow s \ t) \]

\[ \text{Gate} \ (c_{b1}, r_1, r'_1) \equiv \forall t. r'_1 \ t = (c_{b1} \ t \Rightarrow \text{Mk} \_\text{Tri} \ (r_1 \ t) \downarrow \text{Float}) \]

\[ \text{Gate} \ (c_{b2}, s, s') \equiv \forall t. s' \ t = (c_{b2} \ t \Rightarrow \text{Mk} \_\text{Tri} \ (s \ t) \downarrow \text{Float}) \]

\[ \text{Bus} \ (r'_1, s', i) \equiv \forall t. i \ t = \text{Dest} \_\text{Tri} \ (r'_1 \ t \cup s' \ t) \]

We can get the piece of an abstract microprogram, its two microinstruction:

\[ (\forall x. E_1 x \Rightarrow c_{b1} \ x \wedge c_g \ x \wedge c_r \ x \wedge E_2 \ (x + 1)) \]

\[ \land \]

\[ (\forall x. E_2 x \Rightarrow c_{b2} \ x \wedge c_h \ x \wedge c_f \ x \wedge E_o \ (x + 1)) \]

and a datapath:

\[ \text{Reg} \ (c_r, r_2, s) \land \text{Gate} \ (c_{b1}, r_1, r'_1) \land \text{Gate} \ (c_{b2}, s, s') \land \text{Bus} \ (r'_1, s', i) \land \]

\[ D_{gh} (c_g, c_h, i, s_2) \land D_f (c_f, s_2, s_1) \]

We can further instantiate the piece of abstract microprogram to a concrete microprogram, for example:

\[ (\forall x. \text{address} \ x = 11 \Rightarrow c_{b1} \ x \wedge c_g \ x \wedge c_r \ x \wedge \text{address} \ (x + 1) = 12) \]

\[ \land \]

\[ (\forall x. \text{address} \ x = 12 \Rightarrow c_{b2} \ x \wedge c_h \ x \wedge c_f \ x \wedge \text{address} \ (x + 1) = 13) \]

Finally, we get an implementation:

**Implementation 1:**
In summary, starting from the specification 1 (page 82), we follow a nearly mechanical process to derive an implementation as above. In the process two important selections are allocation and scheduling. By allocation we mean that we select a device $D_{gh}$ to implement two function units $g$ and $h$ and another device for $f$ and by scheduling we mean that we put $f$ and $h$ into the same time interval to implement. Mainly, the selection influence and lead to the implementation: its microprogram and datapath.

Let us introduce other two different selections which lead to quite different implementations.

If we use a device $D_{fgh}$ to implement the three functions $f$, $g$ and $h$ then we will derive a piece of microprogram and a datapath as below. (Note: in microprogram we always omit the negative control signal, for example, the second microinstruction includes a $\neg(c_{r1} x)$.)

$$\forall x. \text{address } x = 5 \Rightarrow c_{b1} x \land c_g x \land c_{r1} x \land \text{address } (x + 1) = 6$$

$$\land$$

$$\forall x. \text{address } x = 6 \Rightarrow c_{b3} x \land c_f x \land \text{address } (x + 1) = 7$$

$$\land$$

$$\forall x. \text{address } x = 7 \Rightarrow c_{r2} x \land c_{b2} x \land c_h x \land \text{address } (x + 1) = 8$$

**Implementation 2:**
If we use three devices for three function unit, namely a device is used for a function unit, then no bus is needed and we have a clearer microprogram and datapath as below.

\[(\forall x. \text{address } x = 3 \Rightarrow c_g x \land c_h x \land \text{address } (x + 1) = 4) \land \]

\[(\forall x. \text{address } x = 4 \Rightarrow c_f x \land c_r x \land \text{address } (x + 1) = 5)\]

**Implementation 3:**
9.1 Introducing Bus, Register and Connection

For a specification at the scheme 7 level we present a general and nearly mechanical process to introduce bus, register and connection.

9.1.1 Introducing Bus

A bus is needed if in the linear chains of the scheme 7 there are two or more connection units which have the same output. In the case we need a device which has the same output as its output and all the inputs of these connection units as its inputs. The device is called “bus”.

For any two connection units of a bus if they are in the same linear chain (we need not consider the case that they are in the different linear chains by the theorem 8.1.4.1) then we need to introduce a constraint: that in the all links between the two connection links, which contain the two connection units respectively, there is at least a link which is added with a time constraint:

\[ x < y \]

A skill is that we often select to put the time constraint to the function link as, usually, a function unit needs a time interval to be executed, therefore, the constraint is automatically satisfied.

We need to introduce the constraint otherwise a danger inconsistency can be raised. For example, two connection units in two different connection links have the same output:

\[ s \cdot y = r_1 \cdot x \]
\[ s \cdot y = r_2 \cdot x \]

but, if without the time constraint then it is quite possible that they are finally merged into a link whose time interval is zero \((y)\) is replaced by \(x\) (see the definition
8.1.2.2 (page 86)) and whose specification body contains an inconsistency:

\[ s \ x = r_1 \ x \land s \ x = r_2 \ x \]

If in the scheme 7 there are not two connection units having the same output then the device bus will be unnecessary. However, this means that we have enough devices such that each function unit is assigned to a unique device to implement it (Note: even the same function unit but if they are at different position in the scheme 7 then they are implemented by difference devices).

Here, we need to distinguish two different concepts: device kind and device which should be more exactly called as device instance. Let us use an example. When we write

\[ \text{Reg} (c, i, o) \equiv \forall t. o (t + 1) = c \ t \Rightarrow i \ t \downarrow o \ t \]

we mean a device kind, the register kind. Its ports \( c \), \( i \) and \( o \) are free-variables. When we use the concrete constants to instantiate the free-variable ports of a device kind we get a device (device instance). For example in the implementation 1 (page 113) we introduce a register instance

\[ \text{Reg} (c_r, r_2, s) \]

where, the free-variables \( c \), \( i \) and \( o \) are replaced by \( c_r \), \( r_2 \), \( s \). So, two register instances if they have different ports then are viewed as different registers.

Let us introduce definitions to exactly describe bus.

**Definition 9.1.1.1. [exclusive connection units]:**

Two connection units are exclusive from each other if they are in the different linear chains or in the same chain but having had the time constraint that:

In the all links between the two connection links, which contain the two connection units respectively, there is at least a link which is added with a time constraint: \( x < y \)

**Definition 9.1.1.2. [minimal bus]:**

In the scheme 7 if two or more connection units having the same output and different inputs and any two of them are exclusive from each other then we put
these connection units together and use a device to implement them. We call the
device the minimal bus which has the same output as its output and all the inputs
of these connection units as its inputs. \[ \square \]

We read “minimal bus” as that those connection units must be collected to-
gether (through the exclusion constraint (definition 9.1.1.1)) to form a device and
we cannot separate them (this will possibly lead to an inconsistency). However,
we can artificially further put two or more minimal bus and other connection units
together to form a bus. In such case we call it the compound bus.

**Definition 9.1.1.3. [compound bus]:**

In the scheme 7 for two or more minimal buses or connection units which do
not belong to minimal bus we can put all their connection units together to form
a compound bus if we meanwhile constrain that any two connection units to be
exclusive from each other. \[ \square \]

However, in the case of introducing a compound bus a trouble is that we
need unify the outputs of all the connection units of the compound bus to get
one output as the output of the compound bus. And we need to introduce new
connection unit to connect the output of the compound bus to the original port
which is unified to the output of the compound bus. For this we must go back to
the beginning of the construction 7, in fact, in such a case we need to make the
selection for a compound bus at the beginning of the construction 7.

In the real world, for example following the way [50] (page 10 and 11), to
implement a bus we need to introduce two kind devices:

- **Gate** \((c, i, o) \equiv \forall t. o t = (c t \Rightarrow \text{Mk.Tri} (i t) \downarrow \text{Float})\) ................. (gs)
- **Bus** \((i_1, i_2, \ldots, i_n, o) \equiv \forall t. o t = \text{Dest.Tri} (i_1 t \cup i_2 t \cup \ldots \cup i_n t)\) ........... (bs)

and the two axioms that

\(\vdash \forall s. \text{Dest.Tri} (\text{Mk.Tri} s) = s \quad (s \text{ is a word})\) .......................(dmt)

\(\vdash \forall s. (\text{Float} \cup s = s) \land (s \cup \text{Float} = s) \quad (s \text{ is a tri.state})\) ................ (bo)

(see page 9 [50] for details)
For this we reconstruct each connection unit in a bus:

\[ s \ y = r \ x \]

to three parts:

\[ s \ y = r' \ y \] \hspace{1cm} (c1) \\
\land \\
\[ r' \ y = r \ y \] \hspace{1cm} (c2) \\
\land \\
\[ r \ y = r \ x \] \hspace{1cm} (c3)

Here, (c1) and (c2) are executed by a Bus and a Gate as above respectively.

Generally, for a case of bus formed as

\[ \cdots \land (\Gamma_{1l} \land s \ y = r_1 \ x \land \Gamma_{1r}) \ [E_a, E_b] \land \]
\[ \cdots \land (\Gamma_{2l} \land s \ y = r_2 \ x \land \Gamma_{2r}) \ [E_c, E_d] \land \]
\[ \cdots \land (\Gamma_{nl} \land s \ y = r_n \ x \land \Gamma_{nr}) \ [E_e, E_f] \land \cdots \]

Here, \( \Gamma_{il} \) and \( \Gamma_{ir} \) (1 \( \leq i \leq n \)) represent the conjunction of zero or more specification entities.

We first transform it to

\[ \cdots \land (\Gamma_{1l} \land s \ y = r'_1 \ y \land r'_1 \ y = r_1 \ y \land r_1 \ y = r_1 \ x \land \Gamma_{1r}) \ [E_a, E_b] \land \]
\[ \cdots \land (\Gamma_{2l} \land s \ y = r'_2 \ y \land r'_2 \ y = r_2 \ y \land r_2 \ y = r_2 \ x \land \Gamma_{2r}) \ [E_c, E_d] \land \]
\[ \cdots \land (\Gamma_{nl} \land s \ y = r'_n \ y \land r'_n \ y = r_n \ y \land r_n \ y = r_n \ x \land \Gamma_{nr}) \ [E_e, E_f] \land \cdots \]

Then we have an implementation formed as

\[ \cdots \land (\Gamma_{1l} \land c_{g1} \ y \land r_1 \ y = r_1 \ x \land \Gamma_{1r}) \ [E_a, E_b] \land \]
\[ \cdots \land (\Gamma_{2l} \land c_{g2} \ y \land r_2 \ y = r_2 \ x \land \Gamma_{2r}) \ [E_c, E_d] \land \]
\[ \cdots \land (\Gamma_{nl} \land c_{gn} \ y \land r_n \ y = r_n \ x \land \Gamma_{nr}) \ [E_e, E_f] \land \]
\[ \cdots \land \text{Gate}_1 \ (c_{g1} \ r_1, \ r'_1) \land \text{Gate}_2 \ (c_{g2} \ r_2, \ r'_2) \land \cdots \land \text{Gate}_n \ (c_{gn} \ r_n, \ r'_n) \land \]
\[ \cdots \land \text{Bus} \ (r'_1, r'_2, \cdots, r'_n, s) \land \cdots \]
9.1.2 Introducing Register

Similar to the adding the time constraint for introducing bus we also need to add time constraint into the linear chain to avoid those possible inconsistencies incurred by using the same device to implement two or more function units. (In the case it is possible that these function units have the same output).

**Definition 9.1.2.1. [exclusive function units]:**

Two function units are exclusive from each other if they are in the different linear chains or in the same chain but having had the time constraint that:

In the all links between the two function links, which contain the two function units respectively, there is at least a link which is added with a time constraint:

\[ x < y \]

With the definition we need to put the time constraint into a linear chain if it has two or more exclusive function units.

The third case where we need to introduce a time constraint \(x < y\) is to satisfy the requirement from the convenience 5.3.1.1 (page 33).

A possible case is that a function unit, for example,

\[ o \left( x + 1 \right) = f \left( i_1 x, i_2 x \right) \]

automatically imply a time constraint

\[ x < x + 1. \]

**Check-for-Register 9.1.2.1:**

In the case that in a connection link there are a connection unit with the time constraint:

\[ s y = r x \land x < y \]

The connection unit is selected to be implemented by a register. □

We introduce a register \(\text{Reg} \left( c, r, s \right)\) whose kind form is:

\[ \text{Reg} \left( c, i, o \right) \equiv \forall t. o \left( t + 1 \right) = \left( c t \Rightarrow i t \downarrow o t \right) \]
Then a connection link formed as
\[(\Gamma_l \land s \, y = r \, x \land x < y \land \Gamma_r) \, [E_l, E_o]\]
Here, we use \(\Gamma_l\) and \(\Gamma_r\) to represent the conjunction of zero or more specification entities.

will lead to an implementation formed as:
\[(\Gamma_l \land c \, x \land x < y \land \Gamma_r) \, [E_l, E_o] \land \text{Reg} (c, r, s)\]
However, a special case is that if in a model formed as
\[(s \, y = r \, x) \, [E_l, E_o]\]
\(s\) has been established as the output of a register then we decompose the model to two models:
\[(i \, y = r \, x) \, [E_l, E_m]\]
\[(s \, y = i \, x) \, [E_m, E_o]\]
Here, \(i\) is the input of the register. **Check-and-Change-the-Chain-Containing-Delay-Sequence 9.1.2.2:**

In the case that there is a chain or part chain which has form:
\[(\Gamma_{1l} \land r \, y = r \, x \land x < y \land \Gamma_{1r}) \, [E_1, E_2], \ldots, [E_n, E_{n+1}]\](cc1)

\(\land\)
\[(\Gamma_{2l} \land r \, y = r \, x \land \Gamma_{2r}) \, [E_2, E_3] \land \ldots \land\]
\[(\Gamma_{n \land r} \land r \, y = r \, x \land \Gamma_{nr}) \, [E_n, E_{n+1}]\]
\((n \geq 1)\)

Here, \(\Gamma_{il}\) and \(\Gamma_{ir}\) \((1 \leq i \leq n)\) are the conjunction of zero or more specification entities. (cc1) is the first connection link in the chain, which contains a time constraint \(x < y\).

we change it to
\[(\Gamma_{1l} \land s \, y = r \, x \land x < y \land \Gamma_{1r}) \, [E_1, E_2] \land\]
\[(\Gamma_{2l} \land s \, y = s \, x \land \Gamma_{2r}) \, [E_2, E_3] \land \ldots \land\]
\[(\Gamma_{nl} \land s \, y = s \, x \land \Gamma_{nr}) \, [E_n, E_{n+1}]\]
and use the substitution \([s/r]\) to the link which has the \(r\).

We need the change as we have to use register to implement these delays. In the case it can be implemented by

\[
(\Gamma_{il} \land c \land \Gamma_{lr}) [E_1, E_2] \land \\
(\Gamma_{2i} \land s \land y = s \land x \land \Gamma_{2r}) [E_2, E_3] \land \cdots \land \\
(\Gamma_{ni} \land s \land y = s \land x \land \Gamma_{nr}) [E_n, E_{n+1}] \land \\
\text{Reg} (c, r, s)
\]

A special case if the \(r\) has been established as the output of a register then we need to do nothing.

### 9.1.3 Introducing Connection

For a connection unit in a connection link formed as

\[
(\Gamma_{l} \land s \land y = r \land x \land \Gamma_{r}) [E_{i}, E_{o}],
\]

here, \(\Gamma_{l}\) and \(\Gamma_{r}\) represent the conjunction of zero or more connection units and zero or more delay units.

if there is not a time constraint \(x < y\) we can use the time point \(x\) to replace the time point \(y\), select \(E_{i}\) (\(E_{o}\)) to replace \(E_{o}\) (\(E_{i}\)) and select \(r\) (\(s\)) to replace \(s\) (\(r\)).

Therefore we can get, for example,

\[
(\Gamma_{l} \land r \land x = r \land x \land \Gamma_{r}) [E_{i}, E_{i}]
\]

It is reduced to

\[
(\Gamma_{l} \land \Gamma_{r}) [E_{i}, E_{i}]
\]

In this case we need to pass those replacements, which involve the entry predicates and ports used in other specification entities or links, to the relevant parts.

For example for the example above, we use the substitutions below to all other part in the specification at the scheme 7 level.

\([E_{i}/E_{o}, r/s] \]
Similarly, for a delay unit in a connection link formed as

\[(\Gamma_l \land s \ y = s \ x \land \Gamma_r \times E_i, E_o)\]

here, \(\Gamma_l\) and \(\Gamma_r\) represent the conjunction of zero or more connection units and zero or more delay units.

if there is not a time constraint \(x < y\) we can use the time point \(x\) to replace the time point \(y\) and select \(E_i\) \((E_o)\) to replace \(E_o\) \((E_i)\). Therefore we can get, for example,

\[(\Gamma_l \land s \ x = s \ x \land \Gamma_r \times E_i, E_i)\]

It is reduced to

\[(\Gamma_l \land \Gamma_r \times E_i, E_i)\]

In this case we need to pass the replacement about the entry predicates used in other links, to the relevant parts.

For example for the example above, we use the substitutions below to all other part in the specification at the scheme 7 level.

\[[E_i/E_o]\]

For a connection link the two processes above can be repeated until finally we get a model, for example,

\[(T) \times E_i, E_i\]

It is further reduced to the boolean constant true \(T\) which is called the weakest specification.

In some special case, for example, in a function link all the function units can complete their execution in a zero time interval, for example, such formed function unit:

\[o \ x = f \ (i_1 \ x, i_2 \ x)\]

This case is equal to the case of without time constraint \(x < y\). So, we can deal
with all the delay units in the function link as the the delay in the connection link above.

### 9.1.4 Abstract Algorithm 8, Construction 8 and Specification Scheme 8

Let us use abstract algorithm 8 and construction 8 to summarize the processes of the preceding subsections.

The abstract algorithm 8 is used to mark our selections to add the time constraint and the check for bus, register and connection.

**Abstract Algorithm 8.**

\([A_8 : \text{selecting and marking for bus, register and connection}]\):

For a specification at the scheme 7 level

1. Following the definitions 9.1.1.1 and 9.1.2.1 to add time constraint \(x < y\) to suitable link.

2. To mark a connection unit \(s \ y = r \ x\) which belongs to a bus by
   \[s \ y \overset{b}{=} r \ x.\]
   And to add subscript to the “\(b\)” if it is necessary to distinguish different buses.

3. To mark a connection unit if it is checked being suitable for register by the Check-for-Register 9.1.2.1. For \(s \ y = r \ x\) to mark it with
   \[s \ y \overset{r}{=} r \ x.\]
   And to add subscript to the “\(r\)” if it is necessary to distinguish different registers.

4. Following the Check-and-Change-the-Chain-Containing-Delay-Sequence 9.1.2.2 to further mark the connection and delay units like as 3 for the register implementation.
5. Following the subsection 9.1.3 to check and then mark the connection units 
and delay units which are suitable for “device” Connection. For \( s y = r x \)
and \( s y = s x \) to mark them by
\[
\begin{align*}
s y & \overset{c}{=} r x \\
s y & \overset{c}{=} s x
\end{align*}
\]
And then to pass substitutions for entry predicate and port to the whole 
specification.

6. Above process may need to be repeated if later step leads to a new situation
which needs to be treated with earlier steps. For example, a dealing with
connection can lead to introduce a model which needs to be implemented by
a Register.

The construction 8 is applied to the specification at the scheme 7 which has
been treated by the abstract algorithm 8 to produce a specification at the scheme 8
level. The construction 8 only changes the connection unit which has been marked
for bus to two or three parts for the implementations by Bus, Gate and Register.

**Construction 8:**

9.1.4.1. \( \Psi_B (s y \overset{b}{=} r x) \overset{b'}{=} s y \overset{b'}{=} r' y \land r' y \overset{g}{=} r y \)

if the connection link which the \( s y \overset{b}{=} r x \) comes from has not
a constraint \( x < y \)

9.1.4.2. \( \Psi_B (s y \overset{b}{=} r x) \overset{b'}{=} s y \overset{b'}{=} r'' y \land r'' y \overset{g}{=} r' y \land r' y \overset{r}{=} r x \)

if the connection link which the \( s y \overset{b}{=} r x \) comes from has
a constraint \( x < y \)

In the construction 8 we introduce the new marks “b’’” and “g”. We write
\( s y \overset{b'}{=} r y \)
to mark that it will be implemented by a Bus in a Bus-Gate device group (see
page 118). And we write
\[ s \, y \overset{g}{\Rightarrow} r \, y \]

to mark that it will be implemented by a Gate in a Bus-Gate device group.

**Example 9.1.4.1.** [for understanding \( A_8 \) and \( \Psi_8 \)]:

\[
(i \, y = r_1 \, x \wedge r_2 \, y = r_2 \, x) \, [E_i, \, E_i] \quad (d1)
\wedge
\]

\[
(o \, y \overset{a}{\Rightarrow} g \,(i \, x) \wedge r_2 \, y = r_2 \, x) \, [E_1, \, E_2] \quad (d2)
\wedge
\]

\[
(i_2 \, y = o \, x \wedge i \, y = r_2 \, x) \, [E_2, \, E_3] \quad (d3)
\wedge
\]

\[
(o_2 \, y \overset{a}{\Rightarrow} f \,(i_2 \, x) \wedge o \, y \overset{a}{\Rightarrow} h \,(i \, x)) \, [E_3, \, E_4] \quad (d4)
\wedge
\]

\[
(s_1 \, y = o_2 \, x \wedge s_2 \, y = o \, x) \, [E_4, \, E_o] \quad (d5)
\]

\[
= (i \, x \overset{b}{\Rightarrow} r_1 \, x \wedge r_2 \, x \overset{c}{\Rightarrow} r_2 \, x) \, [E_i, \, E_i] \quad (e'1)
\wedge
\]

\[
(o \, y \overset{a}{\Rightarrow} g \,(i \, x) \wedge s \, y \overset{r}{\Rightarrow} r_2 \, x \wedge x < y) \, [E_i, \, E_2] \quad (e'2)
\wedge
\]

\[
(s_2 \, x \overset{c}{\Rightarrow} s_2 \, x \wedge i \, x \overset{b}{\Rightarrow} s \, x) \, [E_2, \, E_2] \quad (e'3)
\wedge
\]

\[
(s_1 \, y \overset{a}{\Rightarrow} f \,(s_2 \, x) \wedge s_2 \, y \overset{a}{\Rightarrow} h \,(i \, x)) \, [E_2, \, E_o] \quad (e'4)
\wedge
\]

\[
(s_1 \, x \overset{c}{\Rightarrow} s_1 \, x \wedge s_2 \, x \overset{c}{\Rightarrow} s_2 \, x) \, [E_o, \, E_o] \quad (e'5)
\]

\[
\text{........................................... by the abstract algorithm 8}
\]

\[
= (i \, x \overset{b'}{\Rightarrow} r'_1 \, x \wedge r'_1 \, x \overset{g}{\Rightarrow} r_1 \, x \wedge r_2 \, x \overset{c}{\Rightarrow} r_2 \, x) \, [E_i, \, E_i] \quad (f'1)
\wedge
\]

\[
(s_2 \, y \overset{a}{\Rightarrow} g \,(i \, x) \wedge s \, y \overset{r}{\Rightarrow} r_2 \, x \wedge x < y) \, [E_i, \, E_2] \quad (f'2)
\wedge
\]

\[
(s_2 \, x \overset{c}{\Rightarrow} s_2 \, x \wedge i \, x \overset{b'}{\Rightarrow} s' \, x \wedge s' \, x \overset{g}{\Rightarrow} s \, x) \, [E_2, \, E_2] \quad (f'3)
\]
\(\wedge\)
\[(s_1 \ y \overset{a}{=} f \ (s_2 \ x) \wedge s_2 \ y \overset{a}{=} h \ (i \ x)) \ [E_2, \ E_o]\]  \hspace{1cm} (f'4)
\(\wedge\)
\[(s_1 \ x \overset{c}{=} s_1 \ x \wedge s_2 \ x \overset{c}{=} s_2 \ x) \ [E_o, \ E_o]\]  \hspace{1cm} (f'5)

\[\text{by the construction 8}\]

A specification at the scheme 7 level passed through \(\mathcal{A}_8\) and \(\Psi_8\) will become a specification at the scheme 8 level as below.

**Specification Scheme 8:**

\(E \ 0 \wedge \ fms\)

**Function unit and register**

<table>
<thead>
<tr>
<th>model set (fms):</th>
<th>Abbreviation:</th>
</tr>
</thead>
<tbody>
<tr>
<td>(fms := \ fms)</td>
<td>(fms : \text{set of } fms)</td>
</tr>
<tr>
<td>(fms \wedge \ fms)</td>
<td>(fms : \text{function and register model})</td>
</tr>
<tr>
<td>(fms)</td>
<td>(fms : \text{function and register model})</td>
</tr>
<tr>
<td>(fms \wedge \ fms)</td>
<td>(fms : \text{set of function units})</td>
</tr>
<tr>
<td>(fus)</td>
<td>(fus : \text{set of function units})</td>
</tr>
<tr>
<td>(fus \wedge \ fus)</td>
<td>(fus : \text{set of function units})</td>
</tr>
<tr>
<td>(rus)</td>
<td>(rus : \text{set of register units})</td>
</tr>
<tr>
<td>(rus \wedge \ rus)</td>
<td>(rus : \text{set of register units})</td>
</tr>
<tr>
<td>(fu)</td>
<td>(fu : \text{function unit})</td>
</tr>
<tr>
<td>(fu \wedge \ fu)</td>
<td>(fu : \text{function unit})</td>
</tr>
<tr>
<td>(ru)</td>
<td>(ru : \text{register units})</td>
</tr>
<tr>
<td>(ru \wedge \ ru)</td>
<td>(ru : \text{register units})</td>
</tr>
<tr>
<td>(o \ y \overset{a}{=} f \ (i_1 \ x, \cdots, i_n \ x))</td>
<td>(f : \text{function unit})</td>
</tr>
<tr>
<td>(o \ y \overset{a}{=} c)</td>
<td>(c : \text{constant function unit})</td>
</tr>
<tr>
<td>(s \ y \overset{a}{=} s \ x)</td>
<td>(s : \text{delay for Register})</td>
</tr>
<tr>
<td>$s y = s x$</td>
<td>delay for Connection</td>
</tr>
<tr>
<td>$ru ::= s y = r y$</td>
<td>connection for Bus</td>
</tr>
<tr>
<td>$s y = r y$</td>
<td>connection for Gate</td>
</tr>
<tr>
<td>$s y = r x$</td>
<td>connection for Register</td>
</tr>
<tr>
<td>$s y = s x$</td>
<td>delay for Register</td>
</tr>
<tr>
<td>$s y = c$</td>
<td>connection for Connection</td>
</tr>
<tr>
<td>$s y = s x$</td>
<td>delay for Connection</td>
</tr>
</tbody>
</table>

Note: in the scheme we write “Connection” for implementation and “connection” for specification.

The character of the specification at the scheme 8 is that it has every piece specification to be directly executable by a device at the register transfer level: function unit, bus, gate and register.

It is obvious that the abstract algorithm 8 and the construction 8 is total for the scheme 7 and correct for scheme 8 and 7.

9.2 Introducing Datapath and Microprogram

In the specification scheme 8 each specification piece has been marked for a corresponding device to implement, for example,

$$s y = r x$$

is marked to be implemented by a register. In this section we further introduce datapath and microprogram.

By the means of the specification scheme we get an abstract parameterized register transfer level. Function unit is kept throughout the whole derivation process of computer as a parameter which we do not further consider its inside structure. What we gain through the specification scheme and formal construction are the relations among these function units. These relations are expressed by
connection link which including two aspects: data relation through specification entities: bus, register and connection and control relation through logic boundary: entry predicate which is further constructed to get microprogram.

To gain a realistic implementation for computer we have selected certain devices at register transfer level from the real world, as we have done. However, we still need to introduce some further constraints.

**Constraint 9.2.1. [variable time unit computer]:**

All the function units in a function link should be implemented in the same time interval. Or at least they can be improved to do so and these improvements are not further discussed.

Under the constraint all the function units in a function link are guaranteed to be completed in the same time interval. However, the different function links can take the different time intervals. If we take the time interval of a function link as a time unit of computer which also has the time intervals of the other function and connection links as its time units then in this case we can get a variable time unit computer.

To get a constant time unit computer we need to introduce further constraint as below.

**Constraint 9.2.2. [constant time unit computer]:**

A computer is of constant time unit if for it the time intervals of all the function and connection links are the same, namely, the execution of its functions and registers can be completed in the same time interval.

Under the constraint all function units and registers at the register transfer level are assumed to be implemented in the same time unit which is traditionally called as the time unit at microinstruction level.

We can construct the implementation of computer under the looser constraint 9.2.1 if there are not Register and Device for function unit which are used in the same function link. Otherwise, we will use the tighter constraint 9.2.2. Of course,
we can consider to introduce some finer constraints. However, this is not essential and is omitted.

9.2.1 Introducing Datapath and Microprogram

The introductions of datapath and microprogram are done simultaneously. When introducing a device we also introduce the corresponding controls if the implementation of the device requires. These devices will constitute the datapath of computer and these controls as codes with the logic boundary (entry and leaving predicates) will constitute the microprogram of computer. For the specification scheme 8 we take the construction steps as below:

(1). To replace
\[ o \ y \overset{\Delta}{=} f \ (i_1 \ x, \ldots, i_n \ x) \]
by
\[ c_f \ x \land D_f \ (\vec{c}, \vec{i}, \vec{o}) \]
Where, \( D_f \) is a device which has the function of \( f \) (and possibly has other functions also). \( \vec{c} \) is the control vector of the device \( D_f \) and \( c_f \) is a member of \( \vec{c} \). Each \( i_j \ (1 \leq j \leq n) \) is a member of \( \vec{i} \) which is the vector of inputs of the device \( D_f \). The \( o \) is a member of \( \vec{o} \) which is a vector of outputs of the device \( D_f \). Further we assume that
\[ \vdash c_f \ x \land D_f \ (\vec{c}, \vec{i}, \vec{o}) \Rightarrow o \ (x + \delta) \overset{\Delta}{=} f \ (i_1 \ x, \ldots, i_n \ x) \]
Here, \( y \) has been replaced by \( x + \delta \) and \( \delta \) is a constant e.g. 1.

For example, for
\[ o \ y \overset{\Delta}{=} ADD \ (i_1 \ x, \ i_2 \ x) \]
we introduce:
\[ add \ x \land ALU \ (add, sub, inc, i_1, i_2, o) \]
Here,
\[ ALU \ (add, sub, inc, i_1, i_2, o) \equiv \forall t. \ o \ (t + 1) = (add \ t \Rightarrow ADD \ (i_1 \ t, \ i_2 \ t)) \downarrow \]
Similarly, we can do for the constant function unit.

(2). To replace
\[ s\ y \overset{b}{=} r_i\ y \quad \text{or} \quad s\ x \overset{b}{=} r_i\ x \]
By
Bus \( \overrightarrow{r}, s \)
Where, \( r_i \) is a member of \( \overrightarrow{r} \) which is the input vector of the Bus.

(3). To replace
\[ s\ x \overset{g}{=} r\ x \]
By
\[ c_g\ x \land \text{Gate} (c_g, r, s) \]
Where, \( c_g \) is the control signal of the Gate.

(4). To replace
\[ s\ y \overset{r}{=} r\ x \]
By
\[ c_r\ x \land \text{Reg} (c_r, r, s) \]
Where, \( c_r \) is the control signal of the Register.

(5). To replace
\[ s\ y \overset{s}{=} s\ x \]
By
\[ \neg(c_r\ x) \land \text{Reg} (c_r, r, s) \]
In the specification of microinstruction the \( \neg(c_r\ x) \) is omitted traditionally.

(6). To replace
\[ s\ y \overset{c}{=} r\ x \quad \text{and} \quad s\ y \overset{c}{=} s\ x \]
By the boolean constant \( T \).
The replacements above can lead to the redundancy in specification: a device are written two or more copies and boolean constant T. We can use
\[ \vdash \alpha \land \alpha \iff \alpha \]
to eliminate those redundant copies and directly eliminate T.

After the replacements above all the specification entities of derivation models in a specification are replaced by control signals. For example, the specification entity \( s y \models r x \) is replaced by \( c_r x \). And the end time point \( y \) which is an existential variable is replaced by an instance time point formed as \( x + \delta \) (Here, \( \delta \) is an natural number) or by \( x + 1 \) if under the constraint 9.2.2.

To get a microprogram we instantiate each entry predicate which is an existential variable by an instance predicate about the entry address of a microinstruction. For example, to replace
\[ E_i x \]
by
\[ \text{address } x = a_i \]
Here, \( a_i \) is a natural number. The meaning of the instance predicate is that at the time point \( x \) the address of microprogram is equal to \( a_i \).

So, after these replacements a derivation model formed as, for example,
\[ (c_f x \land c_r x) [E_i, E_o] \]
is replaced by
\[ \forall x. \text{address } x = 5 \Rightarrow c_f x \land c_r x \land \text{address } (x + 1) = 6 \]
This is a microinstruction whose entry address is 5, it sends the control signal \( c_f \) and \( c_r \) and turns to another microinstruction whose entry address is 6.

### 9.2.2 About Clock

In the above process it is implied indirectly that we have, in fact, introduced the concept of clock. To gain a variable time unit computer we measure a time point
through entry predicate. More exactly, for a computer at the scheme 8 level we collect all the entry predicates of it as the clock of the computer. Suppose that all the entry predicates of a computer are \( E_j \) \((1 \leq j \leq n)\) then we define the clock of the computer as

\[
\forall t. \ E_1 t \lor E_2 t \lor \cdots \lor E_n t \ \ \ \ \ \ \ \ \ \ \ (c1)
\]

By making the clock be true we get a time point \( t \). By the definition we only need to make one entry predicate in \( (c1) \) to be true and we do not require that any two time intervals must be the same.

Under the constraint 9.2.2 we can further introduce one predicate which can be used to make each entry predicate in the clock true. The predicate is

\[
\forall x. \ \text{address} \ (x + 1) = \text{next-address} \ x \ \ \ \ \ \ \ \ (n1)
\]

We suppose there is a device at the register transfer level for \( (n1) \):

Address (next-address, address)

Note: for the Address the next-address is its input.

However, in this case we need, instead of the preceding replacement, to replace the leaving predicate of a derivation model, for example, “\( E_o \ y \)” by the instance predicate “next-address \( x = a_o \)” and meanwhile to guarantee that when the “\( E_o \ x \)” is taken as the entry predicate it is replaced by “address \( x = a_o \)” Here, the emphasis is that they have the same \( a_o \). For example, to instantiate

\[
(c_f \ x \land c_r \ x) \ [E_i, \ E_o]
\]

By

\[
\forall x. \ \text{address} \ x = 5 \Rightarrow c_f \ x \land c_r \ x \land \text{next-address} \ x = 6 \ \ \ \ \ \ \ \ \ (i1)
\]

We must guarantee to instantiate, for example,

\[
(c_g \ x \land c_h \ x) \ [E_o, \ E_i]
\]

By

\[
\forall x. \ \text{address} \ x = 6 \Rightarrow c_g \ x \land c_h \ x \land \text{next-address} \ x = 7 \ \ \ \ \ \ \ \ (i2)
\]

Here, \( E_o \) is an entry predicate. We call \( (i1) \) and \( (i2) \) microinstructions.
As we have had $E_0$, which can be instantiated, for example, by address $0 = 0$, and through using (n1) we can make each entry predicate in (c1) to be true step by step.

To implement all the microinstructions in a computer we need to introduce a device “MicroProgram”:

MicroProgram (address, next-address, $\bar{c}$)

Here, “address” is input and $\bar{c}$ is a vector of control signals used to control the relevant devices in a computer.

It is emphasized that in our construction process for a class of computers we need not directly introduce the concept of clock. Clock is implied by our derivation process. This is just as the shown by the whole preceding construction process. So, we can call these computers to be of non-clock. When under constraint 9.2.1 we may, but not certainly necessary, introduce the clock (c1) we call it the variable-clock computer. And under constraint 9.2.2 we need to introduce (n1) and through it to define the time point, therefore, we call it the constant-clock computer.

For the branch structure models $b_r s$ of the scheme 8 we replace its entry predicate and all leaving predicates as we do for other models. For example, to replace

$$[E_i, b_1 (E_{o1}, b_2 (E_{o2}, E_{o3}))]$$

by

$$\forall x. \text{ address } x = a_i \rightarrow (b_1 x \Rightarrow \text{ next-address } x = a_{o1} \downarrow$$

$$(b_2 x \Rightarrow \text{ next-address } x = a_{o2} \downarrow \text{ next-address } x = a_{o3}))$$

And further we suppose that there is a corresponding device

Branch$_1$ (b$_1$, b$_2$, address, next-address)

at the register transfer level. More generally, we suppose there are device kind for branch structure model as below:
Branch ($\bar{b}$, address, next-address)
Here, $\bar{b}$ and “address” are inputs.

9.2.3 Implied Zero Time Point

In [50] J. J. Joyce writes that “One of the most interesting discoveries of this exercise was that the formal verification of the design missed a design error: there is no reset button to initialize the microcode program counter”. For our derivation the “reset” is to get $E_0$. However, it is just one of the advantages of our approach that derivation which is based on our derivation model cannot miss the “reset”. It is very natural to construct the $E_0$ which is presented as a goal to be constructed even from specification scheme 1. In the specifications of a computer from the scheme 1 to the scheme 8 an initiation $E_0$ is always necessary because the other part of specification scheme, we read it specification body and write it $sb$, is a loop on the execution of machine instructions and the loop needs a start time point. Although we can use a special device to do the job of $E_0$, we will prove in this section that the body $sb$ itself can do this.

In physics when we switch on power to make a computer start its work the start time point is completely stochastic such that it can make any address of a microinstruction become the initial point to start the running of microprogram of the computer. For derivation this is arbitrarily to make an entry predicate of the computer become true. We do not know which entry predicate will be at all but we can always affirm that there is one and only one such predicate.

Postulate 9.2.3.1. [Stochastic starting]:

At the level of specification scheme 8 that a computer powers up at a time point is expressed by the possibility that there is and only is an arbitrary entry predicate which can be true at the time point.

Here, we assume that a machine at the scheme 8 level. But this is not essential, in fact, we can do at the higher or lower scheme levels.
Let us prove the the initiation $E_0$ is implied by specification body $sb$.

**Theorem 9.2.3.1. [implied 0 time point]:**

For the specification body $sb$ and initiation $E_0$ of the scheme 8 we have

$\vdash sb \Rightarrow E_0$

**Proof:**

By the postulate 9.2.3.1 when switching on power we get a time point $t_s$ and an entry predicate $E_s$ such $E_s t_s$ is true. Then there are two cases.

1. If the $E_s$ is just at the $E$ then we can take the time point $t_s$ as zero time point. Therefore we get $E_0$.

2. If the $E_s$ is not $E$ then we can prove through a finite time interval a $t_e$ will be reached such that $E t_e$ true, and we can take the $t_e$ as zero time point of this computer.

- Commencing from the scheme 2 which has the form as

  $\alpha[E,E]$ 

  all later constructions continually introduce new model which introduce new entry predicates to decompose and only finite entry predicates are introduced in these construction steps. Therefore, starting from the $E_s t_s$ through finite links and branch structures a time point $t_e$ can always be reached such that $E t_e$ is true. Note: Branch structures are constituted by conditional structures formed as $P \Rightarrow \alpha \downarrow \beta$. When passing the condition structure the running of computer is directed to either $\alpha$ or $\beta$.  

\[\square\]

**9.2.4 Abstract Algorithm 9 and Specification Scheme 9**

We use abstract algorithm 9 to collect that we have done in this section up to this.
Abstract Algorithm 9. [constructing database and microprogram]:

For a specification at the scheme 8 level following the four steps below to construct a specification at the scheme 9 level:

1. For the specification entities formed as $\alpha \equiv \beta$, $\alpha \equiv \beta$, $\alpha \equiv \beta$, $\alpha \equiv \beta$, $\alpha \equiv \beta$, and $\alpha \equiv \beta$, to replace them, following the described steps in the subsection 9.2.1, by the corresponding control signals which come from the devices for function unit, Register, Bus, Gate and boolean constant T respectively (but for Bus and T there are control signals necessary) and these devices are also introduced separately.

2. For the logic boundary, entry and leaving predicates, of the derivation models and the branch structure models following the subsections 9.2.1 and 9.2.2 to replace them and then further replace these models by a device "MicroProgram" and a group of device "Branch".

3. Introduce the device "Address" as mentioned in the subsection 9.2.2.

4. Eliminate $E_0$.

We will discuss the specification scheme 9 in the section 10.1 as it is taken as the end point of our formal derivation of a class of computers.

For the introductions of Devices for function unit, Register and Connection their corrections are obvious. For Bus-Gate structure we introduce a theorem below to prove its correctness. In the subsection 9.1.1 we have three steps (bg1), (bg2) and (bg3) to construction a Bus-Gate structure as below:

\[
\begin{align*}
\cdots \land (\Gamma_{11} \land s \ y = r_1 \ x \land \Gamma_{1r}) \ [E_a, \ E_b] \land \\
\cdots \land (\Gamma_{21} \land s \ y = r_2 \ x \land \Gamma_{2r}) \ [E_c, \ E_d] \land \\
\cdots \land (\Gamma_{1r} \land s \ y = r_n \ x \land \Gamma_{nr}) \ [E_e, \ E_f] \land \cdots \cdots \cdots \cdots \cdots \cdots \cdots \cdots \cdots \cdots \cdots (bg1)
\end{align*}
\]

Here, $\Gamma_{ii}$ and $\Gamma_{ir}$ ($1 \leq i \leq n$) represent the conjunction of zero or more specification entities.

We first transform it to

\[
\begin{align*}
\cdots \land (\Gamma_{11} \land s \ y = r'_1 \ y \land r'_1 \ y = r_1 \ y \land r_1 \ y = r_1 \ x \land \Gamma_{1r}) \ [E_a, \ E_b] \land \\
\cdots \land (\Gamma_{21} \land s \ y = r'_2 \ y \land r'_2 \ y = r_2 \ y \land r_2 \ y = r_2 \ x \land \Gamma_{2r}) \ [E_c, \ E_d] \land
\end{align*}
\]
\[ \cdots \land (\Gamma_{nl} \land s \equiv r'_n y \land r'_n y = r_n y \land r_n y = r_n x \land \Gamma_{nr}) [E_e, E_f] \land \cdots \]

........................................................................................................ (bg2)

Then we have an implementation formed as

\[ \cdots \land (\Gamma_{il} \land c_{g_1} y \land r_1 y = r_1 x \land \Gamma_{ir}) [E_a, E_b] \land \]

\[ \cdots \land (\Gamma_{2i} \land c_{g_2} y \land r_2 y = r_2 x \land \Gamma_{2r}) [E_c, E_d] \land \]

\[ \cdots \land (\Gamma_{ni} \land c_{g_n} y \land r_n y = r_n x \land \Gamma_{nr}) [E_e, E_f] \land \]

\[ \cdots \land \text{Gate}_1 (c_{g_1} r_1, r'_1) \land \text{Gate}_2 (c_{g_2} r_2, r'_2) \land \cdots \land \text{Gate}_n (c_{g_n} r_n, r'_n) \land \]

\[ \cdots \land \text{Bus} (r'_1, r'_2, \cdots, r'_n, s) \land \cdots ................................................ (bg3) \]

We can see that (bg1) can be viewed as specification and (bg3) as the implementation. However, we should specially point that the (bg2) is only taken as a middle step and description means but not used to correctness proof.

**Theorem 9.2.4.1. [correctness of bus-gate structure]:**

\[ \vdash (bg3) \Rightarrow (bg1) \]

**Proof:**

By the theorem 8.1.4.1 (page 105) at any time point \( y \) there is one and only one model in (bg3), let us suppose it is formed as

\[ (\Gamma_{ii} \land c_{g_i} y \land r_i y = r_i x \land \Gamma_{ir}) [E_{\alpha}, E_{\beta}] \ (1 \leq i \leq n), \]

its entry predicate \( E_{\alpha} \) gets boolean value True. And all other entry predicates get boolean value False, therefore the corresponding model become True. Further, only the gate-control signal \( c_{g_i} \) of the model gets value True and all others gate-control signal get value False. By the specification of Gate (gs) (page 118) at the time point \( y \) we only have one Gate\(_i\) (\( c_{g_i}, r_i, r'_i \))

\[ r'_i y = r_i y \]

and all other gates:

\[ r'_j y = \text{Float} \ (1 \leq j \leq n \land j \neq i) \]

By the specification of Bus (bs) (page 118) we have

\[ s y = \text{Dest_Tri (Float} \cup \cdots \cup \text{Float} \cup \text{Mk_Tri (r}_i y) \cup \text{Float} \cup \cdots \cup \text{Float}) \]
\[ = \text{Dest.Tri (Mk.Tri (r y))} \quad \text{by the axiom (bo) (see page 118)} \]
\[ = r_i y \quad \text{by the axiom (dmt) (see page 118)} \]

Hence, we get
\[ (\Gamma_{il} \land s \ y = r_i x \land \Gamma_{ir}) [E_\alpha, E_\beta]. \]

As all other model in (bg1) have become True therefore we can get (bg1).

Based on the theorem 9.2.4.1 and refer to the specification scheme 9 in the next chapter it is obvious about totalness and correctness of the abstract algorithm 9.
Chapter 10

Structural Specification Scheme

10.1 Structural Specification Scheme

We call the specification scheme 9 the structural specification scheme as it is considered to have enough detail for implementation at register transfer level, therefore, can be taken as the end point of our formal derivation for a class of computers.

Specification Scheme 9 as Structural Specification Scheme

Specification Scheme: \( SSS \)

Abbreviation: \( SSS : \) Structural Spec. Scheme

\[
SSS \quad ::= \quad Control \quad \wedge Datapath
\]

\[
Control \quad ::= \quad Address \ (\text{next-address, address}) \quad \wedge \quad \text{MicroProgram} \ (\text{address, next-address, } \bar{c}) \quad \wedge \quad Brs
\]

\[
Brs \quad ::= \quad \text{branch structure set}
\]

\[
Datapath \quad ::= \quad Device
\]

\[
\quad | \quad Device \wedge Datapath
\]

\[
Brs \quad ::= \quad \text{Branch} \ (\bar{b}, \text{address, next-address}) \quad | \quad \text{Branch} \ (\bar{b}, \text{address, next-address}) \wedge Brs
\]

\[
Device \quad ::= \quad \text{Fun} \ (\bar{c}, \bar{i}, \bar{o}) \quad \text{Fun} \quad : \quad \text{Device for function unit}
\]

\[
\quad | \quad \text{Reg} \ (c_r, i, o) \quad c_r \quad : \quad \text{control signal of register}
\]

\[
\quad | \quad \text{Bus} \ (\bar{i}, \bar{o}) \quad i/o \quad : \quad \text{in/output signal of gate}
\]
A computer at the structural specification scheme level consists of two parts: control and datapath. The control is constituted by a device Address, a device MicroProgram and a kind of devices Branch. The datapath is constituted by the four kinds of devices: device for function unit, bus, register and gate. In the inside of the control there are internal signals: address and next-address and in the inside of the datapath there are internal signals: input and output signals $i$.
and \( \bar{c} \). The communications between control and datapath are through control (boolean valued) signals \( \bar{c} \) from control to datapath for controlling the data flow \( \bar{i} \) and \( \bar{o} \) and control signals \( \bar{b} \) from datapath to control for controlling the address flow of microprogram.

An interesting observation to the formally derived computer class is that there is a control circle:

\[
\{ \bar{a}, \bar{c}, \bar{b} \}.
\]

Here, \( \bar{a} = < \text{address, next-address} > \). However, \( \bar{a} \) is the center of the circle.
Part III

Case Study and Further Discussion
Chapter 11

Deriving Gordon’s Computer

In this chapter we will apply the stepwise constructions to Gordon’s computer following the schemes of specification, construction, and proof as laid down in Part 2. We will formally construct three correct implementations at register transfer level from a formal specification at machine instruction level. Finally we will compare these derived implementations and the implementation verified by [50].

One point should be emphasized. In the formal derivation we deal with pure structure and try to keep the specification as general as possible. We leave any further detail uninterpreted if it is irrelevant to the pure structure and construction.

11.1 Introduction to Gordon’s Computer

Gordon’s computer is a simple general-purpose computer invented as a nontrivial example for the formal specification and verification of hardware [50]. We take this computer as a case study for formal derivation.

A top-level specification of the computer is at machine instruction level. This is typically what an assembler language programmer would need to write a program.
At the top level Gordon’s computer has a memory and two registers: the program counter PC and the accumulator ACC. The top-level specification includes two parts: the front panel operation and the machine instructions.

The front panel of the computer has a button, a four-position knob, and switches. When the computer is running pushing the button interrupts the execution of the program and the computer idles. The effect of pushing the button when the computer is idling is determined by the position of the knob. If the knob is in position 0 the effect is that the word determined by the switches is loaded into the PC. Pushing the button when the knob is in position 1 loads the word determined by the switches into the ACC. When the knob is in position 2, the contents of the ACC is stored in the memory at the location held in the PC. Finally, knob position 3 is used to start the execution of the program stored in memory beginning at the location in the PC.

Knob = 0    Load PC
Knob = 1    Load ACC
Knob = 2    Store ACC at PC
Knob = 3    Start execution at PC

The computer has eight machine instructions HALT, JMP, JZR, ADD, SUB, LOAD, STORE and SKIP, with opcode 0-7, respectively.

HALT    Stop execution
JMP L    Jump to address L
JZR L    Jump to address L if ACC = 0
ADD L    Add contents of address L to ACC
SUB L    Subtract contents of address L from ACC
LD L     Load contents of address L into ACC
ST L     Store contents of ACC in address L
SKIP     Skip to next instruction
Above is the informal description for Gordon’s computer that will formally be dealt with in the next section. We will follow [50] exactly and, as mentioned, we only deal formally with pure structure and leave out all irrelevant details.

11.2 Formal Derivation

In this section we show 11 construction steps staring from the behavior specification and ending in the structural specification for constructing an implementation for Gordon’s computer.

11.2.1 Behavior Specification

Specification 0:

Computer (button, knob, switches, memory, pc, acc, idle) ≡
∀u. (memory (u + 1), pc (u + 1), acc (u + 1), idle (u + 1)) =
(idle u ⇒ Idle ↓ Busy)

Idle ≡ button u ⇒
(Val₂ (knob u) = 0) ⇒
(memory u, Cut₁₆₁₃ (switches u), acc u, T) ↓
(Val₂ (knob u) = 1) ⇒
(memory u, pc u, switches u, T) ↓
(Val₂ (knob u) = 2) ⇒
(Store₁₃ pc u (acc u) (memory u), pc u, acc u, T) ↓
(memory u, pc u, acc u, F)) ↓
(memory u, pc u, acc u, T)

Busy ≡ button u ⇒
(memory u, pc u, acc u, T) ↓
let op = Val₃ (Opcode (Fetch₁₃ memory u) (pc u)) in
let addr = Cut_{16..13} (Fetch_{13} (memory u) (pc u)) in

( (op = 0) ⇒ (memory u, pc u, acc u, T) ↓
( (op = 1) ⇒ (memory u, addr, acc u, F) ↓
( (op = 2) ⇒
  ( (Val_{16} (acc u) = 0) ⇒
    (memory u, addr, acc u, F) ↓
    (memory u, Inc_{13} (pc u), acc u, F) ↓
( (op = 3) ⇒ (memory u,
    Inc_{13} (pc u),
    Add_{16} (acc u) (Fetch_{13} (memory u) addr),
    F) ↓
( (op = 4) ⇒ (memory u,
    Inc_{13} (pc u),
    Sub_{16} (acc u) (Fetch_{13} (memory u) addr),
    F) ↓
( (op = 5) ⇒ (memory u,
    Inc_{13} (pc u),
    Fetch_{13} (memory u) addr,
    F) ↓
( (op = 6) ⇒ (Store_{13} addr (acc u) (memory u),
    Inc_{13} (pc u),
    acc u,
    F) ↓
    (memory u, Inc_{13} (pc u), acc u, F))

The meanings of the functions Val_{2}, Cut_{16..13}, Store_{13}, Opcode and Fetch_{13} are
exactly as in [50]. T and F are our boolean constants for true and false.
11.2.2 Refining Time

The specification 0 is at the machine instruction time level. In this subsection following the construction 1 and 2 we change the time level to the microinstruction level, namely, from time unit \(< u, u + 1 >\) to the time interval \(< t_1, t_2 >\). In the construction we introduce a global entry predicate \(E\) and two constraints:
\[
\forall t. t_1 < t < t_2 \Rightarrow \neg (E t) \text{ and } t_1 < t_2.
\]

**Specification 1:**

\[
E \, 0 \land
\forall t_1. \exists t_2. E \, t_1 \Rightarrow (\text{memory} (t_2), \, pc \, (t_2), \, acc \, (t_2), \, idle \, (t_2)) =
\]
\[
(idle \, t_1 \Rightarrow Idle \downarrow Busy) \land
\]
\[
(\forall t. t_1 < t < t_2 \Rightarrow \neg (E \, t)) \land
\]
\[
t_1 < t_2 \land
\]
\[
E \, t_2
\]

\(Idle \equiv \text{button} \, t_1 \Rightarrow \)
\[
(\text{Val}_2 (\text{knob} \, t_1) = 0) \Rightarrow
\]
\[
(\text{memory} \, t_1, \, \text{Cut}_{16 \cdot 13} (\text{switches} \, t_1), \, acc \, t_1, \, T) \downarrow
\]
\[
(\text{Val}_2 (\text{knob} \, t_1) = 1) \Rightarrow
\]
\[
(\text{memory} \, t_1, \, pc \, t_1, \, switches \, t_1, \, T) \downarrow
\]
\[
(\text{Val}_2 (\text{knob} \, t_1) = 2) \Rightarrow
\]
\[
(\text{Store}_{13} (pc \, t_1) (acc \, t_1) (memory \, t_1), \, pc \, t_1, \, acc \, t_1, \, T) \downarrow
\]
\[
(\text{memory} \, t_1, \, pc \, t_1, \, acc \, t_1, \, F)) \downarrow
\]
\[
(\text{memory} \, t_1, \, pc \, t_1, \, acc \, t_1, \, T)
\]

\(Busy \equiv \text{button} \, t_1 \Rightarrow \)
\[
(\text{memory} \, t_1, \, pc \, t_1, \, acc \, t_1, \, T) \downarrow
\]

let \(op = \text{Val}_3 (\text{Opcode} (\text{Fetch}_{13} (\text{memory} \, t_1) (pc \, t_1)))\) in

let \(addr = \text{Cut}_{16 \cdot 13} (\text{Fetch}_{13} (\text{memory} \, t_1) (pc \, t_1))\) in

\( (\text{op} = 0) \Rightarrow (\text{memory} \, t_1, \, pc \, t_1, \, acc \, t_1, \, T) \downarrow \)
\[(op = 1) \Rightarrow (memory \ t_1, \ addr, \ acc \ t_1, \ F) \downarrow\]

\[(op = 2) \Rightarrow\]

\[((\text{Val}_{16} (acc \ t_1) = 0) \Rightarrow\]

\[(memory \ t_1, \ addr, \ acc \ t_1, \ F) \downarrow\]

\[(memory \ t_1, \ \text{Inc}_{13} (pc \ t_1), \ acc \ t_1, \ F)) \downarrow\]

\[(op = 3) \Rightarrow (memory \ t_1,\]

\[\quad \text{Inc}_{13} (pc \ t_1),\]

\[\quad \text{Add}_{16} (acc \ t_1) \ (\text{Fetch}_{13} (memory \ t_1) \ addr),\]

\[\quad F) \downarrow\]

\[(op = 4) \Rightarrow (memory \ t_1,\]

\[\quad \text{Inc}_{13} (pc \ t_1),\]

\[\quad \text{Sub}_{16} (acc \ t_1) \ (\text{Fetch}_{13} (memory \ t_1) \ addr),\]

\[\quad F) \downarrow\]

\[(op = 5) \Rightarrow (memory \ t_1,\]

\[\quad \text{Inc}_{13} (pc \ t_1),\]

\[\quad \text{Fetch}_{13} (memory \ t_1) \ addr,\]

\[\quad F) \downarrow\]

\[(op = 6) \Rightarrow (\text{Store}_{13} \ addr \ (acc \ t_1) \ (memory \ t_1),\]

\[\quad \text{Inc}_{13} (pc \ t_1),\]

\[\quad acc \ t_1,\]

\[\quad F) \downarrow\]

\[(memory \ t_1, \ \text{Inc}_{13} (pc \ t_1), \ acc \ t_1, \ F))\]

11.2.3 Setting State Transition

Following the construction 3 we set a corresponding state transition for each machine instruction. There are eleven state transitions which correspond to all possible front panel operations and machine instructions of Gordon’s computer.

**Specification 2:**
$E 0 \land$

$\forall t_1. \exists t_2. E t_1 \Rightarrow (idle t_1 \Rightarrow Idle_{st} \downarrow Busy_{st}) \land$

$(\forall t. t_1 < t < t_2 \Rightarrow \neg E t) \land t_1 \leq t_2 \land E t_2$

$Idle_{st} \doteq button t_1 \Rightarrow$

$((Val_2 (knob t_1) = 0) \Rightarrow state_{transition}_0 \downarrow$

$(Val_2 (knob t_1) = 1) \Rightarrow state_{transition}_1 \downarrow$

$(Val_2 (knob t_1) = 2) \Rightarrow state_{transition}_2 \downarrow state_{transition}_3 \downarrow$

$state_{transition}_4$

$Busy_{st} \doteq button t_1 \Rightarrow state_{transition}_4 \downarrow$

let $op = Val_3 (Opcode (Fetch_{13} (memory t_1) (pc t_1)))$ in

let $addr = Cut_{16..13} (Fetch_{13} (memory t_1) (pc t_1))$ in

$(op = 0) \Rightarrow state_{transition}_4 \downarrow$

$(op = 1) \Rightarrow state_{transition}_5 \downarrow$

$(op = 2) \Rightarrow$

$((Val_{16} (acc t_1) = 0) \Rightarrow state_{transition}_5 \downarrow state_{transition}_6) \downarrow$

$(op = 3) \Rightarrow state_{transition}_7 \downarrow$

$(op = 4) \Rightarrow state_{transition}_8 \downarrow$

$(op = 5) \Rightarrow state_{transition}_9 \downarrow$

$(op = 6) \Rightarrow state_{transition}_10 \downarrow state_{transition}_6) \downarrow$

$state_{transition}_0 \doteq (memory t_2 = memory t_1 \land$

$pc t_2 = Cut_{16..13} (switches t_1) \land$

$acc t_2 = acc t_1 \land$

$idle t_2 = T) \downarrow$

$state_{transition}_1 \doteq (memory t_2 = memory t_1 \land$

$pc t_2 = pc t_1 \land$

$acc t_2 = switches t_1 \land$

$idle t_2 = T) \downarrow$
\[
\text{state}_1 \cdot \text{transition} \equiv \begin{align*}
\text{memory } t_2 & = \text{Store}_{13} (\text{pc } t_1) (\text{acc } t_1) (\text{memory } t_1) \land \\
\text{pc } t_2 & = \text{pc } t_1 \\
\text{acc } t_2 & = \text{acc } t_1 \land \\
\text{idle } t_2 & = \text{T} 
\end{align*}
\]

\[
\text{state}_2 \cdot \text{transition} \equiv \begin{align*}
\text{memory } t_2 & = \text{memory } t_1 \land \\
\text{pc } t_2 & = \text{pc } t_1 \land \\
\text{acc } t_2 & = \text{acc } t_1 \land \\
\text{idle } t_2 & = \text{F} 
\end{align*}
\]

\[
\text{state}_3 \cdot \text{transition} \equiv \begin{align*}
\text{memory } t_2 & = \text{memory } t_1 \land \\
\text{pc } t_2 & = \text{pc } t_1 \land \\
\text{acc } t_2 & = \text{acc } t_1 \land \\
\text{idle } t_2 & = \text{T} 
\end{align*}
\]

\[
\text{state}_4 \cdot \text{transition} \equiv \begin{align*}
\text{memory } t_2 & = \text{memory } t_1 \land \\
\text{pc } t_2 & = \text{pc } t_1 \land \\
\text{acc } t_2 & = \text{acc } t_1 \land \\
\text{idle } t_2 & = \text{F} 
\end{align*}
\]

\[
\text{state}_5 \cdot \text{transition} \equiv \begin{align*}
\text{memory } t_2 & = \text{memory } t_1 \land \\
\text{pc } t_2 & = \text{addr} \land \\
\text{acc } t_2 & = \text{acc } t_1 \land \\
\text{idle } t_2 & = \text{F} 
\end{align*}
\]

\[
\text{state}_6 \cdot \text{transition} \equiv \begin{align*}
\text{memory } t_2 & = \text{memory } t_1 \land \\
\text{pc } t_2 & = \text{Inc}_{13} (\text{pc } t_1) \land \\
\text{acc } t_2 & = \text{acc } t_1 \land \\
\text{idle } t_2 & = \text{F} 
\end{align*}
\]

\[
\text{state}_7 \cdot \text{transition} \equiv \begin{align*}
\text{memory } t_2 & = \text{memory } t_1 \land \\
\text{pc } t_2 & = \text{Inc}_{13} (\text{pc } t_1) \land \\
\text{acc } t_2 & = \text{Add}_{16} (\text{acc } t_1) (\text{Fetch}_{13} (\text{memory } t_1) \text{addr}) \land \\
\text{idle } t_2 & = \text{F} 
\end{align*}
\]

\[
\text{state}_8 \cdot \text{transition} \equiv \begin{align*}
\text{memory } t_2 & = \text{memory } t_1 \land \\
\text{pc } t_2 & = \text{Inc}_{13} (\text{pc } t_1) \land \\
\text{acc } t_2 & = \text{Sub}_{16} (\text{acc } t_1) (\text{Fetch}_{13} (\text{memory } t_1) \text{addr}) \land \\
\text{idle } t_2 & = \text{F} 
\end{align*}
\]

\[
\text{state}_9 \cdot \text{transition} \equiv \begin{align*}
\text{memory } t_2 & = \text{memory } t_1 \land \\
\text{pc } t_2 & = \text{Inc}_{13} (\text{pc } t_1) \land \\
\text{acc } t_2 & = \text{acc } t_1 \land \\
\text{idle } t_2 & = \text{F} 
\end{align*}
\]
\[ \begin{align*}
\text{pc} \ t_2 &= \text{Inc}_{13} (\text{pc} \ t_1) \land \\
\text{acc} \ t_2 &= \text{Fetch}_{13} (\text{memory} \ t_1) \ (\text{addr} \land \\
\text{idle} \ t_2 &= \text{F}) \\
\text{state}_{\text{transition}}_{10} \equiv (\text{memory} \ t_2 &= \text{Store}_{13} \ (\text{addr} \ (\text{acc} \ t_1) \ (\text{memory} \ t_1)) \land \\
\text{pc} \ t_2 &= \text{Inc}_{13} (\text{pc} \ t_1) \land \\
\text{acc} \ t_2 &= \text{acc} \ t_1 \land \\
\text{idle} \ t_2 &= \text{F})
\end{align*} \]

### 11.2.4 Resetting Global Structure

From the specification 2 we rearrange its \text{let}..\text{in} structure and then follow the construction 4 and 5 to select common blocks (sub-state transition set), and use the bracket to mark our selection for the parallel/sequential execution for the assignments of \text{let}..\text{in} sentences and predicate parts of condition sentences.

- **Re-arranging \text{let}..\text{in} structure:**
  
  We introduce
  
  \[
  \text{let} \ v = \text{Fetch}_{13} (\text{memory} \ t_1) \ (\text{pc} \ t_1) \ \text{in}
  \]
  
  to replace the two \text{let}..\text{in} sentences of the specification 2

  \[
  \text{let} \ \text{op} = \text{Val}_{3} (\text{Opcode} \ (\text{Fetch}_{13} (\text{memory} \ t_1) \ (\text{pc} \ t_1)))) \ \text{in}
  \]
  
  \[
  \text{let} \ \text{addr} = \text{Cut}_{16..13} (\text{Fetch}_{13} (\text{memory} \ t_1) \ (\text{pc} \ t_1)) \ \text{in}
  \]

  and use \text{Val}_{3} (\text{Opcode} \ v) \text{ to replace \text{op} and use} \text{Cut}_{16..13} \ (v) \text{ to replace} \text{addr} \text{ for the specification 2, because we suppose that \text{Val}_{3}, \text{Opcode}, \text{Fetch}_{13}, and} \text{Cut}_{13} \text{ can be implemented directly by primitive components at register transfer level.}

- **Selecting common block:**
  
  We select
  
  \[
  \text{sub}_{\text{state}_{\text{transition}}} \equiv (\text{pc} \ t_2 = \text{Inc}_{13} (\text{pc} \ t_1) \ \land \text{idle} \ t_2 = \text{F})
  \]
because it is shared as common sub-state transition set by state.transition 6, 7, 8, 9, and 10.

- **Selecting for parallel/sequential execution:**

  We select
  
  \[
  \text{Val}_2 \ (\text{knob} \ t_1) = i \ (0 \leq i \leq 3) \quad \text{and} \\
  \text{Val}_3 \ (\text{Opcode} \ v) = i \ (0 \leq i \leq 7)
  \]

  as parallel condition predicates, because they are easy to be executed in parallel by a branch structure such as the one which is a part of DECODE in [50].

**Specification 3:**

\[
E \ 0 \land \\
\forall t_1. \exists t_2. \ E \ t_1 \Rightarrow (\text{idle} \ t_1 \Rightarrow \{\text{Idle}_{st}\} \downarrow \{\text{Busy}_{rs}\}) \land \\
(\forall t. \ t_1 < t < t_2 \Rightarrow \neg E \ t) \land t_1 \leq t_2 \land E \ t_2
\]

\[
\text{Idle}_{st} \equiv \text{button} \ t_1 \Rightarrow \\
\{ (\text{Val}_2 \ (\text{knob} \ t_1) = 0) \Rightarrow \{\text{state.transition}_0\} \downarrow \\
(\text{Val}_2 \ (\text{knob} \ t_1) = 1) \Rightarrow \{\text{state.transition}_1\} \downarrow \\
(\text{Val}_2 \ (\text{knob} \ t_1) = 2) \Rightarrow \{\text{state.transition}_2\} \downarrow \\
\{\text{state.transition}_3\} \downarrow
\]

\[
\text{Busy}_{rs} \equiv \text{button} \ t_1 \Rightarrow \{\text{state.transition}_4\} \downarrow \\
\{ \text{let} \ v = \text{Fetch}_{13} \ (\text{memory} \ t_1) \ (\text{pc} \ t_1) \ \text{in} \\
(\text{Val}_3 \ (\text{Opcode} \ v) = 0) \Rightarrow \{\text{state.transition}_4\} \downarrow \\
(\text{Val}_3 \ (\text{Opcode} \ v) = 1) \Rightarrow \{\text{state.transition}_5\} \downarrow \\
(\text{Val}_3 \ (\text{Opcode} \ v) = 2) \Rightarrow \\
\{ (\text{Val}_{16} \ (\text{acc} \ t_1) = 0) \Rightarrow \{\text{state.transition}_5\} \downarrow \\
\{\text{state.transition}_6\} \downarrow\}
\]

\[
(\text{Val}_3 \ (\text{Opcode} \ v) = 3) \Rightarrow \{\text{state.transition}_7\} \downarrow
\]
\[
(\text{Val}_3 (\text{Opcode } v) = 4) \Rightarrow \{\text{state}\_\text{transition}_8\} \downarrow \\
(\text{Val}_3 (\text{Opcode } v) = 5) \Rightarrow \{\text{state}\_\text{transition}_9\} \downarrow \\
(\text{Val}_3 (\text{Opcode } v) = 6) \Rightarrow \{\text{state}\_\text{transition}_{10}\} \downarrow \\
\{\text{state}\_\text{transition}_6\}\}
\]

\[
\text{state}\_\text{transition}_0 \equiv (\text{memory } t_2 = \text{memory } t_1 \land \\
\quad \text{pc } t_2 = \text{Cut}_{16..13} (\text{switches } t_1) \land \\
\quad \text{acc } t_2 = \text{acc } t_1 \land \\
\quad \text{idle } t_2 = \text{T})
\]

\[
\text{state}\_\text{transition}_1 \equiv (\text{memory } t_2 = \text{memory } t_1 \land \\
\quad \text{pc } t_2 = \text{pc } t_1 \land \\
\quad \text{acc } t_2 = \text{switches } t_1 \land \\
\quad \text{idle } t_2 = \text{T})
\]

\[
\text{state}\_\text{transition}_2 \equiv (\text{memory } t_2 = \text{Store}_{13} (\text{pc } t_1) (\text{acc } t_1) (\text{memory } t_1) \land \\
\quad \text{pc } t_2 = \text{pc } t_1 \\
\quad \text{acc } t_2 = \text{acc } t_1 \land \\
\quad \text{idle } t_2 = \text{T})
\]

\[
\text{state}\_\text{transition}_3 \equiv (\text{memory } t_2 = \text{memory } t_1 \land \\
\quad \text{pc } t_2 = \text{pc } t_1 \land \\
\quad \text{acc } t_2 = \text{acc } t_1 \land \\
\quad \text{idle } t_2 = \text{F})
\]

\[
\text{state}\_\text{transition}_4 \equiv (\text{memory } t_2 = \text{memory } t_1 \land \\
\quad \text{pc } t_2 = \text{pc } t_1 \land \\
\quad \text{acc } t_2 = \text{acc } t_1 \land \\
\quad \text{idle } t_2 = \text{T})
\]

\[
\text{state}\_\text{transition}_5 \equiv (\text{memory } t_2 = \text{memory } t_1 \land \\
\quad \text{pc } t_2 = \text{Cut}_{16..13} v \land \\
\quad \text{acc } t_2 = \text{acc } t_1 \land \\
\quad \text{idle } t_2 = \text{F})
\]

\[
\text{state}\_\text{transition}_6 \equiv \text{sub}\_\text{state}\_\text{transition}_0 \land [\text{sub}\_\text{state}\_\text{transition}_5]
\]
\[\text{state}\_\text{transition}_7 \equiv \text{sub\_state}\_\text{transition}_1 \land [\text{sub\_state}\_\text{transition}_5]\]
\[\text{state}\_\text{transition}_8 \equiv \text{sub\_state}\_\text{transition}_2 \land [\text{sub\_state}\_\text{transition}_5]\]
\[\text{state}\_\text{transition}_9 \equiv \text{sub\_state}\_\text{transition}_3 \land [\text{sub\_state}\_\text{transition}_5]\]
\[\text{state}\_\text{transition}_{10} \equiv \text{sub\_state}\_\text{transition}_4 \land [\text{sub\_state}\_\text{transition}_5]\]
\[\text{sub\_state}\_\text{transition}_0 \equiv (\text{memory } t_2 = \text{memory } t_1 \land\]
\[\text{acc } t_2 = \text{acc } t_1)\]

\[\text{sub\_state}\_\text{transition}_1\]
\[\equiv (\text{memory } t_2 = \text{memory } t_1 \land\]
\[\text{acc } t_2 = \text{Add}_{16} (\text{acc } t_1) (\text{Fetch}_{13} (\text{memory } t_1) (\text{Cut}_{16..13} v)))\]

\[\text{sub\_state}\_\text{transition}_2\]
\[\equiv (\text{memory } t_2 = \text{memory } t_1 \land\]
\[\text{acc } t_2 = \text{Sub}_{16} (\text{acc } t_1) (\text{Fetch}_{13} (\text{memory } t_1) (\text{Cut}_{16..13} v)))\]

\[\text{sub\_state}\_\text{transition}_3\]
\[\equiv (\text{memory } t_2 = \text{memory } t_1 \land\]
\[\text{acc } t_2 = \text{Fetch}_{13} (\text{memory } t_1) (\text{Cut}_{16..13} v))\]

\[\text{sub\_state}\_\text{transition}_4\]
\[\equiv (\text{memory } t_2 = \text{Store}_{13} (\text{Cut}_{16..13} v) (\text{acc } t_1) (\text{memory } t_1) \land\]
\[\text{acc } t_2 = \text{acc } t_1)\]

\[\text{sub\_state}\_\text{transition}_5 \equiv (\text{pc } t_2 = \text{Inc}_{13} (\text{pc } t_1) \land\]
\[\text{idle } t_2 = \text{F})\]

### 11.2.5 Decomposing to Term Transition

This step is essential for the construction of a class of computers. We follow the construction 6 to decompose the specification 3 to a specification 4 below which is at term transition level.

As the mentioned by the convenience 5.3.1.1 in the further construction process the two constraints introduced in the specification 1
\[ \forall t. t_1 < t < t_2 \Rightarrow \neg (E \ t) \text{ and } t_1 < t_2. \]

will be omitted but they will be certainly satisfied by the process.

From this subsection we will use the forms of the derivation models. We suggest reader to review them if you have not been familiar with them. To suit the convention we use \( x \) and \( y \) to replace \( t_1 \) and \( t_2 \).

**Specification 4:**

\[ E_0 \land \]

\[ (idle \ y = idle \ x \land kss_0) [E, \ E] \land \]

\[ [E, \ (idle \ x) \ (E_1, \ E_2)] \land \]

\[ (button \ y = button \ x \land kss_1) [E_1, \ E_1] \land \]

\[ [E_1, \ (button \ x) \ (E_3, \ E_4)] \land \]

\[ (button \ y = button \ x \land kss_2) [E_2, \ E_2] \land \]

\[ [E_2, \ (button \ x) \ (E_4, \ E_5)] \land \]

\[ (knob \ y = knob \ x \land kss_3) [E_3, \ E_3] \land \]

\[ [E_3, \ (Val_2 (knob \ x) = 0) \ (E_6, \]

\[ \quad (Val_2 (knob \ x) = 1) \ (E_7, \]

\[ \quad \quad (Val_2 (knob \ x) = 2) \ (E_8, \]

\[ \quad \quad \quad (E_9)))] \land \]

\[ (v \ y = \text{Fetch}_13 (memory \ x) \ (pc \ x) \land kss_4) [E_5, \ E_{10}] \land \]

\[ (v \ y = v \ x \land kss_5) [E_{10}, \ E_{10}] \]

\[ [E_{10}, \ (Val_3 (Opcodex(v \ x)) = 0) \]

\[ (E_4, \ (Val_3 (Opcodex(v \ x)) = 1) \]

\[ (E_{11}, \ (Val_3 (Opcodex(v \ x)) = 2) \]

\[ (E_{12}, \ (Val_3 (Opcodex(v \ x)) = 3) \]

\[ (E_{13}, \ (Val_3 (Opcodex(v \ x)) = 4) \]

\[ (E_{14}, \ (Val_3 (Opcodex(v \ x)) = 5) \]

\[ (E_{15}, \ (Val_3 (Opcodex(v \ x)) = 6) \]

\[ (E_{16}, \ E_{17}))])]. \]
(\text{acc } y = \text{acc } x \land kss_6) \ [E_{12}, E_{12}] \land \\
[E_{12}, (\text{Val}_{16} (\text{acc } x) = 0) \ (E_{11}, E_{18})] \land \\
st_0 \ [E_9, E] \land st_1 \ [E_7, E] \land st_2 \ [E_8, E] \land \\
st_3 \ [E_9, E] \land st_4 \ [E_4, E] \land st_5 \ [E_{11}, E] \land \\
(stst_1 \land kss_7) \ [E_{13}, E_{19}] \land (stst_2 \land kss_7) \ [E_{14}, E_{19}] \land \\
(stst_3 \land kss_7) \ [E_{15}, E_{19}] \land (stst_4 \land kss_7) \ [E_{16}, E_{19}] \land \\
(stst_0 \land kss_7) \ [E_{17}, E_{19}] \land (stst_0 \land kss_7) \ [E_{18}, E_{19}] \land \\
(stst_5 \land kss_8) \ [E_{19}, E] \\

Here,

\[
st_0 \equiv (\text{memory } y = \text{memory } x \land \\
 pc y = \text{Cut}_{16,13} (\text{switches } x) \land \\
 acc y = \text{acc } x \land \\
 idle y = T) \\
\]

\[
st_1 \equiv (\text{memory } y = \text{memory } x \land \\
 pc y = pc x \land \\
 acc y = \text{switches } x \land \\
 idle y = T) \\
\]

\[
st_2 \equiv (\text{memory } y = \text{Store}_{13} (pc x) (\text{acc } x) (\text{memory } x) \land \\
 pc y = pc x \\
 acc y = \text{acc } x \land \\
 idle y = T) \\
\]

\[
st_3 \equiv (\text{memory } y = \text{memory } x \land \\
 pc y = pc x \land \\
 acc y = \text{acc } x \land \\
 idle y = \text{F}) \\
\]

\[
st_4 \equiv (\text{memory } y = \text{memory } x \land \\
 pc y = pc x \land \\
 acc y = \text{acc } x \land \\
 idle y = T) \\
\]
\( st_5 \triangleq (\text{memory } y = \text{memory } x \land \\
\quad \quad pc \ y = \text{Cut}_{16..13} \ (v \ x) \land \\
\quad \quad acc \ y = acc \ x \land \\
\quad \quad idle \ y = F) \)

\( st_0 \triangleq (\text{memory } y = \text{memory } x \land \\
\quad \quad acc \ y = acc \ x) \)

\( st_1 \triangleq (\text{memory } y = \text{memory } x \land \\
\quad \quad acc \ y = \text{Add}_{16} \ (acc \ x) \ (\text{Fetch}_{13} \ (\text{memory } x) \ (\text{Cut}_{16..13} \ v))) \)

\( st_2 \triangleq (\text{memory } y = \text{memory } x \land \\
\quad \quad acc \ y = \text{Sub}_{16} \ (acc \ x) \ (\text{Fetch}_{13} \ (\text{memory } x) \ (\text{Cut}_{16..13} \ v))) \)

\( st_3 \triangleq (\text{memory } y = \text{memory } x \land \\
\quad \quad acc \ y = \text{Fetch}_{13} \ (\text{memory } x) \ (\text{Cut}_{16..13} \ v)) \)

\( st_4 \triangleq (\text{memory } y = \text{Store}_{13} \ (\text{Cut}_{16..13} \ v) \ (acc \ x) \ (\text{memory } x) \land \\
\quad \quad acc \ y = acc \ x) \)

\( st_5 \triangleq (pc \ y = \text{Inc}_{13} \ (pc \ x) \land \\
\quad \quad idle \ y = F) \)

\( kss_0 \triangleq \text{button } y = \text{button } x \land kss_1 \)

\( kss_1 \triangleq \text{knob } y = \text{knob } x \land kss_3 \)

\( kss_2 \triangleq kss_6 \)

\( kss_3 \triangleq \text{switches } y = \text{switches } x \land kss_4 \)

\( kss_4 \triangleq kss_6 \)

\( kss_5 \triangleq v \ y = v \ x \land kss_6 \)

\( kss_6 \triangleq kss_7 \land kss_8 \)

\( kss_7 \triangleq pc \ y = pc \ x \)

\( kss_8 \triangleq \text{memory } y = \text{memory } x \land acc \ y = acc \ x \)

It should be aware that although two models:

\( (st_0 \land kss_7) [E_{17}, E_{19}] \) and \( (st_0 \land kss_7) [E_{18}, E_{19}] \)
have the same specification entity but which has not been selected as common block. This is because its implementation does not expense device resource.

From the specification 4 we can get an abstract microprogram structure which describes the logic connection relation between the term transitions of the specification 4. This is a generally existent high level architecture in the design process of computer. It is significant that this high level architecture is derived by out derivation model very naturally.
Figure 11.2.5.1 Flowchart of the specification of Gordon’s computer at term transition level.

11.2.6 Decomposing to Function Unit and Connection

We follow the construction 7 to decompose the specification 4 to a specification which is at function unit and connection level and has its function units to be assigned to devices. We need to give more details of this construction process as, we will see, some Register and Gate from realistic world are assumed to have function of, instead of only a function, composition of two or more functions. For example, instead of ideal register:

\[
\text{Reg} (c, i, o) \equiv \forall t. o (t + 1) = (c \uparrow i \downarrow o \downarrow t)
\]

we use one from [50]:

\[
\text{Reg}_c (c, i, o) \equiv \forall t. o (t + 1) = (c \uparrow_i t \Rightarrow \text{Cut}_{16..13} (i \downarrow t) \downarrow o \downarrow t)
\]

The register has an additional function \text{Cut}_{16..13}.

**Convenience 11.2.6.1:**

From this subsection we have a convenience that when we write a specification we will only write its new part and will omit the unchanged part.

Separating out Single Complex Term Transition

We introduce the specification 5 as the first step of construction from the specification 4. In the specification 5 the single complex term transitions \text{term.transition}_i (0 \leq i \leq 9) are pick out from state transition \text{st}_i (0 \leq i \leq 5) and sub-state transition \text{sst}_i (0 \leq i \leq 5).

**Specification 5:**

\[
\begin{align*}
st_0 &= \text{keep.memory} \land \text{term.transition}_0 \land \text{keep.acc} \land \text{idle.is.true}, \\
st_1 &= \text{keep.memory} \land \text{keep.pc} \land \text{term.transition}_1 \land \text{idle.is.true},
\end{align*}
\]
\[ st_2 = \text{term.transition}_2 \land \text{keep.pc} \land \text{keep.acc} \land \text{idle.is.true}, \]
\[ st_3 = \text{keep.memory} \land \text{keep.pc} \land \text{keep.acc} \land \text{idle.is.false}, \]
\[ st_4 = \text{keep.memory} \land \text{keep.pc} \land \text{keep.acc} \land \text{idle.is.true}, \]
\[ st_5 = \text{keep.memory} \land \text{term.transition}_3 \land \text{keep.acc} \land \text{idle.is.false}, \]
\[ ss_0 = \text{keep.memory} \land \text{keep.acc}, \]
\[ ss_1 = \text{keep.memory} \land \text{term.transition}_4, \]
\[ ss_2 = \text{keep.memory} \land \text{term.transition}_5, \]
\[ ss_3 = \text{keep.memory} \land \text{term.transition}_6, \]
\[ ss_4 = \text{term.transition}_7 \land \text{keep.acc}, \]
\[ ss_5 = \text{term.transition}_8 \land \text{idle.is.false} \]

Here,
\[ \text{keep.memory} \triangleq \text{memory} \quad y = \text{memory} \quad x \]
\[ \text{keep.pc} \triangleq \text{pc} \quad y = \text{pc} \quad x \]
\[ \text{keep.acc} \triangleq \text{acc} \quad y = \text{acc} \quad x \]
\[ \text{idle.is.true} \triangleq \text{idle} \quad y = \text{T} \]
\[ \text{idle.is.false} \triangleq \text{idle} \quad y = \text{F} \]
\[ \text{term.transition}_0 \triangleq \text{pc} \quad y = \text{Cut}_{16..13} \quad (\text{switches} \quad x) \]
\[ \text{term.transition}_1 \triangleq \text{acc} \quad y = \text{switches} \quad x \]
\[ \text{term.transition}_2 \triangleq \text{memory} \quad y = \text{Store}_{13} \quad (\text{pc} \quad x) \quad (\text{acc} \quad x) \quad (\text{memory} \quad x) \]
\[ \text{term.transition}_3 \triangleq \text{pc} \quad y = \text{Cut}_{16..13} \quad (\text{v} \quad x) \]
\[ \text{term.transition}_4 \triangleq \text{acc} \quad y = \text{Add}_{16} \quad (\text{acc} \quad x) \quad (\text{Fetch}_{13} \quad (\text{memory} \quad x) \quad (\text{Cut}_{16..13} \quad (\text{v} \quad x))) \]
\[ \text{term.transition}_5 \triangleq \text{acc} \quad y = \text{Sub}_{16} \quad (\text{acc} \quad x) \quad (\text{Fetch}_{13} \quad (\text{memory} \quad x) \quad (\text{Cut}_{16..13} \quad (\text{v} \quad x))) \]
\[ \text{term.transition}_6 \triangleq \text{acc} \quad y = \text{Fetch}_{13} \quad (\text{memory} \quad x) \quad (\text{Cut}_{16..13} \quad (\text{v} \quad x)) \]
\[ \text{term.transition}_7 \triangleq \text{memory} \quad y = \text{Store}_{13} \quad (\text{Cut}_{16..13} \quad (\text{v} \quad x)) \quad (\text{acc} \quad x) \quad (\text{memory} \quad x) \]
\[ \text{term.transition}_8 \triangleq \text{pc} \quad y = \text{Inc}_{13} \quad (\text{pc} \quad x) \]

And we assume
\[ \text{term.transition}_9 \triangleq \text{v} \quad y = \text{Fetch}_{13} \quad (\text{memory} \quad x) \quad (\text{pc} \quad x) \]
Decomposing to Function Unit and Connection Level

In this subsubsection we construction a specification 6 from the specification 5 by decomposing those complex term transitions to function units and connections.

Before the decomposition we need a data refinement: for Inc\textsubscript{13} we invoke the extra axiom from [50] as below:

**Axiom 11.2.6.1. [for Inc\textsubscript{13}]:**

\[ \vdash \forall x.\text{Inc}_13 \ x = \text{Cut}_13 \ (\text{Inc}_16 \ (\text{Pad}_16 \ x)) \]
Therefore for the specification 5 its Inc\textsubscript{13} \( x \) will be replaced by

\[ \text{Cut}_13 \ (\text{Inc}_16 \ (\text{Pad}_16 \ x)) \].

In this construction step we introduce new entry predicates to break up each term transition term\_transition\textsubscript{1} \((0 \leq i \leq 9)\) and introduce new ports (as output and input ports). For example, for term\_transition\textsubscript{0} we introduce the entry predicates \( E_{0a} \) and \( E_{0b} \) and the new ports \( i_{0a} \) and \( o_{0a} \).

**Specification 6:**

\[
\begin{align*}
\text{term\_transition}_0 \ [E_6, E] &= \\
(i_{0a} y = \text{switches} \ x) \ [E_6, E_{0a}] \land \\
(o_{0a} y = \text{Cut}_13 (i_{0a} x)) \ [E_{0a}, E_{0b}] \land \\
(pc y = o_{0a} x) \ [E_{0b}, E] \\
\text{term\_transition}_1 \ [E_7, E] &= \\
\text{acc} y = \text{switches} \ x[E_7, E] \\
\text{term\_transition}_2 \ [E_8, E] &= \\
(i_{2a} y = \text{pc} \ x) \ [E_8, E_{2a}] \land \\
(i_{2b} y = \text{acc} \ x) \ [E_8, E_{2a}] \land \\
(i_{2c} y = \text{memory} \ x) \ [E_8, E_{2a}] \land \\
(o_{2a} y = \text{Store}_13 (i_{2a} x) (i_{2b} x) (i_{2c} x)) \ [E_{2a}, E_{2b}] \land \\
\text{memory} \ y = o_{2a} x [E_{2b}, E] \\
\text{term\_transition}_3 \ [E_{11}, E] &=
\end{align*}
\]
\[(i_{3a} \ y = v \ x) \ [E_{11}, \ E_{3a}] \land \]
\[(o_{3a} \ y = \text{Cut}_{16..13} \ (i_{3a} \ x)) \ [E_{3a}, \ E_{3b}] \land \]
\[(pc \ y = o_{3a} \ x) \ [E_{3b}, \ E] \]

\[(\text{term}_\text{transitions}_4) \ [E_{13}, \ E_{19}] = \]
\[(i_{4a} \ y = acc \ x) \ [E_{13}, \ E_{4a}] \land \]
\[(i_{4b} \ y = memory \ x) \ [E_{13}, \ E_{4b}] \land \]
\[(i_{4c} \ y = v \ x) \ [E_{13}, \ E_{4c}] \land \]
\[(o_{4a} \ y = \text{Cut}_{16..13} \ (i_{4c} \ x)) \ [E_{4c}, \ E_{4d}] \land \]
\[(i_{4d} \ y = o_{4a} \ x) \ [E_{4d}, \ E_{4b}] \land \]
\[(o_{4b} \ y = \text{Fetch}_{13} \ (i_{4b} \ x) \ (i_{4d} \ x)) \ [E_{4b}, \ E_{4e}] \land \]
\[(i_{4e} \ y = o_{4b} \ x) \ [E_{4e}, \ E_{4a}] \land \]
\[(o_{4c} \ y = \text{Add}_{16} \ (i_{4a} \ x) \ (i_{4e} \ x)) \ [E_{4a}, \ E_{4f}] \land \]
\[(acc \ y = o_{4c} \ x) \ [E_{4f}, \ E_{19}] \]

\[(\text{term}_\text{transitions}_5) \ [E_{14}, \ E_{19}] = \]
\[(i_{5a} \ y = acc \ x) \ [E_{14}, \ E_{5a}] \land \]
\[(i_{5b} \ y = memory \ x) \ [E_{14}, \ E_{5b}] \land \]
\[(i_{5c} \ y = v \ x) \ [E_{14}, \ E_{5c}] \land \]
\[(o_{5a} \ y = \text{Cut}_{16..13} \ (i_{5c} \ x)) \ [E_{5c}, \ E_{5d}] \land \]
\[(i_{5d} \ y = o_{5a} \ x) \ [E_{5d}, \ E_{5b}] \land \]
\[(o_{5b} \ y = \text{Fetch}_{13} \ (i_{5b} \ x) \ (i_{5d} \ x)) \ [E_{5b}, \ E_{5e}] \land \]
\[(i_{5e} \ y = o_{5b} \ x) \ [E_{5e}, \ E_{5a}] \land \]
\[(o_{5c} \ y = \text{Sub}_{16} \ (i_{5a} \ x) \ (i_{5e} \ x)) \ [E_{5a}, \ E_{5f}] \land \]
\[(acc \ y = o_{5c} \ x) \ [E_{5f}, \ E_{19}] \]

\[(\text{term}_\text{transitions}_6) \ [E_{15}, \ E_{19}] = \]
\[(i_{6a} \ y = memory \ x) \ [E_{15}, \ E_{6a}] \land \]
\[(i_{6b} \ y = v \ x) \ [E_{15}, \ E_{6b}] \]
\[(o_{6a} \ y = \text{Cut}_{16..13} \ (i_{6b} \ x)) \ [E_{6b}, \ E_{6c}] \land \]
\[(i_{6c} \ y = o_{6a} \ x) \ [E_{6c}, \ E_{6a}] \land \]
\[(o_{6b} \ y = \text{Fetch}_{13} \ (i_{6a} \ x) \ (i_{6c} \ x)) \ [E_{6a}, \ E_{6d}] \land \]
(acc y = o_{6b} x) [E_{6d}, E_{19}]

(term.transition_7) [E_{16}, E_{19}] =

(i_{7a} y = v x) [E_{16}, E_{7a}] \land

(o_{7a} y = \text{Cut}_{13} (i_{7a} x)) [E_{7a}, E_{7b}] \land

(i_{7b} y = o_{7a} x) [E_{7b}, E_{7c}] \land

(i_{7c} y = \text{acc} x) [E_{16}, E_{7c}] \land

(i_{7d} y = \text{memory} x) [E_{16}, E_{7c}] \land

(o_{7b} y = \text{Store}_{13} (i_{7b} x) (i_{7c} x) (i_{7d} x)) [E_{7c}, E_{7d}] \land

(memory y = o_{7b} x) [E_{7d}, E_{19}]

(term.transition_8) [E_{19}, E] =

(i_{8a} y = \text{pc} x) [E_{19}, E_{8a}] \land

(o_{8a} y = \text{Pad}_{16} (i_{8a} x)) [E_{8a}, E_{8b}] \land

(i_{8b} y = o_{8a} x) [E_{8b}, E_{8c}] \land

(o_{8b} y = \text{Inc}_{16} (i_{8b} x)) [E_{8c}, E_{8d}] \land

(i_{8c} y = o_{8b} x) [E_{8d}, E_{8e}] \land

(o_{8c} y = \text{Cut}_{13} (i_{8c} x)) [E_{8e}, E_{8f}] \land

(pc y = o_{8c} x) [E_{8f}, E]

(term.transition_9) [E_{5}, E_{10}] =

(i_{9a} y = \text{memory} x) [E_{5}, E_{9a}] \land

(i_{9b} y = \text{pc} x) [E_{5}, E_{9b}] \land

(o_{9a} y = \text{Fetch}_{13} (i_{9a} x) (i_{9b} x)) [E_{9a}, E_{9b}] \land

(v y = o_{9a} x) [E_{9b}, E_{10}]

Allocation

In the specification 6 we have not allocate devices to function units. In this subsubsection we do for this. Meanwhile we will treat register like as function unit as they have some non-register functions. We introduce two device kinds:

Mem (read, write, addr, date, mem, memout)
\[\equiv \forall t. (\text{memout } t = (\text{read } t \Rightarrow \text{Fetch}_{13} (\text{mem } t) (\text{addr} s \ t) \downarrow \text{Float}_{16}))\]
\[\land \]
\[\ (\text{mem } (t + 1) = (\text{write } t \Rightarrow \text{Store}_{13} (\text{addr} s \ t) (\text{data } t) (\text{mem } t) \downarrow \text{mem } t))\]
\[\text{Alu } (\text{add}, \text{sub}, \text{inc}, i_1, i_2, o)\]
\[\equiv \forall t. \ o \ t = (\text{add } t \Rightarrow \text{Add}_{16} (i_1 \ t) (i_2 \ t) \downarrow\]
\[\sub t \Rightarrow \text{Sub}_{16} (i_1 \ t) (i_2 \ t) \downarrow\]
\[\text{inc } t \Rightarrow \text{Inc}_{16} (i_2 \ t) \downarrow \text{Float}_{16})\]

Correspondingly, in the specification 7 we introduce device instances:

\[\text{Mem } (r_{\text{mem}}, \ w_{\text{mem}}, \ \text{addr}, \ \text{data}, \ \text{mem}, \ \text{mout})\]
\[\text{Alu } (\text{add}, \sub, \text{inc}, \text{ain} 1, \text{ain2}, \text{aout})\]

And four registers:

\[\text{Reg } (\text{w acc}, \ \text{i acc}, \ \text{acc})\]
\[\text{Reg} (\text{w pc}, \ \text{i pc}, \ \text{o pc})\]
\[\text{Reg } (\text{w ir}, \ \text{i ir}, \ \text{v})\]
\[\text{Reg} (\text{w mar}, \ \text{i mar}, \ \text{o mar})\]

Reg and Reg are two different registers:

\[\text{Reg } (c, \ i, \ o) \equiv \forall t. \ o \ (t + 1) = (c \ t \Rightarrow i \ t \downarrow o \ t)\]
\[\text{Reg} (c, \ i, \ o) \equiv \forall t. \ o \ (t + 1) = (c \ t \Rightarrow \text{Cut}_{16,13} (i \ t) \downarrow o \ t)\]

We still keep the arguments of these devices as variables until the last step to instantiating them to get an implementation.

In the construction we decompose the models formed as

\[(\text{acc } y = \alpha) \ [E_i, \ E_o]\]
\[(\text{v } y = \beta) \ [E_i, \ E_o]\]

to

\[(\text{i acc } y = \alpha) \ [E_i, \ E_m] \land (\text{acc } y = i_{\text{acc}} \ x) \ [E_m, \ E_o]\]
\[(\text{i ir } y = \beta) \ [E_i, \ E_m] \land (\text{v } y = i_{\text{ir}} \ x) \ [E_m, \ E_o]\]

This is because in the specification 4 and 5 the state signal acc and the signal v
appear in delay units with time constraint \( x < y \) (the constraint must be taken as, at least, the realistic devices require a time unit delay) and they must be selected as the output of registers.

**Specification 7:**

\[
(t_{\text{term\_transition}0})[E_6,E] = \\
(i_{\text{pc}}\ y = \text{switches } x)\ [E_6, E_{0a}] \land \\
(o_{\text{pc}}\ y \overset{\text{t}}{=} \text{Cut}_{113} (i_{\text{pc}} \ x))\ [E_{0a}, E_{0b}] \land \\
(pc\ y = o_{\text{pc}} \ x)\ [E_{0b}, E]
\]

\[
(t_{\text{term\_transition}1})[E_7,E] = \\
i_{\text{acc}}\ y = \text{switches } x[E_7, E_{7a}] \land \\
acc\ y \overset{\text{t}}{=} i_{\text{acc}} \ x[E_{7a}, E]
\]

\[
(t_{\text{term\_transition}2})[E_8, E] = \\
(addr\ y = pc \ x)\ [E_8, E_{2a}] \land \\
data\ y = \text{acc } x)\ [E_8, E_{2a}] \land \\
(mem\ y = \text{memory } x)\ [E_8, E_{2a}] \land \\
(mem\ y \overset{\text{m}}{=} \text{Store}_{13} (addr \ x) \ (data \ x) \ (mem \ x))\ [E_{2a}, E_{2b}] \land \\
(m\ y = \text{mem } x)\ [E_{2b}, E]
\]

\[
(t_{\text{term\_transition}3})[E_{11}, E] = \\
i_{\text{pc}}\ y = \nu \ x)\ [E_{11}, E_{3a}] \land \\
(o_{\text{pc}}\ y \overset{\text{t}}{=} \text{Cut}_{16\ldots13} (i_{\text{pc}} \ x))\ [E_{3a}, E_{3b}] \land \\
(pc\ y = o_{\text{pc}} \ x)\ [E_{3b}, E]
\]

\[
(t_{\text{term\_transition}4})[E_{13}, E_{19}] = \\
(\text{ain1 } y = \text{acc } x)\ [E_{13}, E_{4a}] \land \\
(mem\ y = \text{memory } x)\ [E_{13}, E_{4b}] \land \\
i_{\text{mar}}\ y = \nu \ x)\ [E_{13}, E_{4c}] \land \\
o_{\text{mar}}\ y \overset{\text{m}}{=} \text{Cut}_{16\ldots13} (i_{\text{mar}} \ x))\ [E_{4c}, E_{4d}] \land \\
(addr\ y = o_{\text{mar}} \ x)\ [E_{4d}, E_{4b}] \land \\
(m\ y \overset{\text{m}}{=} \text{Fetch}_{13} (mem \ x) \ (addr \ x))\ [E_{4b}, E_{4e}] \land \\
(\text{ain2 } y = \text{mout } x)\ [E_{4e}, E_{4a}] \land
\]
(aout y ≜ Add_16 (ain1 x) (ain2 x)) [E_{4a}, E_{4f}] \land
(i_{acc} y = aout x) [E_{4f}, E_{4g}] \land
(acc y \models i_{acc} x) [E_{4g}, E_{19}]

(term_transitions_5) [E_{14}, E_{19}] =
(a1n1 y = acc x) [E_{14}, E_{5a}] \land
(mem y = memory x) [E_{14}, E_{5b}] \land
(i_{mar} y = v x) [E_{14}, E_{5c}] \land
(o_{mar} y \models Cut_{16-13} (i_{mar} x)) [E_{5c}, E_{5d}] \land
(addr y = o_{mar} x) [E_{5d}, E_{5e}] \land
(mout y \models Fetch_{13} (mem x) (addr x)) [E_{5b}, E_{5e}] \land
(a1n2 y = mout x) [E_{5e}, E_{5a}] \land
(aout y \equiv Sub_{16} (a1n x) (a1n2 x)) [E_{5a}, E_{5f}] \land
(i_{acc} y = aout x) [E_{5f}, E_{5g}] \land
(acc y \models i_{acc} x) [E_{5g}, E_{19}]

(term_transitions_6) [E_{15}, E_{19}] =
(mem y = memory x) [E_{15}, E_{6a}] \land
(i_{mar} y = v x) [E_{15}, E_{6b}] \land
(o_{mar} y \models Cut_{16-13} (i_{mar} x)) [E_{6b}, E_{6c}] \land
(addr y = o_{mar} x) [E_{6c}, E_{6a}] \land
(mout y \models Fetch_{13} (mem x) (addr x)) [E_{6a}, E_{6d}] \land
(i_{acc} y = mout x) [E_{6d}, E_{6e}] \land
(acc y \models i_{acc} x) [E_{6e}, E_{19}]

(term_transitions_7) [E_{16}, E_{19}] =
(i_{mar} y = v x) [E_{16}, E_{7a}] \land
(o_{mar} y \models Cut_{13} (i_{mar} x)) [E_{7a}, E_{7b}] \land
(addr y = o_{mar} x) [E_{7b}, E_{7c}] \land
(data y = acc x) [E_{16}, E_{7c}] \land
(mem y = memory x) [E_{16}, E_{7c}] \land
(mem y \models Store_{13} (addr x) (data x) (mem x)) [E_{7c}, E_{7d}] \land
(memory \( y = \text{mem} \ x \)) \([E_{7\theta}, \ E_{19}]\)

\(\text{term\_transitions}_8 \ [E_{19}, \ E] =\)
\(\ (i_{8\theta} \ y = \text{pc} \ x \) \ [E_{19}, \ E_{8\theta}] \wedge\)
\(\ (o_{8\theta} \ y = \text{Pad}_{16} \ (i_{8\theta} \ x) \) \ [E_{8\theta}, \ E_{8\theta}] \wedge\)
\(\ (\text{ain2} \ y = o_{8\theta} \ x \) \ [E_{8\theta}, \ E_{8c}] \wedge\)
\(\ (\text{aout} \ y \ \overset{\eta}{=} \text{Inc}_{16} \ (\text{ain2} \ x) \) \ [E_{8c}, \ E_{8d}] \wedge\)
\(\ (i_{pc} \ y = \text{aout} \ x \) \ [E_{8d}, \ E_{8e}] \wedge\)
\(\ (o_{pc} \ y \overset{\text{r}_{\theta}}{=} \text{Cut}_{13} \ (i_{pc} \ x) \) \ [E_{8e}, \ E_{8f}] \wedge\)
\(\ (\text{pc} \ y = o_{pc} \ x \) \ [E_{8f}, \ E]\)

\(\text{term\_transitions}_9 \ [E_{5}, \ E_{10}] =\)
\(\ (\text{mem} \ y = \text{memory} \ x \) \ [E_{5}, \ E_{9a}] \wedge\)
\(\ (\text{addr} \ y = \text{pc} \ x \) \ [E_{5}, \ E_{9a}] \wedge\)
\(\ (\text{mout} \ y \overset{\text{m}}{=} \text{Fetch}_{13} \ (\text{mem} \ x) \ (\text{addr} \ x) \) \ [E_{9a}, \ E_{9b}] \wedge\)
\(\ (i_{ir} \ y = \text{mout} \ x \) \ [E_{9b}, \ E_{9c}] \wedge\)
\(\ (v \ y \overset{\text{r}_{\theta}}{=} i_{ir} \ x \) \ [E_{9c}, \ E_{10}]\)

**Scheduling**

In specification 8 we give the result of scheduling for \(\text{term\_transition}_i \quad (0 \leq i \leq 9)\). The scheduling is somewhat simple as the main work is to introduce delays for state signal \text{memory} and \text{acc}.

**Specification 8:**

\(\text{term\_transitions}_0 \ [E_{6}, \ E] =\)
\(\ (i_{pc} \ y = \text{switches} \ x \) \ [E_{6}, \ E_{0a}] \wedge\)
\(\ (o_{pc} \ y \overset{\text{r}_{\theta}}{=} \text{Cut}_{13} \ (i_{pc} \ x) \) \ [E_{0a}, \ E_{0b}] \wedge\)
\(\ (\text{pc} \ y = o_{pc} \ x \) \ [E_{0b}, \ E]\)

\(\text{term\_transitions}_1 \ [E_{7}, \ E] =\)
\(\ i_{acc} \ y = \text{switches} \ x[\ E_{7}, \ E_{7a}] \wedge\)
\(\ \text{acc} \ y \overset{\text{r}_{\theta}}{=} i_{acc} \ x[\ E_{7a}, \ E] \)
(term.transition2) \([E_8, E] = \)

\((\text{addr} y = \text{pc} x \land \text{data} y = \text{acc} x \land \text{mem} y = \text{memory} x) [E_8, E_{2a}] \land\)

\((\text{mem} y \leftarrow \text{Store}_{13} (\text{addr} x) (\text{data} x) (\text{mem} x)) [E_{2a}, E_{2b}] \land\)

\((\text{memory} y = \text{mem} x) [E_{2b}, E]\)

(term.transition3) \([E_{11}, E] = \)

\((i_{pc} y = v x) [E_{11}, E_{3a}] \land\)

\((o_{pc} y \leftarrow \text{Cut}_{16..13} (i_{pc} x)) [E_{3a}, E_{3b}] \land\)

\((pc y = o_{pc} x) [E_{3b}, E]\)

(term.transition4) \([E_{13}, E_{19}] = \)

\((i_{mar} y = v x \land \text{memory} y = \text{memory} x \land \text{acc} y = \text{acc} x) [E_{13}, E_{4c}] \land\)

\((o_{mar} y \leftarrow \text{Cut}_{16..13} (i_{mar} x) \land\)

\(\text{memory} y = \text{memory} x \land \text{acc} y = \text{acc} x) [E_{4c}, E_{4d}] \land\)

\((\text{addr} y = o_{mar} x \land \text{mem} y = \text{memory} x \land \text{acc} y = \text{acc} x) [E_{4d}, E_{4b}] \land\)

\((\text{mout} y \leftarrow \text{Fetch}_{13} (\text{mem} x) (\text{addr} x) \land \text{acc} y = \text{acc} x) [E_{4b}, E_{4e}] \land\)

\((\text{ain2} y = \text{mout} x \land \text{ain1} y = \text{acc} x) [E_{4e}, E_{4a}] \land\)

\((\text{aout} y \leftarrow \text{Add}_{16} (\text{ain1} x) (\text{ain2} x)) [E_{4a}, E_{4f}] \land\)

\((i_{acc} y = \text{aout} x) [E_{4f}, E_{4g}] \land\)

\((\text{acc} y \leftarrow i_{acc} x) [E_{4g}, E_{19}]\)

(term.transition5) \([E_{14}, E_{19}] = \)

\((i_{mar} y = v x \land \text{memory} y = \text{memory} x \land \text{acc} y = \text{acc} x) [E_{14}, E_{5c}] \land\)

\((o_{mar} y \leftarrow \text{Cut}_{16..13} (i_{mar} x) \land\)

\(\text{memory} y = \text{memory} x \land \text{acc} y = \text{acc} x) [E_{5c}, E_{5d}] \land\)

\((\text{addr} y = o_{mar} x \land \text{mem} y = \text{memory} x \land \text{acc} y = \text{acc} x) [E_{5d}, E_{5b}] \land\)

\((\text{mout} y \leftarrow \text{Fetch}_{13} (\text{mem} x) (\text{addr} x) \land \text{acc} y = \text{acc} x) [E_{5b}, E_{5e}] \land\)

\((\text{ain2} y = \text{mout} x \land \text{ain1} y = \text{acc} x) [E_{5e}, E_{5a}] \land\)

\((\text{aout} y \leftarrow \text{Sub}_{16} (\text{ain1} x) (\text{ain2} x)) [E_{5a}, E_{5f}] \land\)

\((i_{acc} y = \text{aout} x) [E_{5f}, E_{5g}] \land\)

\((\text{acc} y \leftarrow i_{acc} x) [E_{5g}, E_{19}]\)

(term.transition6) \([E_{15}, E_{19}] = \)
\((i_{mar} \ y = \ v \ x \land \ memory \ y = \ memory \ x) \ [E_{15}, \ E_{6b}] \land\)
\((o_{mar} \ y \overset{r}{=} \text{Cut}_{16,13} \ (i_{mar} \ x) \land \ memory \ y = \ memory \ x) \ [E_{6b}, \ E_{6c}] \land\)
\((\text{addr} \ y = o_{mar} \ x \land \ mem \ y = \ memory \ x) \ [E_{6c}, \ E_{6a}] \land\)
\((\text{mout} \ y \overset{m}{=} \text{Fetch}_{13} \ (\text{mem} \ x) \ (\text{addr} \ x)) \ [E_{6a}, \ E_{6d}] \land\)
\((i_{acc} \ y = \text{mout} \ x) \ [E_{6d}, \ E_{6e}] \land\)
\((acc \ y \overset{r}{=} \ i_{acc} \ x) \ [E_{6e}, \ E_{19}]\)

\((\text{term}_\text{transitions}_7) \ [E_{16}, \ E_{19}] =\)
\((i_{mar} \ y = \ v \ x \land \ memory \ y = \ memory \ x \land \ acc \ y = \ acc \ x) \ [E_{16}, \ E_{7a}] \land\)
\((o_{mar} \ y \overset{r}{=} \text{Cut}_{13} \ (i_{mar} \ x) \land\)
\(memory \ y = \ memory \ x \land \ acc \ y = \ acc \ x) \ [E_{7a}, \ E_{7b}] \land\)
\((\text{addr} \ y = o_{mar} \ x \land \ data \ y = \ acc \ x \land \ mem \ y = \ memory \ x) \ [E_{7b}, \ E_{7c}] \land\)
\((\text{mem} \ y \overset{m}{=} \text{Store}_{13} \ (\text{addr} \ x) \ (\text{data} \ x) \ (\text{mem} \ x)) \ [E_{7c}, \ E_{7d}] \land\)
\((memory \ y = \ mem \ x) \ [E_{7d}, \ E_{19}]\)

\((\text{term}_\text{transitions}_8) \ [E_{19}, \ E] =\)
\((i_{8a} \ y = \ pc \ x) \ [E_{19}, \ E_{8a}] \land\)
\((o_{8a} \ y = \text{Pad}_{16} \ (i_{8a} \ x)) \ [E_{8a}, \ E_{8b}] \land\)
\((a_{in2} \ y = o_{8a} \ x) \ [E_{8b}, \ E_{8c}] \land\)
\((a_{out} \ y \overset{a}{=} \text{Inc}_{16} \ (a_{in2} \ x)) \ [E_{8c}, \ E_{8d}] \land\)
\((i_{pc} \ y = \text{aout} \ x) \ [E_{8d}, \ E_{8e}] \land\)
\((o_{pc} \ y \overset{r}{=} \text{Cut}_{13} \ (i_{pc} \ x)) \ [E_{8e}, \ E_{8f}] \land\)
\((pc \ y = o_{pc} \ x) \ [E_{8f}, \ E]\)

\((\text{term}_\text{transitions}_9) \ [E_{5}, \ E_{10}] =\)
\((\text{mem} \ y = \ memory \ x \land \ addr \ y = \ pc \ x) \ [E_{5}, \ E_{9a}] \land\)
\((\text{mout} \ y \overset{m}{=} \text{Fetch}_{13} \ (\text{mem} \ x) \ (\text{addr} \ x)) \ [E_{9a}, \ E_{9b}] \land\)
\((i_{ir} \ y = \text{mout} \ x) \ [E_{9b}, \ E_{9c}]\)
\((v \ y \overset{t}{=} \ i_{ir} \ x) \ [E_{9c}, \ E_{10}]\)

The specification 8 has established the main structure of the implementation of Gordon's computer because from the specification 4 and 5 we can see all other parts only are about delay unit and branch structures.
11.2.7 Introducing Bus, Connection and Microinstruction

Introducing Bus, Gate and Connection

Following the specification we introduce bus, gate and connection.

We introduce a group of Buses and Gates (see page 118):

Bus \( (o_{11}, o_{12}, o_{13}, i_{ace}) \) \( \cdots \) called Bus1 \( \cdots \) (r1)
Bus \( (o_{11}, o_{12}, o_{23}, i_{pc}) \) \( \cdots \) called Bus2 \( \cdots \) (r2)
Bus \( (o_{31}, o_{32}, i_{a2}) \) \( \cdots \) called Bus3 \( \cdots \) (r3)
Bus \( (o_{32}, v, i_{u}) \) \( \cdots \) called Bus4 \( \cdots \) (r4)
Gate \( (rsw, switches, o_{11}) \) \( \cdots \) (Gate1)
Gate \( (ralu, o_{u}, o_{12}) \) \( \cdots \) (Gate2)
Gate \( (rm_1, o_m, o_{13}) \) \( \cdots \) (Gate3)
Gate \( (rir, v, o_{23}) \) \( \cdots \) (Gate4)
Gate\(_p\) \( (rm_3, o_{m}, o_{31}) \) \( \cdots \) (Gate5)
Gate\(_p\) \( (rpc, pc, o_{32}) \) \( \cdots \) (Gate6)

Note: above we write \( o_u \) and \( o_m \) for \( aout \) and \( mout \) respectively.

Here, we introduce a new type gate:

Gate\(_p\) \( (c, i, o) \equiv \forall t. o \; t = (c \; t \Rightarrow \text{Mk	extunderscore{}Tri} (\text{Pad}_{16\textunderscore{}13} (i \; t)) \downarrow \text{Float}_{16}) \cdots \) (gs\(_p\))

To introduce connection we use \( pc, memory, o_{mar}, acc \) and \( mout \) to replace \( o_{pc}, mem, \text{addr}, a1 \) and \( i_{ir} \) respectively. Meanwhile when introduce a connection we make time interval become zero (instantiating \( y \) by \( x \))

As the function Fetch\(_{13}\) and the device Alu takes zero time interval to complete their functions we can make the logic boundary of their corresponding models become to have the same entry and leaving predicates and the same begin and end time points (instantiating \( y \) by \( x \)).

For the connection link which has been selected for Bus its logic boundary is also constructed with the same entry and leaving predicate and zero time interval.
For example

\[(i_{pc} y \xrightarrow{b} \text{switches } x) [E_6, E_{0a}]\]

is constructed to

\[(i_{pc} x \xrightarrow{b} \text{switches } x) [E_6, E_6].\]

Here, \(E_{0a}\) is replaced by \(E_6\) and \(y\) is replaced by \(x\).

A special case is that we replace

\[(i_{8a} y = pc x) [E_{19}, E_{8a}] \land\]
\[(o_{8a} y = \text{Pad}_{16} (i_{8a} x)) [E_{8a}, E_{8b}] \land\]
\[(ain2 y = o_{8a} x) [E_{8b}, E_{8c}]\]

by

\[(ain2 x \xrightarrow{b} \text{Pad}_{16} (pc x)) [E_{19}, E_{19}]\]

because we need to introduce a special \(\text{Gate}_p\) for Bus [50].

As \(\text{addr}\) is replaced by \(o_{\text{mar}}\) which is the output of register \(\text{Mar}\) in \(\text{term.transition}_2\) and \(\text{term.transition}_9\) we replace

\[(\alpha_1 \land \text{addr } y = \text{Pad}_{16..13} (pc x) \land \alpha_2) [E_{\beta}, E_{\gamma}]\]

by

\[\left(\alpha_1 \land \text{mar } x \xrightarrow{b} pc x \land \alpha_2\right) [E_{\beta}, E_{\beta}] \text{ (Again, Gate}_p\text{ for Bus)}\]
\[\land\]
\[\left(\alpha_1 \land o_{\text{mar}} x \xrightarrow{r} \text{Cut}_{16..13} (i_{\text{mar}} x) \land \alpha_2\right) [E_{\beta}, E_{\gamma}]\]

**Specification 9:**

\[(\text{term.transition}_0) [E_6, E] =\]
\[\left(i_{ipe} x \xrightarrow{b} \text{switches } x\right) [E_6, E_6] \land\]
\[\left(pc y \xrightarrow{r} \text{Cut}_{13} (i_{pc} x)\right) [E_6, E] \land\]
\[\left(pc x \xrightarrow{c} pc x\right) [E, E]\]

\[(\text{term.transition}_1) [E_7, E] =\]
\[\left(i_{acc} x \xrightarrow{b} \text{switches } x\right) [E_7, E_7] \land\]
\[(\text{acc } y \xrightarrow{r} \text{iacc } x) [E_7, E]\]

\[\text{(term.transitions)}_2 [E_8, E] = \]
\[(i_{\text{mar}} \ x \xrightarrow{b} \text{Pad}_{16,13} (pc \ x) \land \text{acc } x = \text{acc } x \land \text{memory } x = \text{memory } x) [E_8, E_8] \land\]
\[(o_{\text{mar}} \ y \xrightarrow{r} \text{Cut}_{16,13} (i_{\text{mar}} \ x) \land\]
\[\text{acc } y = \text{acc } x \land \text{memory } y = \text{memory } x) [E_8, E_{2a}] \land\]
\[(\text{memory } y \xrightarrow{m} \text{Store}_{13} (o_{\text{mar}} \ x) (\text{acc } x) (\text{memory } x)) [E_{2a}, E] \land\]
\[(\text{memory } x \xrightarrow{c} \text{memory } x) [E, E]\]

\[\text{(term.transitions)}_3 [E_{11}, E] = \]
\[(i_{pc} \ x \xrightarrow{b} v \ x) [E_{11}, E_{11}] \land\]
\[(pc \ y \xrightarrow{r} \text{Cut}_{16,13} (i_{pc} \ x)) [E_{11}, E] \land\]
\[(pc \ x \xrightarrow{c} pc \ x) [E, E]\]

\[\text{(term.transitions)}_4 [E_{13}, E_{19}] = \]
\[(i_{\text{mar}} \ x \xrightarrow{b} v \ x \land \text{memory } x = \text{memory } x \land \text{acc } x = \text{acc } x) [E_{13}, E_{13}] \land\]
\[(o_{\text{mar}} \ y \xrightarrow{r} \text{Cut}_{16,13} (i_{\text{mar}} \ x) \land\]
\[\text{memory } y = \text{memory } x \land \text{acc } y = \text{acc } x) [E_{13}, E_{4d}] \land\]
\[(o_{\text{mar}} \ x \xrightarrow{c} o_{\text{mar}} \ x \land \text{memory } x = \text{memory } x \land \text{acc } x = \text{acc } x) [E_{4d}, E_{4d}] \land\]
\[(\text{mout } x \xrightarrow{m} \text{Fetch}_{13} (\text{memory } x) (o_{\text{mar}} \ x) \land \text{acc } x = \text{acc } x) [E_{4d}, E_{4d}] \land\]
\[(\text{ain2 } x \xrightarrow{b} \text{mout } x \land \text{acc } x \xrightarrow{c} \text{acc } x) [E_{4d}, E_{4d}] \land\]
\[(\text{aout } x \xrightarrow{a} \text{Add}_{16} (\text{acc } x) (\text{ain2 } x)) [E_{4d}, E_{4d}] \land\]
\[(\text{iacc } x \xrightarrow{b} \text{aout } x) [E_{4d}, E_{4d}] \land\]
\[(\text{acc } y \xrightarrow{r} \text{iacc } x) [E_{4d}, E_{19}]\]

\[\text{(term.transitions)}_5 [E_{14}, E_{19}] = \]
\[(i_{\text{mar}} \ x \xrightarrow{b} v \ x \land \text{memory } x = \text{memory } x \land \text{acc } x = \text{acc } x) [E_{14}, E_{14}] \land\]
\[(o_{\text{mar}} \ y \xrightarrow{r} \text{Cut}_{16,13} (i_{\text{mar}} \ x) \land\]
\[\text{memory } y = \text{memory } x \land \text{acc } y = \text{acc } x) [E_{14}, E_{5d}] \land\]
\[(o_{\text{mar}} \ x \xrightarrow{c} o_{\text{mar}} \ x \land \text{memory } x = \text{memory } x \land \text{acc } x = \text{acc } x) [E_{5d}, E_{5d}] \land\]
\[(\text{mout } x \xrightarrow{m} \text{Fetch}_{13} (\text{memory } x) (o_{\text{mar}} \ x) \land \text{acc } x \xrightarrow{c} \text{acc } x) [E_{5d}, E_{5d}] \land\]
\[(\text{ain2 } x \xrightarrow{b} \text{mout } x \land \text{acc } x = \text{acc } x) [E_{5d}, E_{5d}] \land\]
\[(\text{aout } x \xrightarrow{a} \text{Sub}_{16} (\text{acc } x) (\text{ain2 } x)) [E_{5d}, E_{5d}] \land\]
(i_{acc} \ x \stackrel{b_{1}}{=} \ aout \ x) \ [E_{5d}, \ E_{5d}] \land \\
(\text{acc} \ y \stackrel{r_{a}}{=} \ i_{acc} \ x) \ [E_{5d}, \ E_{19}] \\
(\text{term transitions}_{6}) \ [E_{15}, \ E_{19}] = \\
(i_{mar} \ x \stackrel{b_{2}}{=} \ v \ x \land \text{memory} \ x = \text{memory} \ x) \ [E_{15}, \ E_{16}] \land \\
(o_{mar} \ y \stackrel{r_{a}}{=} \text{Cut}_{16-13} (i_{mar} \ x) \land \text{memory} \ y = \text{memory} \ x) \ [E_{15}, \ E_{6c}] \land \\
(o_{mar} \ x \stackrel{c}{=} \ o_{mar} \ x \land \text{memory} \ x = \text{memory} \ x) \ [E_{6c}, \ E_{6c}] \land \\
(mout \ x \stackrel{m}{=} \text{Fetch}_{13} (\text{memory} \ x) \ (o_{mar} \ x)) \ [E_{6c}, \ E_{6c}] \land \\
(i_{acc} \ x \stackrel{b_{1}}{=} \ mout \ x) \ [E_{6c}, \ E_{6c}] \land \\
(\text{acc} \ y \stackrel{r_{a}}{=} \ i_{acc} \ x) \ [E_{6c}, \ E_{19}] \\
(\text{term transitions}_{7}) \ [E_{16}, \ E_{19}] = \\
(i_{mar} \ x \stackrel{b_{2}}{=} \ v \ x \land \text{memory} \ x = \text{memory} \ x \land \text{acc} \ x = \text{acc} \ x) \ [E_{16}, \ E_{16}] \land \\
(o_{mar} \ y \stackrel{r_{a}}{=} \text{Cut}_{13} (i_{mar} \ x) \land \\
\text{memory} \ y = \text{memory} \ x \land \text{acc} \ y = \text{acc} \ x) \ [E_{16}, \ E_{7b}] \land \\
(o_{mar} \ x \stackrel{c}{=} \ o_{mar} \ x \land \text{acc} \ x \stackrel{c}{=} \text{acc} \ x \land \text{memory} \ x \stackrel{c}{=} \text{memory} \ x) \ [E_{7b}, \ E_{7b}] \land \\
(memory \ y \stackrel{m}{=} \text{Store}_{13} (o_{mar} \ x) (\text{acc} \ x) (\text{memory} \ x)) \ [E_{7b}, \ E_{19}] \land \\
(memory \ y \stackrel{c}{=} \text{memory} \ x) \ [E_{19}, \ E_{19}] \\
(\text{term transitions}_{8}) \ [E_{19}, \ E] = \\
(ain2 \ x \stackrel{b_{2}}{=} \text{Pad}_{16} (pc \ x)) \ [E_{19}, \ E_{19}] \land \\
(aout \ x \stackrel{a}{=} \text{Inc}_{16} (ain2 \ x)) \ [E_{19}, \ E_{19}] \land \\
(i_{pc} \ x \stackrel{b_{2}}{=} \ aout \ x) \ [E_{19}, \ E_{19}] \land \\
(pc \ y \stackrel{r_{a}}{=} \text{Cut}_{13} (i_{pc} \ x)) \ [E_{19}, \ E] \land \\
(pc \ x \stackrel{c}{=} \ pc \ x) \ [E, \ E] \\
(\text{term transitions}_{9}) \ [E_{5}, \ E_{10}] = \\
(memory \ x \stackrel{c}{=} \text{memory} \ x \land i_{mar} \ x \stackrel{b_{2}}{=} \text{Pad}_{16-13} (pc \ x)) \ [E_{5}, \ E_{5}] \land \\
(memory \ y = \text{memory} \ x \land o_{mar} \ y \stackrel{r_{a}}{=} \text{Cut}_{16-13} (i_{mar} \ x)) \ [E_{5}, \ E_{9a}] \land \\
(mout \ y \stackrel{m}{=} \text{Fetch}_{13} (\text{memory} \ x) \ (o_{mar} \ x)) \ [E_{9a}, \ E_{9a}] \land \\
(mout \ y \stackrel{c}{=} \ mout \ x) \ [E_{9a}, \ E_{9a}] \land \\
(v \ y \stackrel{r_{a}}{=} \ mout \ x) \ [E_{9a}, \ E_{10}]
A more technical detail is about the structure of group of Bus and Gates which are connected each other as a fixed module. In principle a Gate is only used for a Bus. But for saving device recourse it is possible to use a Gate for two or more Bus.

For example, we must use two Gates: Gate3: Gate \((rm_1, o_m, o_{13})\) for the Bus1 connecting \(mout\) to \(i_{acc}\) in \(term\_transition_6\) and Gate5: Gate \((rm_3, o_m, o_{31})\) for the Bus3 connecting \(mout\) to \(ain2\) in the \(term\_transition_4\) and \(term\_transition_5\) respectively. Otherwise, namely we use one Gate, let suppose its control signal is \(rm\) to replace \(rm_1\) and \(rm_3\), for the two Buses, it can lead to an inconsistency. By \(term\_transition_5\) we get one \(i_{acc}\), however, by \(term\_transition_6\) we can also get another \(i_{acc}\) simultaneously.

However, we can use one Gate for two Buses, for example Gate1 for Bus1 and Bus2 as this does not lead to inconsistency.

**Introducing Abstract Microinstruction**

In this subsection we clean up the specification 9 by combine models such that a model can be just corresponding to an abstract microinstruction.

**Specification 10:**

\[
\text{\( (term\_transition_0) [E_6, E] = \)}
\]

\[
(i_{pc} x \overset{b_2}{\Rightarrow} \text{switches } x) \land
\]

\[
pc y \overset{r_y}{\Rightarrow} \text{Cut}_{13} (i_{pc} x)
\]

\[
) [E_6, E]
\]

\[
\text{\( (term\_transition_1) [E_7, E] = \)}
\]

\[
(i_{acc} x \overset{b_1}{\Rightarrow} \text{switches } x) \land
\]

\[
acc y \overset{r_a}{\Rightarrow} i_{acc} x
\]

\[
) [E_7, E]
\]

\[
\text{\( (term\_transition_2) [E_8, E] = \)}
\]
\[(i_{mar} \ x \ \overset{b}{\rightarrow} \ \text{Pad}_{16 \cdot 13} (pc \ x) \ \wedge \ acc \ x = acc \ x \wedge \text{memory} \ x = \text{memory} \ x) \ \wedge \]
\[o_{mar} \ y \ \overset{r}{\rightarrow} \ \text{Cut}_{16 \cdot 13} (i_{mar} \ x) \ \wedge \]
\[acc \ y = acc \ x \wedge \text{memory} \ y = \text{memory} \ x \]
\)[E_8, E_{2a}] \wedge
\]
\[(\text{memory} \ y \ \overset{m}{=} \ \text{Store}_{13} (o_{mar} \ x) (acc \ x) (\text{memory} \ x) \]
\)[E_{2a}, E]
\]
\[(\text{term.transition}_3) \ [E_{11}, E] = \]
\[(i_{pc} \ x \ \overset{b_2}{\rightarrow} \ v \ x \wedge \]
\[pc \ y \ \overset{r}{\rightarrow} \ \text{Cut}_{16 \cdot 13} (i_{pc} \ x) \]
\)[E_{11}, E]
\]
\[(\text{term.transition}_4) \ [E_{13}, E_{19}] = \]
\[(i_{mar} \ x \ \overset{b}{\rightarrow} \ v \ x \wedge \text{memory} \ x = \text{memory} \ x \wedge \ acc \ x = acc \ x \wedge \]
\[o_{mar} \ y \ \overset{r}{\rightarrow} \ \text{Cut}_{16 \cdot 13} (i_{mar} \ x) \wedge \]
\[\text{memory} \ y = \text{memory} \ x \wedge \ acc \ y = acc \ x \]
\)[E_{13}, E_{4a}] \wedge
\]
\[(\text{mout} \ x \ \overset{m}{=} \ \text{Fetch}_{13} (\text{memory} \ x) (o_{mar} \ x) \wedge \ acc \ x = acc \ x \wedge \]
\[a_{in2} \ x \ \overset{b_2}{\rightarrow} \ \text{mout} \ x \wedge \ acc \ x \overset{c}{=} \ acc \ x \wedge \]
\[a_{out} \ x \ \overset{a}{=} \ \text{Add}_{16} (acc \ x) (a_{in2} \ x) \wedge \]
\[i_{acc} \ x \ \overset{b_1}{\rightarrow} \ a_{out} \ x \wedge \]
\[acc \ y \ \overset{r}{\rightarrow} \ i_{acc} \ x \]
\)[E_{4a}, E_{19}]
\]
\[(\text{term.transition}_5) \ [E_{14}, E_{19}] = \]
\[(i_{mar} \ x \ \overset{b}{\rightarrow} \ v \ x \wedge \text{memory} \ x = \text{memory} \ x \wedge \ acc \ x = acc \ x \wedge \]
\[o_{mar} \ y \ \overset{r}{\rightarrow} \ \text{Cut}_{16 \cdot 13} (i_{mar} \ x) \wedge \]
\[\text{memory} \ y = \text{memory} \ x \wedge \ acc \ y = acc \ x \]
\)[E_{14}, E_{6a}] \wedge
\]
\[(\text{mout} \ x \ \overset{m}{=} \ \text{Fetch}_{13} (\text{memory} \ x) (o_{mar} \ x) \wedge \ acc \ x \overset{c}{=} \ acc \ x \wedge \]
\[a_{in2} \ x \ \overset{b_2}{\rightarrow} \ \text{mout} \ x \wedge \ acc \ x = acc \ x \wedge \]
\[a_{out} \ x \ \overset{a}{=} \ \text{Sub}_{16} (acc \ x) (a_{in2} \ x) \wedge \]
\[
\begin{align*}
i_{acc} x & \overset{b_{\text{in}}}= aout x \land \\
acc y & \overset{r_{\text{in}}}= i_{acc} x \\
) [E_{5a}, E_{19}] \\
\text{(term.transitions)} [E_{15}, E_{19}] =
\begin{align*}
(i_{mar} x & \overset{b_{\text{in}}}= v x \land \text{memory } x = \text{memory } x \land \\
o_{mar} y & \overset{r_{\text{in}}}= \text{Cut}_{16..13} (i_{mar} x) \land \text{memory } y = \text{memory } x \land \\
) [E_{15}, E_{6a}] \land \\
(mout x & \overset{m}= \text{Fetch}_{13} (\text{memory } x) (o_{mar} x) \land \\
i_{acc} x & \overset{b_{\text{in}}}= mout x \land \\
acc y & \overset{r_{\text{in}}}= i_{acc} x \\
) [E_{6a}, E_{19}] \\
\text{(term.transition) } [E_{16}, E_{19}] =
\begin{align*}
(i_{mar} x & \overset{b_{\text{in}}}= v x \land \text{memory } x = \text{memory } x \land \text{acc } x = \text{acc } x \land \\
o_{mar} y & \overset{r_{\text{in}}}= \text{Cut}_{13} (i_{mar} x) \land \\
\text{memory } y = \text{memory } x \land \text{acc } y = \text{acc } x \\
) [E_{16}, E_{7a}] \land \\
(memory y & \overset{m}= \text{Store}_{13} (o_{mar} x) (\text{acc } x) (\text{memory } x) \\
) [E_{7a}, E_{19}] \\
\text{(term.transitions)} [E_{19}, E] =
\begin{align*}
(a_{\text{in}2} x & \overset{b_{\text{in}}}= \text{Pad}_{16} (pc x) \land \\
aout x & \overset{a} = \text{Inc}_{16} (a_{\text{in}2} x) \land \\
i_{pc} x & \overset{b_{\text{out}}}= aout x \land \\
pc y & \overset{r_{\text{in}}}= \text{Cut}_{13} (i_{pc} x) \\
) [E_{19}, E] \\
\text{(term.transitions)} [E_{5}, E_{10}] =
\begin{align*}
(memory x & \overset{c} = \text{memory } x \land i_{mar} x \overset{b_{\text{in}}}= \text{Pad}_{16..13} (pc x) \land \\
\text{memory } y = \text{memory } x \land o_{mar} y & \overset{r_{\text{in}}}= \text{Cut}_{16..13} (i_{mar} x) \\
) [E_{5}, E_{9a}] \land \\
(mout x & \overset{m} = \text{Fetch}_{13} (\text{memory } x) (o_{mar} x) \land
\end{align*}
\end{align*}
\]
\[ v \, y \overset{F_{20}}{=} mout \, x \]
\[ ) \, [E_{9a}, E_{10}] \]

For a shorter writing form before to derive an implementation we have replacements for signals. We use “\(i_a\)” to replace “\(i_{mar}\)”, “\(a\)” to replace “\(o_{mar}\)”, “\(i_{a2}\)” to replace “\(ain2\)”, “\(o_a\)” to replace “\(aout\)” and “\(o_m\)” to replace “\(mout\)” to get a specification 10’.

**Specification 10’:**

\[
\text{(term.transition)}_0 \, [E_6, E] = \\
\text{(i}_{pc} \, x \overset{B_2}{=} \text{switches} \, x) \land \\
p_c \, y \overset{E_9}{=} \text{Cut}_{13} \, (i_{pc} \, x) \\
) \, [E_6, E] \\
\]

\[
\text{(term.transition)}_1 \, [E_7, E] = \\
\text{(i}_{acc} \, x \overset{B_1}{=} \text{switches} \, x) \land \\
\text{acc} \, y \overset{E_9}{=} \text{i}_{acc} \, x \\
) \, [E_7, E] \\
\]

\[
\text{(term.transition)}_2 \, [E_8, E] = \\
\text{(i}_a \, x \overset{B_9}{=} \text{Pad}_{16}_{13} \, (p_c \, x) \land \text{acc} \, x = \text{acc} \, x \land \text{memory} \, x = \text{memory} \, x) \land \\
\text{a} \, y \overset{E_9}{=} \text{Cut}_{16}_{13} \, (i_a \, x) \land \\
\text{acc} \, y = \text{acc} \, x \land \text{memory} \, y = \text{memory} \, x \\
) \, [E_8, E_{2a}] \land \\
\text{(memory} \, y \overset{E}{=} \text{Store}_{13} \, \text{(a} \, x) \, (\text{acc} \, x) \, (\text{memory} \, x)) \\
) \, [E_{2a}, E] \\
\]

\[
\text{(term.transition)}_3 \, [E_{11}, E] = \\
\text{(i}_{pc} \, x \overset{B_2}{=} v \, x \land \\
p_c \, y \overset{E_9}{=} \text{Cut}_{16}_{13} \, (i_{pc} \, x) \\
) \, [E_{11}, E] \\
\]

\[
\text{(term.transition)}_4 \, [E_{13}, E_{19}] = \\
\text{(i}_a \, x \overset{B_9}{=} v \, x \land \text{memory} \, x = \text{memory} \, x \land \text{acc} \, x = \text{acc} \, x \land \\
)
\[ a \ y \overset{r_m}{\rightarrow} \text{Cut}_{16-13} (i_a \ x) \ \land \\
\text{memory } y = \text{memory } x \land \text{acc } y = \text{acc } x \]

\[ ) \ [E_{13}, \ E_{14}] \land \]

\[ (o_m \ x \overset{m}{\rightarrow} \text{Fetch}_{13} (\text{memory } x) \ (a \ x) \land \text{acc } x = \text{acc } x \land \\
i_{a2} \ x \overset{b_3}{\rightarrow} o_m \ x \land \text{acc } x \overset{c}{=} \text{acc } x \land \\
o_a \ x \overset{a}{\rightarrow} \text{Add}_{16} (\text{acc } x) \ (i_{a2} \ x) \land \\
i_{\text{acc}} \ x \overset{b_1}{\rightarrow} o_a \ x \land \\
\text{acc } y \overset{r_a}{\rightarrow} i_{\text{acc}} \ x \]

\[ ) \ [E_{1a}, \ E_{19}] \]

\[ \text{(term.transition5) } [E_{14}, \ E_{19}] = \]

\[ (i_a \ x \overset{b_a}{\rightarrow} v \ x \land \text{memory } x = \text{memory } x \land \text{acc } x = \text{acc } x \land \\
a \ y \overset{r_m}{\rightarrow} \text{Cut}_{16-13} (i_a \ x) \land \\
\text{memory } y = \text{memory } x \land \text{acc } y = \text{acc } x \]

\[ ) \ [E_{14}, \ E_{5a}] \land \]

\[ (o_m \ x \overset{m}{\rightarrow} \text{Fetch}_{13} (\text{memory } x) \ (a \ x) \land \text{acc } x \overset{c}{=} \text{acc } x \land \\
i_{a2} \ x \overset{b_3}{\rightarrow} o_m \ x \land \text{acc } x = \text{acc } x \land \\
o_a \ x \overset{a}{\rightarrow} \text{Sub}_{16} (\text{acc } x) \ (i_{a2} \ x) \land \\
i_{\text{acc}} \ x \overset{b_1}{\rightarrow} o_a \ x \land \\
\text{acc } y \overset{r_a}{\rightarrow} i_{\text{acc}} \ x \]

\[ ) \ [E_{5a}, \ E_{19}] \]

\[ \text{(term.transition6) } [E_{15}, \ E_{19}] = \]

\[ (i_a \ x \overset{b_a}{\rightarrow} v \ x \land \text{memory } x = \text{memory } x \land \\
a \ y \overset{r_m}{\rightarrow} \text{Cut}_{16-13} (i_a \ x) \land \text{memory } y = \text{memory } x \land \\
) \ [E_{15}, \ E_{6a}] \land \]

\[ (o_m \ x \overset{m}{\rightarrow} \text{Fetch}_{13} (\text{memory } x) \ (a \ x) \land \\
i_{\text{acc}} \ x \overset{b_1}{\rightarrow} o_m \ x \land \\
\text{acc } y \overset{r_a}{\rightarrow} i_{\text{acc}} \ x \]

\[ ) \ [E_{6a}, \ E_{19}] \]

\[ \text{(term.transition7) } [E_{16}, \ E_{19}] = \]
(i_a x \overset{b_4}{=} v x \land \text{memory } x = \text{memory } x \land \text{acc } x = \text{acc } x \land a y \overset{r_m}{=} \text{Cut}_{13} (i_a x) \land
\text{memory } y = \text{memory } x \land \text{acc } y = \text{acc } x
)
) \ [E_{16}, E_{7a}] \land
(m_{\text{memory } y} \overset{m}{=} \text{Store}_{13} (a x) \overset{m}{=} \text{acc } x \overset{m}{=} \text{memory } x
)
) \ [E_{7a}, E_{19}]

(\text{term.transitions}_8) \ [E_{19}, E] =
(i_{a_2} x \overset{b_3}{=} \text{Pad}_{16} (p c x) \land
o_a x \overset{a}{=} \text{Inc}_{16} (i_{a_2} x) \land
i_{p c} x \overset{b_3}{=} o_a x \land
p c y \overset{r_p}{=} \text{Cut}_{13} (i_{p c} x)
)
) \ [E_{19}, E]

(\text{term.transitions}_9) \ [E_5, E_{10}] =
(m_{\text{memory } x} \overset{c}{=} \text{memory } x \land i_a x \overset{b_4}{=} \text{Pad}_{16..13} (p c x) \land
\text{memory } y = \text{memory } x \land a y \overset{r_m}{=} \text{Cut}_{16..13} (i_a x)
)
) \ [E_5, E_{9a}] \land
(o_m x \overset{m}{=} \text{Fetch}_{13} (\text{memory } x) \overset{m}{=} (a x) \land
v y \overset{r_x}{=} o_m x
)
) \ [E_{9a}, E_{10}]

To get a complete specification of Gordon’s computer at this level we need to complement six points:

1. Following the subsection 9.2.3 to eliminate E 0.

2. Omitting all negative control signals. For example in a model formed as
   \[(\alpha \land \text{acc } y = \text{acc } x \land \beta) \ [E_i, E_o]\]
   for the acc y = acc x the control signal of its corresponding Register is
   \(-w\text{acc } x\) but when transforming the model to a corresponding microinstruction we omit the negative signal. This is because that we suppose that
in a microinstruction a negative signal is existent if corresponding positive signal is not existent.

3. According to 2 above the control signals of $st_3 \ [E_9, \ E], (sst_0 \land kss_7) \ [E_{17}, \ E]$ and $(sst_0 \land kss_7) \ [E_{18}, \ E]$ become empty, therefore they become the weakest specification T. For this we use E replace $E_9$ and $E_{19}$ to replace $E_{17}$ and $E_{18}$.

4. We set a signal “idle_t” in microinstruction for state signal idle. Therefore to implement idle.is.true can be setting the idle_t to be true and by the convenience 2 above we omit to write the negative case idle.is.false into the microinstructions. And also we introduce a device Idle

$$\text{Idle} (\text{idle}_t, \ \text{idle}) \equiv \forall t. \ \text{idle}_t = \text{idle}_t \ t.$$  

5. For hand handing signal $hhs$ such as switches, button and knob we introduce extra axioms

$$(hhs \ y = hhs \ x) \ [E_i, \ E_o]$$

6. Finally, to instantiate all variables except the entry predicates.

Then we get

**Specification 11 = Control $\land$ Datapath**

**Control** $\equiv$

$$[E, \ (idle \ x) \ (E_1, \ E_2)]$$  

$\land$

$$[E_1, \ (button \ x) \ (E_3, \ E_4)]$$  

$\land$

$$[E_3, \ (\text{Val}_2 (\text{knob} \ x) = 0) \ (E_6, \ \text{Val}_2 (\text{knob} \ x) = 1) \ (E_7, \ \text{Val}_2 (\text{knob} \ x) = 2) \ (E_8, \ etc.)]$$
\( E'))))))
\( (rsw \ x \land wpc \ x \land idle_t) [E_6, E] \land \)
\( (rsw \ x \land wacc \ x \land idle_t) [E_7, E] \land \)
\( (rpc \ x \land wmar \ x) [E_8, E_{2a}] \land \)
\( (wmem \ x \land idle_t) [E_{2a}, E] \land \)
\( [E_2, (button \ x) (E_4, E_5)] \land \)
\( (idle_t \ x) [E_4, E] \land \)
\( (rpc \ x \land wmar) [E_5, E_{9a}] \land \)
\( (rmem \ x \land wir \ x) [E_{9a}, E_{10}] \land \)
\( [E_{10}, (Val_3 (Opcode (v \ x)) = 0) \land \)
\( (E_4, (Val_3 (Opcode (v \ x)) = 1) \land \)
\( (E_{11}, (Val_3 (Opcode (v \ x)) = 2) \land \)
\( (E_{12}, (Val_3 (Opcode (v \ x)) = 3) \land \)
\( (E_{13}, (Val_3 (Opcode (v \ x)) = 4) \land \)
\( (E_{14}, (Val_3 (Opcode (v \ x)) = 5) \land \)
\( (E_{15}, (Val_3 (Opcode (v \ x)) = 6) \land \)
\( (E_{16}, E_{19})))]) \land \)
\( (rir \ x \land wpc \ x) [E_{11}, E] \land \)
\( [E_{12}, (Val_{16} (acc \ x) = 0) (E_{11}, E_{19})] \land \land \)
(rir x \land wmar x) [E_{13}, E_{4a}] \land
(rmem x \land rm_3 x \land add x \land ralu x \land wacc x) [E_{4a}, E_{19}] \\
\land
(rir x \land wmar x) [E_{14}, E_{5a}] \land
(rmem x \land rm_3 x \land sub x \land ralu x \land wacc x) [E_{5a}, E_{19}] \\
\land
(rir x \land wmar x) [E_{15}, E_{6a}] \land
(rmem x \land rm_1 x \land wacc x) [E_{6a}, E_{19}] \\
\land
(rir x \land wmar x) [E_{16}, E_{7a}] \land
(wmem x) [E_{7a}, E_{19}] \\
\land
(rpc x \land inc x \land ralu x \land wpc x) [E_{19}, E]

Datapath \equiv

Mem (rmem, wmem, a, acc, memory, o_m) \land
Alu (add, sub, inc, acc, i_{a2}, o_a) \land
Reg (wacc, i_{acc}, acc) \land
Reg_p (wpc, i_{pc}, pc) \land
Reg (wir, o_m, v) \land
Reg_p (wmar, i_a, a) \land

(Gate (rsw, switches, o_{11}) \land Gate (ralu, o_a, o_{12}) \land Gate (rm_1, o_m, o_{13}) \land
Bus (o_{11}, o_{12}, o_{13}, i_{acc})) \land

(Gate (rir, v, o_{23}) \land Bus (o_{11}, o_{12}, o_{23}, i_{pc})) \land

(Gate_p (rm_3, o_m, o_{31}) \land Gate_p (rpc, pc, o_{32}) \land Bus (o_{31}, o_{32}, i_{a2})) \land

Bus (o_{32}, v, i_a)

11.2.8 Structural Specification

To construct the microprogram of Gordon’s computer we introduce below devices:
For control we introduce MicroProgram (note: which has included branch structure) and Address.

MicroProgram (address,
    idle, button, knob, v, acc,
    next-address,
    add, sub, inc, rmem, wmem,
    rsw, ralu, rm1, rir, rm3, rpc,
    wacc, wpc, wir, wmar,
    idle\(_t\) ) ≡

(∀x. address x = a_0 ⇒
   next-address x = (idle x ⇒ a_1 \downarrow a_2))

∧

(∀x. address x = a_1 ⇒
   next-address x = (button x ⇒ a_3 \downarrow a_4))

∧

(∀x. address x = a_3 ⇒
   next-address x = (Val_2 (knob x) = 0) ⇒ a_6 ↓
   (Val_2 (knob x) = 1) ⇒ a_7 ↓
   (Val_2 (knob x) = 2) ⇒ a_8 \downarrow a_0)

∧

(∀x. address x = a_6 ⇒ (rsw x ∧ wpc x) ∧ idle\(_t\) x ∧ next-address x = a_0)

∧

(∀x. address x = a_7 ⇒ (rsw x ∧ wacc x) ∧ idle\(_t\) x ∧ next-address x = a_0)

∧

(∀x. address x = a_8 ⇒ (rpc x ∧ wmar x) ∧ next-address x = a_{2a})

∧

(∀x. address x = a_{2a} ⇒ wmem x ∧ idle\(_t\) x ∧ next-address x = a_0)
\(\wedge\)

\((\forall x. \text{address } x = a_4 \Rightarrow \text{idle}_4 \, x \wedge \text{next-address } x = a_9)\)

\(\wedge\)

\((\forall x. \text{address } x = a_2 \Rightarrow \text{next-address } x = (\text{button } x \Rightarrow a_4 \downarrow a_5))\)

\(\wedge\)

\((\forall x. \text{address } x = a_5 \Rightarrow (\text{rpc } x \wedge \text{wmar } x) \wedge \text{next-address } x = a_{9a})\)

\(\wedge\)

\((\forall x. \text{address } x = a_{9a} \Rightarrow (\text{rmem } x \wedge \text{wir } x) \wedge \text{next-address } x = a_{10})\)

\(\wedge\)

\((\forall x. \text{address } x = a_{10} \Rightarrow \text{next-address } x = \)

\((\text{Val}_3 (\text{Opcode} (v \, x)) = 0 \Rightarrow a_4 \downarrow \)

\(\text{Val}_3 (\text{Opcode} (v \, x)) = 1 \Rightarrow a_{11} \downarrow \)

\(\text{Val}_3 (\text{Opcode} (v \, x)) = 2 \Rightarrow a_{12} \downarrow \)

\(\text{Val}_3 (\text{Opcode} (v \, x)) = 3 \Rightarrow a_{13} \downarrow \)

\(\text{Val}_3 (\text{Opcode} (v \, x)) = 4 \Rightarrow a_{14} \downarrow \)

\(\text{Val}_3 (\text{Opcode} (v \, x)) = 5 \Rightarrow a_{15} \downarrow \)

\(\text{Val}_3 (\text{Opcode} (v \, x)) = 6 \Rightarrow a_{16} \downarrow \text{address}_{19})\)

\(\wedge\)

\((\forall x. \text{address } x = a_{11} \Rightarrow (\text{rir } x \wedge \text{wpc } x) \wedge \text{next-address } x = a_9)\)

\(\wedge\)

\((\forall x. \text{address } x = a_{12} \Rightarrow \text{next-address } x = (\text{acc } x \Rightarrow a_{11} \downarrow a_{19}))\)

\(\wedge\)

\((\forall x. \text{address } x = a_{13} \Rightarrow (\text{rir } x \wedge \text{wmar } x) \wedge \text{next-address } x = a_{4a})\)

\(\wedge\)

\((\forall x. \text{address } x = a_{4a} \Rightarrow (\text{rmem } x \wedge \text{rm}_3 \, x \wedge \text{add } x \wedge \text{ralu } x \wedge \text{wacc } x) \wedge \)
next-address \( x = a_{19} \))

\&

(\forall x. \text{address } x = a_{14} \Rightarrow (\text{rir } x \& \text{wmar } x) \& \text{next-address } x = a_{5a})

\&

(\forall x. \text{address } x = a_{5a} \Rightarrow
\quad (\text{rmem } x \& \text{rm}y \ x \& \text{sub } x \& \text{ralu } x \& \text{wacc } x) \& \text{next-address } x = a_{19})

\&

(\forall x. \text{address } x = a_{15} \Rightarrow (\text{rir } x \& \text{wmar } x) \& \text{next-address } x = a_{6a})

\&

(\forall x. \text{address } x = a_{6a} \Rightarrow
\quad (\text{rmem } x \& \text{rm}1 \ x \& \text{wacc } x) \& \text{next-address } x = a_{19})

\&

(\forall x. \text{address } x = a_{16} \Rightarrow (\text{rir } x \& \text{wmar } x) \& \text{next-address } x = a_{7a})

\&

(\forall x. \text{address } x = a_{7a} \Rightarrow \text{wmem } x \& \text{next-address } x = a_{19})

\&

(\forall x. \text{address } x = a_{19} \Rightarrow
\quad (\text{rpc } x \& \text{inc } x \& \text{ralu } x \& \text{wpc } x) \& \text{next-address } x = a_{0})

Note: we use the \( a_i \) \( (0 \leq i \leq 19) \) as variables above which leaves them for further design selection and keeps our microprogram more general.

Address (next-address, address) \( \equiv \forall t. \) address \( (t + 1) = \) next-address \( t \).

For \textbf{datapath} in addition the one in the preceding subsection we introduce a device \text{Idle} as below:

\text{Idle} (\text{idle}_i, \text{idle}) \( \equiv \forall t. \) idle \( (t + 1) = \text{idle}_i \ t \).
Figure 2: Flowchart of the microprogram of the first implementation

Note: the number on the top left of each frame-box is the subscript of the microinstruction address
Figure 1: Datapath of the first implementation

11.3 Deriving other Implementations

In the whole derivation process each nondeterministic construction step offers different selections which can lead to different constructions and finally result in different implementations. In our derivation the main selections are done in three stages: the selecting of the global structure which will lead to different global structures of the whole computer, the allocation which will lead to different datapaths and microprograms and the merging of parallel chains which will lead to
different linear microprograms. In this section we give two other implementations which are derived by the different selections for buses. We call them the second and third implementation to distinguish them from the preceding one which we refer to as the first implementation.

11.3.1 Two-Bus Implementation

We can merge several minimal buses to get a compound bus. Instead of the three minimal buses Bus1, Bus2, and Bus4 with corresponding outputs $i_{acc}$, $i_{pc}$ and $i_a$ we may use a compound bus Bus5. We use "i" as the output of the compound bus, i.e. we use "i" to replace the original three outputs $i_{acc}$, $i_{pc}$ and $i_a$. This results in the following new datapath, which is graphically depicted in Figure 3.

\[
\text{Datapath} =
\]
\[
\text{Mem (rmem, wmem, a, acc, memory, o_m)} \land
\]
\[
\text{Alu (add, sub, inc, acc, i_{a2}, o_a)} \land
\]
\[
\text{Reg}_c (wpc, i, pc) \land
\]
\[
\text{Reg (wacc, i, acc)} \land
\]
\[
\text{Reg (wmar, i, a)} \land
\]
\[
\text{Reg}_c (wir, o_m, v) \land
\]
\[
(\text{Gate (rsw, switches, o_{11})} \land \text{Gate (ralu, o_a, o_{12})} \land \text{Gate (rm1, o_m, o_{13})} \land
\]
\[
\text{Gate (rir, v, o_{23})} \land \text{Bus (o_{11}, o_{12}, o_{13}, o_{23}, o_{32}, i))} \land
\]
\[
(\text{Gate}_p (rm3, o_m, o_{31}) \land \text{Gate}_p (rpc, pc, o_{32}) \land \text{Bus (o_{31}, o_{32}, i_{a2}))}
\]

The second implementation has the same microprogram as the first one. The second implementation can keep each microinstruction and for each bus there is only a connection executed. This is same as in the first implementation.
11.3.2 One-Bus Implementation

To get the third implementation we further merge the Bus5 and Bus3 of the second implementation into a single Bus6. To this end we need to introduce a Reg for the second input $i_{a2}$ of Alu and merge the two outputs of Bus5 and Bus3 to get one output for Bus6. These two changes need to be taken as a compound bus may have only a pair of input and output signals connected at any time point. The change will further lead to a change in the microprogram which is based on the
new datapath. For the third implementation we have the datapath below which is also shown in Figure 4.

\[
\text{Datapath} \equiv \\
\quad \text{Mem (rmem, wmem, a, acc, memory, o}_m) \land \\
\quad \text{Alu (add, sub, inc, acc, i, o}_a) \land \\
\quad \text{Reg}_c (\text{wpc, i, pc}) \land \\
\quad \text{Reg (wacc, i, acc)} \land \\
\quad \text{Reg (wir, o}_m, v) \land \\
\quad \text{Reg}_c (\text{wmar, i, a}) \land \\
\quad \text{Reg (warg, i, i}_o) \land \\
\quad \text{Gate (rsw, switches, o}_{11}) \land \text{Gate (ralu, o}_a, o_{12}) \land \text{Gate (rm, o}_m, o_{13}) \land \\
\quad \text{Gate (rir, v, o}_{23}) \land \text{Gate\_p (rpc, pc, o}_{32}) \land \\
\quad \text{Bus (o}_{11}, o_{12}, o_{13}, o_{23}, o_{32}, i})
\]

The construction for the microprogram of the third implementation is the same as for the first one except for the microinstructions 4a, 5a, and 19 of the Figure 2. As we introduced a new Reg (Named Arg in Figure 4) we need to put in an extra microinstruction step since to pass a data. For this we replace the microinstruction 4a (5a, 19) of the second implementation by two microinstructions 4a and 4b (5a and 5b, 19 and 19a). Finally we get a microprogram as shown in Figure 5.
Figure 4: Datapath of the third implementation
Figure 5: Flowchart of the microprogram of the third implementation
11.4 Discussion

Through the derivation of Gordon’s computer we show a concrete construction process and its result. The process and result have, directly or indirectly, told us nearly the whole story about what our formal derivation means and how it differs from other methods. In this section let us summarize some important aspects.

We may first compare our implementation with the one of [50]. In the table below “Impl$i$” \((1 \leq i \leq 3)\) refer to our three derived implementations where “Impl$_G$” means the implementation of Gordon’s computer. “Microprogram” means the number of microinstructions, “Microcode” the number of microcodes in a microinstruction, and “Max length” the number of microinstructions of the maximal length execution of a machine instruction.

<table>
<thead>
<tr>
<th>Implementation</th>
<th>Impl$_1$</th>
<th>Impl$_2$</th>
<th>Impl$_3$</th>
<th>Impl$_G$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bus</td>
<td>4</td>
<td>2</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Gate</td>
<td>6</td>
<td>6</td>
<td>5</td>
<td>5</td>
</tr>
<tr>
<td>Register</td>
<td>4</td>
<td>4</td>
<td>5</td>
<td>6</td>
</tr>
<tr>
<td>Microprogram</td>
<td>23</td>
<td>23</td>
<td>26</td>
<td>26</td>
</tr>
<tr>
<td>Microcode</td>
<td>14</td>
<td>14</td>
<td>15</td>
<td>16</td>
</tr>
<tr>
<td>Max length</td>
<td>8</td>
<td>8</td>
<td>10</td>
<td>10</td>
</tr>
</tbody>
</table>

Of course, the truly interesting story is the method and treatment process behind these results. In Part 2 we use abstract structure and its construction in a process of formally deriving a class of computers. Therefore, this is is a formal design driven by formal description and construction. A distinct point is that the basic string of derivation is *proof-driven* as opposed to being *artificially given*.

For example, in the time refinement, the early stage of derivation, we identify the global loop structure of the computer. We introduce a global entry predicate $E$ which has four functions:
1. the invariant of the global loop,

2. the clock at machine instruction level,

3. the synchronization for different time layers,

4. the global entry address of the microprogram.

Therefore, $E$ can replace the “ready” of Gordon’s computer for time synchronization. For our implementations that “ready” is redundant.

In the datapath aspect some connections through bus and some registers of Gordon’s computer are not used by our implementations. For example, the output of register Acc can directly be connected to the corresponding input ports of Memory and Alu, therefore it does not need to pass via the bus. This is similar for the output of register Mar which may be directly connected to the Memory and the output of Memory can be directly connected to the instruction register Ir. Also for our first and second implementations the register Arg and Buf are unnecessary. An essential difference is that with our methods all these above implementation structures are very naturally obtained in a formal derivation process.

The microprogram is the center of a computer and in fact our entire derivation process can be seen as a process of constructing a microprogram. The process includes a top-down hierarchy:

1. Global loop

2. Abstract microprogram structure

3. Linear microprogram

4. Microinstruction

5. Microcode
This hierarchy is unfolded in our formal derivation process and is based on our central concept of the derivation model.

Derivation is structure oriented. Derivation always deals with pure structure and leaves many elements uninterpreted. It is unnecessary to specify details, for example, for $\text{Val}_2$, Opcode, etc except for introducing extra axioms to describe some basic relations among them if this is necessary for the derivation. To deal with pure structure and structure construction is the fundamental point for formal derivation. All irrelevant details are ignored focusing on just that structure which is related to a particular construction step.
Chapter 12

Proof-Based Architecture of Computer

By formal derivation for the formal design of computer we hold two things: a formal construction process and a formal specification sequence. When we interpret them to the design in real world we gain a new view — proof-based architecture of computer. In this chapter let us summarize what we have done by the two aspects: formal construction as design and formal specification as architecture.

We use specification (a) as a start point to describe a hardware device

\[ \forall t : \text{time. } \alpha \].................................\( (a) \)

because it is a basic requirement that a hardware device should be of work order for any time point. The quantification \( \forall t \) is written at the most outside of the (a) because time is a general existential primitive factor for hardware device.

We transform a specification formed as

\[ \forall t. \ S \ (t, \ t + 1) \].................................\( (b) \)

to another specification formed as

\[ \forall x. \ \exists y. \ S \ (x, \ y) \].................................\( (c) \)

Because we need to introduce finer time level to support more detailed implementation. This corresponds to the time refinement in the design of computer from a
time unit \( < t, t+1 > \) at the machine instruction level to a time interval \( < x, y > \)
at the microinstruction level.

We introduce entry predicate \( E \) as a higher order variable to express (c) to (d):
\[
\forall x. \exists y. E \ x \Rightarrow \beta \land E \ y \hspace{1cm} (d)
\]
By derivation model (d) can be rewritten as
\[
(\beta) [E, E] \hspace{1cm} (e)
\]
Through (e) we can see that at the highest level a computer has a loop structure
through a global entry predicate \( E \) which finally is interpreted as the global entry
address of the whole microprogram of computer.

\( \beta \) of (e) has state transition to describe the semantics of machine instruction of
computer – using recursion to express infinite iteration of computer running. The
state transition can have a quite complex structure which reflects the complexity
of a computer.

To deal with the state transition following the strategy of “divide and conquer”,
we decompose condition structure, let \( i \) structure and state transition to
a specification whose components are term transition formed as
\[
(s \ t = tm) [E_i, E_o] \hspace{1cm} (f)
\]
Here, \( tm \) is a term with time point \( x \).

At the level the specification of a computer is reduced to term transitions and
their connection relation through entry predicate.

Each term transition is further decomposed to a linear chain with a continue
entry predicate sequence, for example, \( E_1, E_2, \ldots, E_n \). The link of the chain is
formed as
\[
(\gamma) [E_i, E_o] \hspace{1cm} (g)
\]
Here, \( \gamma \) has two kind forms: of function unit or of connection.

This leads to introducing datapath by assigning \( \gamma \) to devices and introducing
microprogram by instantiating the entry predicate variables \( E_i \) and \( E_o \) to instance ones about the entry address of microprogram.

Therefore the basic process of formal construction of computer is to introducing the finer derivation models of three levels:

\[(\alpha)\ [E_i, \ E_o]\] \hspace{5cm} (m1)

Here, \( \alpha \) is about state transition.

\[(\beta)\ [E_i, \ E_o]\] \hspace{5cm} (m2)

Here, \( \beta \) is about term transition.

\[(\gamma)\ [E_i, \ E_o]\] \hspace{5cm} (m3)

Here, \( \gamma \) is about function unit and connection.

The process is mainly through microprogramming of the five levels:

1. Global loop

2. Abstract microprogram structure

3. Linear microprogram

4. Microinstruction

5. Microcode

Derivation model is the foundation. In fact it has more general significance. In the next chapter we will discuss that the model can be taken as a foundation to clean up our understanding and moreover present a new understanding for hardware construction for its basic structure about logic, space and time.
Chapter 13

Logic, Space and Time of Hardware Construction

The work of hardware verification is an one-dimensional signal world: free-variable as signal argument for predicate representing a hardware component and existential quantified port as connection representing the component composition. The other relationships between circuits are expressed through the dimension in space.

The world of construction is a three-dimensional world in logic (Boolean value), space (signal/port) and time (point). Such a world is for describing a dynamic construction process and the structure of a hardware, the relation of its components, is derived in the process.

13.1 Logic, Space and Time Structure of Model

If we describe circuit by static known specification then we only need an one-dimensional signal world where an inverter circuit and the structure of a delay-inverter are described as below respectively:

\[ \text{Inv} (i, o) \equiv \forall t. o (t + 1) = \neg (i \ t), \]

\[ \exists z. \text{Del} (i, z) \land \text{Inv} (z, o) \]
Such a one-dimensional model originates from the obvious static-topological understanding of physical hardware focusing on spatial structural abstraction [57] (p. 269).

However, if we describe hardware circuit for a dynamic construction to derive unknown specification then we need a more complex three-dimensional world about Boolean value, signal (port) and time, which can be more generally called as logic, space and time. In such a world an Inverter is written as

\[ \text{Inv} (i, o) [E_i, E_o] \equiv \forall x. \exists y. E_i x \Rightarrow o y = \neg (i x) \wedge E_o y \]

In addition to \( i \) and \( o \) two higher order predicate free-variables \( E_i \) and \( E_o \) are introduced and the fixed time interval \( 1 \) is made flexible by \( \exists \) variable \( y \). This is a timed logic relationship: at any time point \( x \) under condition \( E_i x \) Inverter is activated and does its work in the time interval \( \langle x, y \rangle \), then at time point \( y \) to hold up \( E_o y \) to activate other circuit.

This is a higher order predicate model about a circuit related to its environment — through \( E_i \) and \( E_o \) in logic, \( i \) and \( o \) in space and \( x \) and \( y \) in time. They are all variables for constructibility. These variables can be instantiated to obtain different implementations: synchronous, asynchronous etc. and typically the microprogrammed ones.

We present a construction model as the box below:

<table>
<thead>
<tr>
<th>Construction Model:</th>
<th>Syntactic Class:</th>
</tr>
</thead>
<tbody>
<tr>
<td>( cm ::= ) ( mp [E, lps] )</td>
<td>( cm : ) construction model</td>
</tr>
<tr>
<td>( lps ::= E )</td>
<td>( lps : ) leaving predicate structure</td>
</tr>
<tr>
<td>( P ) (( lps ), ( lps ))</td>
<td>( P ) : Boolean term</td>
</tr>
<tr>
<td>( mp : ) model predicate (body)</td>
<td></td>
</tr>
<tr>
<td>( E : ) (entry or leaving) predicate</td>
<td></td>
</tr>
</tbody>
</table>

The semantics of the model is defined by the induction translation:

\[
(mp [E, lps])^* = \forall x. \exists y. E x \Rightarrow mp \wedge lps^*
\]

\[
E^* = E y
\]
(P \langle lps_1, lps_2 \rangle)^* = P \Rightarrow lps_1^* \downarrow lps_2^*

To give an example we may write

\((pc \ y = pc \ x) [E_2, b_1 \ x (E_3, b_2 \ x (E_4, E_5))]\)

to stand for

\(\forall x. \exists y. E_2 \ x \Rightarrow pc \ y = pc \ x \land (b_1 \ x \Rightarrow E_3 \ y \downarrow (b_2 \ x \Rightarrow E_4 \ y \downarrow E_5 \ y))\).

Where \(a \Rightarrow b \downarrow c\) abbreviates the term “if \(a\) then \(b\) else \(c\)” [33].

13.2 Logic, Space and Time Structure of Construction

Construction is to divide specification to subspecifications by introducing new entry predicates in logic, new signals (port) in space and new points in time. For example, decomposing a specification delay-inverter (1) to the subspecifications Delay (2) and Inverter (3):

\((o \ y = \neg (i \ x)) [E_i, E_o] \tag{1}\)

\((z \ y = i \ x) [E_i, E_z] \tag{2}\)

\((o \ y = \neg (z \ x)) [E_z, E_o] \tag{3}\)

By introducing predicate variable \(E_z\), signal variable \(z\) and a time point as the argument of \(E_z\). Generally construction is a map:

\((R (A_1, \cdots, A_n)) [E_i, E_o] \rightarrow R' [E_{r,i}, lps_{r,o}] \land \\land_{k=1}^{n} A'_k [E_{a_k,i}, E_{a_k,o}] \land \land_{i=1}^{m} C_l [E_{c_l,i}, E_{c_l,o}]\)

which maps a specification to the implementation. The specification is about \(n\) parts \(A_k\) (1 ≤ \(k\) ≤ \(n\)) with their relation \(R\). The implementation is constituted by \(n + m + 1\) parts: \(R'\) constructed from \(R\), \(A'_k\) from \(A_k\) (1 ≤ \(k\) ≤ \(n\)) and \(C_l\) (1 ≤ \(l\) ≤ \(m\)) is the connections introduced. In the implementation \(E_i\) and
$E_a$ are kept and each introduced new entry predicate must appears two times as leaving one and entry one for two different parts. Each introduced connection is about two new signals as input and output signals from two different parts.

**Definition**

[Input and output signal protection]:

For a syntactic construct $S$ the input protection is

$$S^i \equiv (r_1 y = r_1 x) \land \cdots \land (r_n y = r_n x),$$

where $r_i$ ($1 \leq i \leq n$) are all the input signals in $S$. Similarly,

$$S^o \equiv (s_1 y = s_1 x) \land \cdots \land (s_m y = s_m x),$$

where $s_i$ ($1 \leq i \leq m$) are all the output signals is the output protection for $S$. □

**Example 13.2.1. [decomposition]:**

A conditional structure is formed as

$$(P \Rightarrow R \downarrow S) [E_i, E_a]$$

By decomposition we get

$$(b y = P \land R^i \land S^i) [E_i, E_a] \land$$

$$(c y = b x \land R^i \land S^i) [E_a, E_a'] \land$$

$$(R^i \land S^i) [E_a', c x (E_b, E_c)] \land$$

$$R [E_b, E_a] \land$$

$$S [E_c, E_a]$$

We can reduce

$$(c y = b x \land R^i \land S^i) [E_a, E_a']$$

to the Boolean constant $T$ by use $b, x$ and $E_a$ to instantiate $c, y$ and $E_a'$ respectively. Therefore we get

$$(b y = P \land R^i \land S^i) [E_i, E_a] \land$$

$$(R^i \land S^i) [E_a, b x (E_b, E_c)] \land$$
13.2.1 Logic Structure of Construction

To keep consistency when introducing an entry predicate variable to divide a specification into two subspecifications the variable always to be made to appears in one subspecification as leaving predicate and in the other subspecification as entry predicate. This lead to the logic structure of construction as followings.

1. Composition: Through \( E_m \) to compose \( \alpha \left[ E_i, E_m \right] \) and \( \beta \left[ E_m, E_o \right] \)

2. Condition: \( \alpha \left[ E_a, \beta \left( E_b, E_c \right) \right] \)

Note: by the construction model we can get more general condition combinations.

3. Loop: Through \( E \) to get a loop: \( \alpha \left[ E, E_a \right] \land \cdots \land \gamma \left[ E_z, E \right] \)

13.2.2 Space Structure of Construction

To suit further constructing the three possible cases: port-connection, bus and register for a new possible connection of two components, initially, we always
introduce a pair of ports: one, say \( r \), as the output of a component and the other one, say \( s \), as the input of the other component. Therefore, in the case we always introduce a new connection formed as

\[
(s \quad y = r \quad r) \quad [E_i, \quad E_o]
\]

The connection can further lead to a port-connection, a bus or a register.

1. **Port-Connection**: \( s \) only appears once. In the case to instantiate \( s(r), \ y \) and \( E_i (E_o) \) by \( r(s), \ x \) and \( E_o (E_i) \) to get the weakest specification — Boolean constant \( T \).

2. **Bus**: \( s \) is the same output of two or more connections.

\[
(s \quad y = r_k) \quad [E_{ik}, \quad E_{ok}] \quad (k \geq 2)
\]

In the case a bus is introduced for these connections.

3. **Register**: if a time constraint is added:

\[
(s \quad y = r \quad r \wedge y > x) \quad [E_i, \quad E_o]
\]

Therefore, through a special model connection the space structure of a circuit can be systematically constructed.

### 13.2.3 Time Structure of Construction

Time variables are always introduced together with entry predicates and we use entry predicate to measure time.

**Definition 13.2.3.1. [time point]:**

Time point \( t \) is one making an entry predicate \( E_x \) at the \( t \) true.

**Definition 13.2.3.2. [time clock & time layer]:**

For a circuit its clock is a disjunction formed as

\[
\bigvee_{k=1}^{n} E_k
\]

Here each \( E_k \) is an entry predicate of the circuit. When one and only one predicate in the clock gets true the clock gets a time point. We call the set of all entry
predicates of a circuit as its time layer. Therefore time clock and layer is relative to the specification level.

**Definition 13.2.3.3. [time construction]:**

The construction from a high specification level whose time layer is \( L_i \) to the adjacent low specification level whose time layer is \( L_{i+1} \) is through introducing a set \( \Delta \) of new middle entry predicates at the low level. Therefore the time construction is

\[
L_i \cup \Delta = L_{i+1}
\]

As \( \Delta \) for each level is always non-empty the whole time construction process of a circuit from top specification 1 to bottom implementation \( n \) is

\[
L_1 \subset L_2 \subset \cdots \subset L_n
\]

**Definition 13.2.3.4. [time unit]:**

For a circuit when each construction model formed as

\[
S (x, y) [E_i, E_o]
\]

is instantiated by

\[
\forall x. E_i x \Rightarrow S (x, x + \delta) \land E_o (x + \delta)
\]

The \( \delta \) is called as a time unit of the circuit.

So, we can have a sequence of clocks for a circuit:

1. Non-Clock: Do not introduce clock.
2. Variable-Clock: Introducing clock and time unit.
3. Constant-Clock: Introducing clock and time unit and moreover introducing a constraint that the \( \delta \) of every model is the same.
13.3 Case Study

The construction of a computer can form mainly the three levels:

1. **State Transition:** This is the behavior specification at machine instruction level and each state transition describes the semantics of a instruction.

   \[ S \ [E, \ E] \]

2. **Term Transition:** This is the mixed behavior-structural specification at term transition, which is a substate transition and similar other ones and corresponds to a linear microprogram, level.

   \[ \wedge_{k=1}^{n} T_k \ [E_i, \ E_o] \ (k \geq 1) \]

3. **Register Transfer:** This is structural specification considered as implementation. Each \( R_l \) of the model in the specification is either a function which is considered implementable at the register transfer level or a connection for port-connection, bus or register.

   \[ \wedge_{l=1}^{m} R_l \ [E_a, \ E_b] \ (l \geq 1) \]

The computer has its logic, space and time structure for each level. Mainly, this is a hierarchy: global loop, abstract microprogram, linear microprogram, microinstruction and microcode.

13.4 Conclusion

For a circuit its three aspects of logic, space and time structures can transform each other by different selections to instantiate variables.

The logic, space and time structures of a circuit are relative to the specification level the circuit currently is at. Construction as a process can unfold a circuit through a sequence of levels. Therefore, we must distinguish these structures for
different levels. It is an interesting new understanding to realize, for example, the same entry predicate, the same bus, the same register or the same time point can express quite different things as it can be at different specification levels. For example, an entry predicate is taken as the entry address of a large microprogram and meanwhile as the entry address of only a microinstruction. A bus or a register is at a high level and the other is at the low level. This time point is shared by three levels and that time point is shared by five levels.

For a circuit through its construction process it forms a logic, space and time world of its own and by itself and for the same circuit the different construction processes can lead to quite different worlds. The world can be encapsulated from the environment but only communicate to through the entry predicate, port and time variables of the top specification of the circuit. Therefore a hierarchy and multi-colored structure of circuit in logic, space and time three aspects can be naturally constructed and unfolded by our construction model, construction scheme and methodology.
Chapter 14

Related Work

In this section we describe the work related specially to our main area of interest: computer verification and formal synthesis.

14.1 Computer Verification

Gordon’s verification of a simple computer, called Gordon’s computer by others, using the LCF_LSM system [31] was an early example of how formal proof and mechanical proof-generation could be used to reason about the design of a microprocessor. This is an elegant example which has a suitable size as a complex case study and holds nearly all main aspects a realistic computer should have. Moreover, its verification process has captured essential aspects of computer verification from an academic research viewpoint. Joyce reverified Gordon’s computer using HOL [50]. Gordon’s computer [31], [50] is a main source for this thesis and is generalized to a class of computers. Gordon’s computer was taken as a case study of formal derivation by the author [73].

Two notable case studies widely seen as the current state-of-the-art in microprocessor verification are Hunt’s verification of the FM8501 [44] and Cohn’s
verification of the VIPER [14], [15]. The FM8501 is a 16-bit microprocessor similar in complexity to a PDP-11. The instruction set of this microprocessor is rich enough to support realistic application. The specification and implementation of this computer are described in pure Lisp and the verification is based on the Boyer-Moore theorem-prover. The work of Hunt is followed by further studies [18] in the FM series and among them an interesting work is [6] on the derivation of a Verified Microprocessor, the DDD-FM9001. The DDD-FM9001 is a 32-bit general purpose microprocessor formally derived directly from Hunt’s mechanically verified Nqthm FM9001 microprocessor specification. This example witnesses an inherent trend from verification to derivation, although the derivation of [6] is based on algebra transformation.

VIPER is a commercially available microprocessor which was designed by the British Ministry of Defence for use in life-critical applications [19], [20]. Cohn has completed two verification levels: The first level is from the major state machine to the top-level specification [14] and the second level is from the block level description to the major state machine [15]. As pointed out in [53] a particularly interesting outcome of the VIPER project is that it revealed weaknesses in the links between designer, verifier and manufacturer. The reason was seen in the fact that a formal description of the block level must be derived from a mixture of engineering documents, partly pictorial and partly textual, supplied by the VIPER designers. In this context Cohn pointed out the scope and limitations of using formal proof to verify microprocessor systems [16].

Still more than the above two notable case studies the works of J.J. Joyce [53] and P. J. Windley [78] have relation to our research concept. Joyce verified the microprocessor TAMARACK-3 and moreover a microprocessor-based system including a simple compiler. An important concept in his work is the generic specification that is parameterized with uninterpreted data types and uninterpreted primitives. Using this concept of generic specification the verification of TAMARACK-3 can encompass the wider domain of microprocessors for which
the uninterpreted data types and primitives are abstract enough to cover each one among them [51] [52] [53].

J.J. Joyce’s concept of generic specification still does not directly describe the specification for a class of computers. His aim is through a concrete microprocessor to introduce somewhat general specification structure and his method is in a concrete microprocessor, TAMARACK-3, to introduce the more generic and abstract notations which are called the uninterpreted data types and primitives. Our aim is to derive a class of computer and therefore we must directly describe what the specification of a class of computers is. For this our top specification scheme and all further derived specification schemes are, fundamentally, to describe abstract computers: not only their primitive components but also their high level structure. Therefore what we deal with is a generic structure called specification scheme and the generic structure can be instantiated to get any concrete computer in a set of computers. The set is described by the generic structure. We also call the set as a class of computers as the title of our thesis.

The work of Windley is even more abstract than Joyce’s. It concentrates on the abstract structure of general verification of microprocessors [74] [75] [77] [78]. Windley presents the concept of generic interpreters including state, state transition function, selection function and interpreter predicate, and, based on these definitions, the abstract representation and the theory obligations. The generic interpreters are a framework covering the common and basic process of the verification of microprocessors.

Windley’s work reveals a generic structure and process of verification of microprocessors. This is a very interesting result. However, two aspects should be pointed out. First, his abstract interpreter is a generic and still somewhat coarser framework. Second, the framework is suitable for verification only. Main limitation is that each level of interpreters of microprocessors are given by the verifier and Windley’s theory of generic interpreters [78] only describes relations between these given interpreters. For our derivation of a class of computers instead of the
given specification our framework must be able to construct specification in each specification scheme. To construct a class of computers what is given is only the top specification scheme at machine instruction level and all lower level specification schemes down to the bottom specification scheme at register transfer level should be constructed step by step from the top specification scheme. Therefore our top specification scheme must be generic and meanwhile accurate enough to make these constructions available. So, for our derivation of a class of computers, we are at a more complex and difficult position that we must present a generic framework not only for the top specification scheme but also for whole construction and verification process from the top scheme down to the bottom scheme.

14.2 Formal Synthesis

Fourman, Hanna, Milne and others pointed out that there are many advantages in combining the processes of design and formal verification [26], [36], [61]. In [26] Fourman presented a formal approach for system level design to deal with complexity and abstraction. It is mainly using formal methodology to express goal-directed design including goal-directed problem solving, proof-search as problem solving, modeling the design state and formalizing design refinement. The formal approach is based on the Lambda logic system [24]. Lambda consists of a prover kernel with rewriting, unification, subgoal and many other functions available in state-of-the-art systems, structured libraries of rules and common logic definitions, and a user-interface which includes a browser and the graphics tool DIALOG. The system is implemented and runs in Poly/ML. The LAMBDA logic is axiomatized as a classical higher-order predicate logic with polymorphic types. The syntax of LAMBDA terms is kept close to ML, enhanced with the common operators of higher-order logic.
[26] describes a general design methodology and a key point is that it can be formally described and executed in LAMBDA system [24]. Some primitive programming tests of our thesis have been done in LAMBDA system. The HOL system [33] and LAMBDA system provide quite powerful support for mechanical proof not only for hardware but also for software and wider areas. The work of our thesis are based on these systems to use proof-based method to deal with certain application area: a class of computer. Although our work concentrates on a limited area comparing with the work of universal design methodology and general design logic below, however, some general significant concepts are presented through our work. This is mainly the concept of derivation model which describe a general logical structure to cover whole construction process of a class of computer and possibly to be extended to whole hardware area.

Hanna clearly introduced the concept of design logic [38], [39]. He used dependent types to achieve a rigorous and yet theoretically simple connection between circuits and their behavioral specifications [40]. He used the VERITAS system in a goal-directed manner to generate interactively a design and a correctness proof. The synthesized design is then translated into the MODEL hardware description language [36]. Dany Suk used Hanna’s idea in his hardware type theory [70].

The concept of dependent type is the cornerstone of Hanna’s system and it is concise and beautiful. He presented the concept of design logic for hardware. This is bold and creative. However, with respect to realistic design practices this is still at a quite abstract description level. Using the design logic to describe complex realistic problems is still quite difficult. This is just as Hanna himself said in [40] pp 55: “in the authors’ opinion, a very open question as to whether these theoretical gains can be translated into practical ones. Thus far, our own computational trials have been limited to very simple circuits and it is not clear how both the computational load and the amount of human guidance required will scale with increasing size of circuit.”

We think that we should find a way to bridge from general design methodology
and general design logic to realistic and complex applications. For this we need to select a trade-off: instead of universal design methodology and general design logic we limit our goal in certain area, for example, the formal design of a class of computers, which is at a level of nearly realistic hardware complexity and meanwhile has a concise form of general logical structure. However, for the limited area we work forward to a goal which is as generic as possible, for example, for our work we deal with a quite generic top specification scheme but not one concrete computer. In the process we can obtain a trade-off point: it is general enough for the certain area of hardware, for us it is the formal design of computers, but still particular for more general design logic and design methodology. For our work these trade-off points are the concepts of specification scheme and derivation model. Instead of abstract and general design methodology and design logic our derivation model as a general hardware model can cover whole construction process of a class of computers and we expect that it can be generalized to cover even whole hardware area.

14.3 Other Relevant Work

In this subsection we cite some work which do not directly related to our work but are interesting referring to them to understand our work. Mainly, these work deal with the verification of microprocessors or proof-based formal hardware design methods.

D. Borrione et al. [5] presented μSPEED, a framework for specifying and verifying microprocessors. It is a proof methodology associating functional semantics to each abstraction level of specification and verification and is implemented by a menu-driven tool working as a specialized editor. This tool automatically produces the functional expressions associated to these specification levels, by way of
a user-friendly interaction with the hardware designer, and calls on a specialized
formal verifier based on rewriting techniques.

M. Langevin and E. Cerny [54] used the formalism of a specific kind of log-
ic formulas which can be represented using BDDs in a verification algorithm
for processor-like circuits. The verification algorithm consists in extracting the
observable behavior of the circuit implementation and in comparing it with the
specification.

J. Chazarain and H. Collavizza [11] developed an object oriented program-
ing based method and a computer algorithm system to prove automatically the
correctness of the translation into micro-instructions of each assembly language
instruction for a given processor.

N. A. Harman and J. V. Tucker, [41] introduced a general algebraic method for
modeling microprocessors at different levels of abstraction by means of iterated
maps to support equational specification and design.

D.A. Basin and P. B. Jackson reported about their work using Nuprl to extract
circuits from constructive proofs [1] [2] [47]. D.A. Basin described a foundation
for using constructive type theory as a tool for combining hardware synthesis and
verification. He specialized the proofs-as-programs paradigm for the Nuprl con-
structive type theory to “proofs-as-circuits”, whereby representations of circuits
can be automatically synthesized from constructive proofs.

H. Busch [7], [8] presented a framework for proof-based transformation of hard-
ware designs.
Chapter 15

Conclusions and Further Work

15.1 Conclusions

Two intentions stimulated our work. One is to investigate the question of how to apply formal methods to complex and realistic hardware designs. The other is to investigate what best formal methods can do for hardware design. These lead to our thesis: the formal derivation of a class of computers.

We formally specified, constructed and verified a class of computers from the machine instruction specification at the behavior level to the register transfer description at the structural level.

The “class” is defined by a behavior specification scheme which is an abstract syntax structure. The structure can cover a set of specifications: any specification which is an instance of the abstract syntax structure belongs to the behavior specification scheme. The behavior level structure is a general semantic framework to describe the state transition semantics of the computer’s machine instructions. It is a scheme because it can be instantiated to specify the semantics of a particular instance of a computer. Therefore the behavior specification scheme establishes a wide class of computers as the target of formal derivation.

Formal construction is a main characteristic of derivation. It can describe exactly and accurately how to produce a new specification from a given one. This is a basic and essential step for formal design. Instead of using proof to replace
construction we proposed to view construction and proof as two independent parts and in particular make a formal construction which is independent of proof. Both aim at the same goal: to treat the design process formally.

We singled out 10 levels of specification schemes, and correspondingly, presented 9 construction steps. Except for the behavior specification scheme which is given each specification scheme was formally constructed from its parent specification scheme and was formally proved correct with respect to its parent scheme. Because a specification scheme covers a set of specifications for each construction step we proved its totalness as a mapping to guarantee that the construction step covers each specification in the parent specification scheme, while correctness guarantees that each constructed specification satisfies its corresponding parent specification. The specification schemes and construction steps cover the formal derivation of a class of computers from the behavior specification scheme to the structural specification scheme (implementation). Thereby we completed the formal derivation for a class of computers.

Derivation differs from verification. Verification concentrates on consistent proofs relating implementation and specification, and passively accepts the designed implementation as a given fact. Derivation on the other hand has the freedom of design: only the specification is given from which it actively constructs the implementation and at the same time to verifies the construction process. Therefore, derivation must deal with both aspects: formal construction and formal verification, especially, it should make clear what formal construction means and how to do it.

However, we should fairly comment on verification, derivation, and their relationship. Verification must be able to treat a having-been-designed implementation by rigorous proof. This makes verification sometimes more difficult than design including derivation as formal design. At root, the great significance of verification is the formal method. By the formal method we can enter a new world: using rigorous mathematics to hold an object. Derivation is developing verification further in
that it avoids the difficulty of verification by putting construction and verification together. Derivation should make its distinct point: formal construction.

Derivation differs from synthesis. Generally speaking, derivation is only a part of synthesis. Synthesis is a somewhat loose concept. Usually the term refers to any process leading from a specification to an implementation. In contrast, derivation only concentrates on the basic aspect of formal synthesis: formally to produce a new specification from a given specification. It cannot include other aspects of synthesis like, for example, optimization. Derivation emphasizes formal construction and logical proof-based verification.

Formal construction is a leading string throughout the whole derivation process. The formal specification and verification of a derivation only serves the formal construction.

In this thesis we presented the derivation model which is the center of our derivation method. With the model we have a clear form to express the relationship between specifications. We described the construction process in the three-dimensional world of logic, space and time. A key concept in the model is the entry predicate which “glues” specifications together and is a general relation existing in the construction of microprogrammed computers. Based on the concepts of the derivation model and entry predicate the construction process of a computer mainly becomes a process of introducing new entry predicates for the logical dimension, new time points in the temporal dimension, and new ports (signal) in the spatial dimension.

Our work of deriving a class of computers established a concept of proof-based computer architecture. It demonstrated that the hierarchical structure of a whole computer can be organized through a dynamic construction and proof process. To understand such a computer architecture we proposed to put it into a three dimensional world of logic, space and time where the logical aspect is the leading one. Based on the derivation model and entry predicate we showed
that the two aspects of unity of opposites, the controlling and the controlled, are
visible throughout the whole hierarchical construction process of a computer from
behavior to structural specification and, similarly, the hierarchical construction
of the microprogram is visible throughout the whole derivation process too. The
primitive element of control is the microinstruction address. We established a
hierarchy of microprogramming whose three high levels are global loop, abstract
microprogram structure, and linear microprogram. We identified a sequence of
time layers throughout the whole derivation process in which the two traditional
time layers of machine instruction and microinstruction are only two extreme ends
(special cases).

We took Gordon’s computer as a typical case study to illustrate the presented
concepts and methods. We derived three different implementations and compared
them with the one of Gordon’s computer.

15.2 Further Work

15.2.1 Mechanical Design System

The natural next step to be taken is to implement and mechanize our work. We
would aim for a semi-automatic or nearly automatic tool for the design and veri-
fication of computers. It would give exact documentations to explain the middle
design processes and results, and especially the abstract and high level structure
of a computer. What the user would need to do is only to give the behavior
specification describing the meaning of machine instruction of the intended com-
puter, and some high level selection including the optimization from the high-level
synthesis. The mechanization of our work is feasible because the formal base and
basic process have been set up and made clear in the present thesis. In fact the
essential part of the mechanization has been reduced to be such exquisite that it
can be implemented by about forty clauses of Prolog.

If we develop a computer-aided design system for the design of computers
along the lines set out in this thesis we have more freedoms of choice. First, for
each of the two aspects of derivation, construction and verification, we can choose
to be either formal or mechanical. For example, we can choose a mechanical
design which can automatically produce a whole class of implementations and a
formal verification which verifies the entire set of constructed specifications to be
correct rather than on a one-by-one basis. Second, we can choose only to derive
a concrete computer following the general scheme of specification, construction,
and verification of a class of computers.

A first step towards an implementation of a mechanical construction and mech-
anical verification using Prolog and LAMBDA has been undertaken by the author
[73] [12] [24]. The design of the whole implementation will be an interesting non-
trivial work. We wish to have a chance to complete this work.

15.2.2 Mathematical Foundation of Hardware Design

To represent circuits in logic it is usual to adopt the following paradigm [32], [40]:

- Components are predicates and ports are free variable.
- Port connections amount to existential hiding.
- Component composition is logical conjunction.

In this thesis the paradigm was extended to the derivation model for deriving mi-
croprogrammed computers. As a further work we will investigate the possibility of
extending the derivation model to a wider area. The idea is that we should find
a suitable form to express the logical relation of hardware components directly
instead of dealing with it in the proof process indirectly. Based on the extended model the formal construction of hardware will become *Logic Structure Refinement*.

Finally, it is author’s personal viewpoint that I believe that in the area of hardware we can also find a formal model and a logical system which is at such a position as Hoare Logic’s fundamental, concise and as a start point for its following formal programming researches. only need about forty clauses
Bibliography


Bibliography


