Formal Design of a Class of Computers

— its high stage: abstract microprogramming[3]

Li-Guo Wang[1]
Department of Computer Science,
University of Edinburgh,
JCMB, The King's Buildings,
Room 1414,
Edinburgh EH9 3JZ, Scotland, U.K.
E-mail: lgw@dcs.ed.ac.uk

Michael Mendler[2]
Department of Computer Science,
Technical University of Denmark,
Building 344,
DK-2800 Lyngby, Denmark
E-mail: mvm@id.dtu.dk

Abstract

We present a novel construction model of hardware and demonstrate how to use it in the entire process of formally designing a class of computers involving their specification, construction, and verification. In this report we focus on the high stages of the design: the refinement from the behaviour specification at machine instruction level to the abstract microprogram at the term transition level. The concept of term transition introduced in this work establishes a new generic high-level design stage which is common for computer architecture and design. Our approach is based on formal syntax and formal proof and constitutes a framework for the rigorous specification and verification of hardware synthesis systems.

1 Introduction

As formal methods are striving to be understood and accepted by industry the formalization of the design process itself — as opposed to mere post-design verification — receives more and more attention [1, 5, 15, 16, 31, 37]. In this report we discuss the formal design of a class of computers. By formal design we mean an approach based on theorem-proving and by a class of computers we mean the specification scheme of an abstract computer characterizing each computer in the class.

We present a *construction model* of hardware which applies all the way from an abstract behaviour specification to an unknown structural implementation. It constitutes a new foundation for formal design where the traditional verification-oriented hardware model does not meet the goal. It admits finer-grained design descriptions in which aspects of hardware and software may coexist, and it abstracts from the concrete mode of operation. The choice, for example, of a synchronous, asynchronous, or pipelined realization may be postponed until later design stages.

Using the construction model the framework comprises a suite of specification, construction, and proof schemes to cover the complete formal design for a class of computers from the behaviour specification at machine instruction level to the structural specifica-

[1] The author is supported by a scholarship from Siemens AG Munich and scholarship from SERC GR/F 35890 U.K.

[2] The author is supported by a Human Capital and Mobility fellowship in the *EuroForm* network.

[3] This report is a slight extension of a paper presented at CHARME'95, *Correct Hardware Design and Verification Methods*, Frankfurt, October 2-4, 1995.

tion at register transfer level. In this paper we discuss the first part of the design process: from the behaviour specification to the high-level abstract microprogram at term transition level.
Through exact formal construction we can identify the *term transition* level as a new and rather natural design stage for computer design. Using term transitions we can capture the abstract and high-level structure of the computer above register-transfer level.

2 Construction Model of Hardware Design

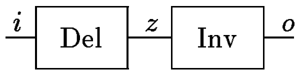
In the field of formal hardware verification a variety of different ways of modelling behaviour have been introduced. The two salient variants are the functional and the relational paradigm. The former is well known from the work of W. Hunt [24] and usually adopted in verification research based on the Boyer-Moore theorem prover. The latter, on the other hand, probably is the most widely accepted and successfully used hardware model in the HOL community [20, 18, 19, 15, 26]. Though one method of specifying hardware often can be encoded in terms of the other, the relational paradigm is usually considered to be the more general one, covering the functional variant as a special case. For instance, in terms of relations we can directly capture phenomena which are not elementary in a purely functional setting such as bidirectionality, partiality, and nondeterminism.
This work presents an extension of the components-as-relations, or to be more precise, the components-as-predicates paradigm. This paradigm, which we simply call the (first-order) *verification model*, can be characterized by three features:

1. Components are represented by predicates with the free variables as their ports.
2. Internal port connection is realized by existential quantification.
3. Composition of components is achieved by logical conjunction.

For example, the delay and inverter circuits would be defined as

$$\begin{aligned} \mathrm{Del}\,(i, z) &\equiv \forall t.\ z\ (t+1) = i\ t \\ \mathrm{Inv}\,(z, o) &\equiv \forall t.\ o\ (t+1) = \neg(z\ t) \end{aligned}$$

and the structure of the delay-inverter



be described as $\exists z.\ \mathrm{Del}\,(i, z) \wedge \mathrm{Inv}\ (z, o)$.
The verification model originates from the obvious static-topological understanding of physical hardware focusing on the port connection structure of hardware and spatial structural abstraction. This is expressed for instance in [30] (p. 269):

"*The type of abstraction most fundamental to hardware verification is* structural *abstraction – the suppression of information about a device's internal structure.*"

Despite its generality a weakness of the verification model is that it is not flexible enough to support a process of formal synthesis and construction. It exhibits the spatial structure of hardware but fails to capture important other structure arising in a top-down construction of hardware. Examples of such non-spatial structure are the interface between software and hardware, microprograms, or behavioural constraints. Therefore the traditional verification model, which flattens away this extra conceptual structure, is intrinsically limited to *post-hoc* verification and transformation of *complete* hardware components

[12, 13, 9, 24, 27, 17, 28, 30]. To capture the top-down construction of *incomplete* hardware, in particular for the construction of microprogrammed computers, more sophisticated structure is needed. One purpose of this paper is to introduce such a new and more general hardware model, the *construction model.* From our point of view the main aspect in the formal design process, taking a fine-grained sequence of refinement steps from an abstract behaviour specification to an eventual structural implementation, is *logic* rather than *spatial* refinement. In order to support this logical decomposition our construction model allows for extra logical communication of a component with its environment.

With the model the inverter is written as

$$\begin{aligned}&\mathrm{Inv}\,(z, o)\,[E_i,\ E_o] \ \equiv \\ &\quad \forall x.\ \exists y.\ E_i\ x \Rightarrow o\ y = \neg(z\ x) \wedge E_o\ y.\end{aligned}$$

This formulation states that whenever a certain condition E_i is true at the start time point x then the inversion of the input will be effected by some time y, and after the execution the predicate E_o will be true. The predicates E_i and E_o are the logical interface to the environment by which the entry and exit of the component is controlled. We call E_i the *entry* and E_o the *leaving* predicate of the model. The difference to the verification model is the fact that it focuses on *what* makes up the behaviour of an inverter, *viz.* the equation $o\,y = \neg(z\,x)$, rather than *how* this is achieved. The original absolute reference to a fixed propagation delay has been abstracted away and replaced by a generic time interval $\langle x, y\rangle$ demarcated only by the conditions $E_i\,x$ and $E_o\,y$ but otherwise undetermined. Here the predicates E_i, E_o are considered parameters, *i.e.* free variables, of the specification rather than concrete predicates. This results in a generic description, which can be instantiated in many ways to obtain a variety of different realizations of an inverter. For instance, we get back to the original formulation by taking $E_i\,x \ \equiv\ x = t$ and $E_o\,y \ \equiv\ y = t+1$, where t is a free variable referring to a global notion of time. The substitution gives

$$\forall x.\ \exists y.\ x = t \ \Rightarrow\ o\,y = \neg(z\,x) \wedge y = t+1,$$

which is logically equivalent to the old $\mathrm{Inv}\,(z, o)$, but it makes explicit the essential mode of control implicit in the original formulation, *viz.* unconstrained operation with a fixed propagation delay relative to a global and absolute concept of time. Other modes of operation can be obtained by other choices of E_i, E_o. For instance, viewing the inverter function as part of a microprogram might lead us to choose $E_i\,x \ \equiv\ \mathrm{addr}\,x = 3$ and $E_o\,y \ \equiv\ \mathrm{addr}\,y = 4$, where addr refers to the microprogram address counter and x, y to the sequence number of the executed microinstruction. Another possibility is given by viewing the inverter as part of a synchronous circuit: we instantiate $E_i\,x \ \equiv\ \mathrm{stab}(z, x, \Delta) \wedge \mathrm{clk}(x, n)$ and $E_o\,y \ \equiv\ \mathrm{clk}(y, n)$ such that $\mathrm{stab}(z, x, \Delta)$ means input z has been stable for at least a period of length Δ prior to time x, and $\mathrm{clk}(x, n)$ means x corresponds to the n-th clock tick. We leave it to the reader to add other instantiations, say to capture input constraints, multiple phase synchronous realizations, or asynchronous handshake. The semantics of the entry and leaving predicates is non-trivial in the sense that not all possible instantiations are permitted. In particular, it is not permitted to let the entry predicate be inconsistent, say $E_i\,x \ \equiv\ 0 = 1$. In this case the construction model would trivialize to the full relation which does not constrain the implementation at all. Our framework avoids these fallacies by keeping a rigid dicipline in the use of the entry and leaving predicates. Technically, the point is that in our constructive setting

$\mathrm{Inv}\,(z,o)\ [E_i,\ E_o]$ does not have the same meaning as $\exists E_i.\ \mathrm{Inv}\,(z,o)\ [E_i,\ E_o]$.
To sum up, the construction model allows us to capture a wide range of possible modes of control and interaction, and — this is the crucial point — to keep the decomposition at an abstract level as long as convenient and defer instantiating to a particular realization until much later in the design process.

Let us consider a very simple example. Suppose in the process of construction we encounter the construction model

$$\begin{aligned}&\forall x.\ \exists y.\ E_i\, x\ \Rightarrow\ \mathrm{rec}\ y = \\ &\qquad \mathrm{ADD}\,(\mathrm{send}_1\ x, \mathrm{send}_2\ x)\ \wedge\ E_o\, y \qquad (1)\end{aligned}$$

as a sub-specification. It says that in the time interval $\langle x, y\rangle$, and under the start condition E_i, we are to implement an add function so that upon completion the condition E_o holds. We must achieve this for arbitrary x but are free to choose the exit time point y. To construct (1) we might naturally (*e.g.* by syntactic decomposition) introduce two new entry predicates E_a and E_b and three new ports i_1, i_2, and o as follows:

$$\begin{aligned}&(\forall x.\ \exists y.\ E_i\ x\ \Rightarrow\ i_1\ y = \mathrm{send}_1\ x \\ &\qquad \wedge\ i_2\ y = \mathrm{send}_2\ x\ \wedge\ E_a\ y) && (2)\\ &\wedge\ (\forall x.\ \exists y.\ E_a\ x\ \Rightarrow\ o\ y = \\ &\qquad \mathrm{ADD}\ (i_1\ x, i_2\ x)\ \wedge\ E_b\ y) && (3)\\ &\wedge\ (\forall x.\ \exists y.\ E_b\ x\ \Rightarrow\ \mathrm{rec}\ y = \\ &\qquad o\ x\ \wedge\ E_o\ y). && (4)\end{aligned}$$

This way we have decomposed the original construction model (1) into a composition of three abstract subcomponents (2)–(4). These subcomponents communicate spatially via ordinary signals such as i_1 or o but also logically via entry and leaving predicates such as E_a. By this extra dimension of interaction we have not only explicit a data-path but also abstract control reflected in the entry and leaving predicates. More concretely, if we condense every equation into a unique control signal to activate a hardware component effecting this equation within one unit of time, then we arrive directly at the following description of an abstract microprogram:

$$\begin{aligned}&(\forall t.\ E_i\ t \Rightarrow \mathrm{c}_{s1}\ t\ \wedge\ \mathrm{c}_{s2}\ t \wedge E_a\ (t+1))\\ \wedge\ &(\forall t.\ E_a\ t \Rightarrow \mathrm{c}_a\ t\ \wedge \qquad\quad E_b\ (t+1))\\ \wedge\ &(\forall t.\ E_b\ t \Rightarrow \mathrm{c}_r\ t\ \wedge \qquad\quad E_o\ (t+1)).\end{aligned}$$

From here we might further instantiate the entry predicates as statements about the state of a program counter to obtain a concrete microprogram:

$$\begin{aligned}&(\forall t.\ \mathrm{address}\ t = 12\ \Rightarrow\ \mathrm{c}_{s1}\ t \wedge \mathrm{c}_{s2}\ t\\ &\quad\wedge\qquad \mathrm{address}\ (t+1) = 13)\ \wedge\\ &(\forall t.\ \mathrm{address}\ t = 13\ \Rightarrow\ \mathrm{c}_a\ t\\ &\quad\wedge\qquad \mathrm{address}\ (t+1) = 14)\ \wedge\\ &(\forall t.\ \mathrm{address}\ t = 14\ \Rightarrow\ \mathrm{c}_r\ t\\ &\quad\wedge\qquad \mathrm{address}\ (t+1) = 15).\end{aligned}$$

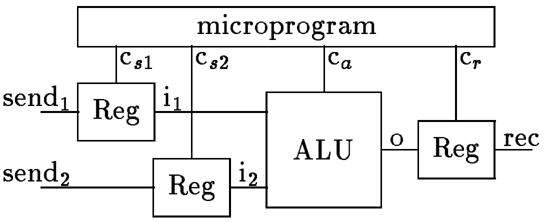
The data-path, which implements the equations within (2)–(4),

$$\begin{aligned}&\mathrm{Reg}\ (\mathrm{c}_{s1}, \mathrm{send}_1, \mathrm{i}_1)\ \wedge\ \mathrm{Reg}\ (\mathrm{c}_{s2}, \mathrm{send}_2, \mathrm{i}_2)\\ &\quad\wedge\ \mathrm{Reg}\ (\mathrm{c}_r, \mathrm{o}, \mathrm{rec})\ \wedge\ \mathrm{ALU}\ (\mathrm{c}_a, \mathrm{i}_1, \mathrm{i}_2, \mathrm{o}),\end{aligned}$$

consists of three registers and one ALU, which might be specified by

$$\begin{aligned}&\mathrm{Reg}\ (c, i, o)\ \equiv\\ &\qquad \forall t.\ o\,(t+1) = (c\,t \Rightarrow i\,t\ |\ o\,t)\\ &\mathrm{ALU}\ (c, i_1, i_2, o)\ \equiv\ \forall t.\ o\,(t+1) =\\ &\qquad (c\,t \Rightarrow \mathrm{ADD}\ (i_1\ t, i_2\ t)\ \downarrow\ i_1\ t),\end{aligned}$$

where $a\ \Rightarrow\ b\ \downarrow\ c$ abbreviates the term "if a then b else c" [19]. All in all we get an implementation as seen the figure below.



The point we wish to make is that in the traditional verification model in which components may communicate only spatially via signals the refinement to the final implementation in terms of data-path and control must be done in *one* step. The intermediate design stage as given by (2)–(4) in which the decomposition into subcomponents has been performed already, but the particular mode of control has neither as yet been fixed nor localized in a separate hardware component, cannot be represented naturally and directly. In other words, the construction model of hardware has the potential for a finer-grained design, *i.e.* for breaking the refinement process into smaller pieces.

Note again that in the description (2)–(4) different instantiations of the entry and leaving predicates would give rise to different realizations of control such as by synchronous clock or asynchronous handshake. Also, a pipelined operation is possible. If, in the abstract microprogram above, we specialize E_i such that $\forall t.\ E_i(t)$ we get a pipelined implementation in which the registers and the ALU process three different values of data at every point in time; we obtain the behaviour $\forall t.\ rec\,(t+3) = ADD(i_1\,(t), i_2\,(t))$. The concrete microprogram chosen above, in contrast, realizes only a single thread of control: the microinstruction address $address\,(t)$ is unique and cannot be 12, 13, and 14 at the same time.

This variability of instantiating the entry and leaving predicates is very useful in practice. For instance, to simulate the functional behaviour one can take a short-cut via a direct instantiation to a synchronous sequential program. In parallel, one may proceed with the actual design path towards a more sophisticated asynchronous pipelined hardware realization, along a number of further refinement steps.

From this viewpoint the entry and leaving predicates are a very flexible means to change or adapt the semantics of a design to a particular mode of operation, and thus go beyond the capabilities of traditional Hoare-style *pre/post conditions* or Lamport-style *assumption/commitment* forms. In fact, we conjecture that both can be incorporated in terms of entry and leaving predicates. However, this analysis is outside the scope of the paper.

To finish off this section let us sum up our definitions formally. Extending what has been introduced above, the refinements will use a construction model in which the leaving predicate is generalized to a *leaving predicate structure*. This allows us to express non-linear control structure in terms of entry and leaving predicates. With this extension the abstract syntax is as given in Fig. 1. In the following we will call any instance of the scheme in Fig. 1 a construction model or simply a model.

The abstract syntax is schematic in the sense that the syntactic classes mp, E, P are not further specified. They are not of interest at this stage and will only be filled up later with more structure when this is necessary to support a particular refinement step. Also, to keep matters simple, we will focus on the formal, read: syntactic, aspects of our constructions and leave aside semantical issues and correctness arguments. It deserves mention, however, that all refinements and transformations presented are based on sound logical arguments. As far as 'meaning' is concerned we content ourselves with a translation into an ambient (higher-order) predicate logic, which is assumed, at an intuitive level at least, to be understood throughout.

In this vein, the semantics of the construction model is 'defined' by the following inductive translation:

$$(mp\,[E, lps])^* = \forall x.\exists y.\ E\,x \Rightarrow mp \wedge lps^*$$

Construction Model	Syntactic Class:
$cm ::= \; mp\,[E, lps]$	cm : construction model
$lps ::= \; E$	lps : leaving predicate structure
$\mid P\;(lps,\; lps)$	
	mp : model predicate
	E : (entry or leaving) predicate
	P : Boolean term

Figure 1: Abstract Syntax of the Construction Model

$$(P(lps_1, lps_2))^* = P \Rightarrow lps_1^* \downarrow lps_2^*$$
$$E^* = E\,y.$$

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 $\forall x.\exists y.\, E_2\,x \;\Rightarrow\; pc\,y \;=\; pc\,x \;\wedge\; (b_1\,x \;\Rightarrow\; E_3\,y \downarrow (b_2\,x \Rightarrow E_4\,y \downarrow E_5\,y))$.
Note, this syntactic translation hardwires the special rôle of the time variables x, y as the generic start and end time points of a construction model: formally speaking, the operator $[E, E]$ is a binding operator for x and y which are assumed to be distinguished throughout this paper. For this to work, of course, we must make sure x and y are used consistently according to this convention. This is but one of a number of other additional well-formedness conditions on our specifications which are not contained in the abstract grammar schemes. In the following all specification schemes are to be understood modulo some tacit well-formedness conditions, which for simplicity will not be mentioned, however.

The construction model $mp[E_i, lps]$ being introduced the refinement process may be viewed as a repeated process of protruding particular structure for the body mp of the model which then is broken down along this structure into smaller pieces $mp_1[E_{1i}, lps_{1o}] \wedge \cdots \wedge mp_n[E_{ni}, lps_{no}]$ until the mp_i can be considered implementable. The main means of decomposition — as explored in this paper — is to break up terms syntactically into sub-terms and to introduce new entry predicates and new ports. For convenience we adopt the following assumption:
Assumption [existential variables]
All new entry predicates and new signals (ports) introduced in the construction process are existential variables.
With this assumption we may drop the existential quantifier $\exists z.$ hiding an internal signal in a composite circuit and simply turn z into an existential variable. The same is done with internal entry and leaving predicates.
Existential variables are supported in some theorem provers, like LAMBDA [14] ('flexible' variables) or ISABELLE [33] ('meta' variables) and have been shown to be of great advantage in practice. In particular the work based on LAMBDA (e.g. [14]) makes heavy use of existential variables to support formal hardware synthesis.

3 Machine Instruction Level

In this section a generic form of behaviour specification for a class of computers at machine instruction level is defined. At this level the semantics of machine instructions is given in terms of a global *state transition*. To capture a class of computers we use abstract syntax for the general structure of this state transition. This abstract grammar is given in Fig. 2. The syntax is abstract because its terminal symbols are uninterpreted, albeit the relationship of the symbols is clearly defined. The syntax is a scheme for a class of computers because all concrete computers whose specifications are obtained as instantiations are treated uniformly along with the construction and refinement of the scheme. In other words, we are aiming at a higher-order refinement: we are not interested in the refinement of a particular instance of a scheme but in the refinement of the scheme itself.

In the outermost layer the behaviour specification scheme of Fig. 2 describes a computer as a global state transition that is to be maintained for all points in time:

$(os = iss)\ [E, E]$.

Here os is the global output state at time point y and iss the input state structure expressing the result of executing a machine cycle in the flexible time interval $\langle x, y\rangle$. Typically, the entry and leaving predicates will be identical, which closes the global loop of instruction execution. The output state is assumed to be composed from a fixed number of state signals $sg_1, \ldots, sg_n$, *viz.*

$os = (sg_1\ y, \ldots, sg_n\ y)$.

The input state structure iss is a term over a number of input states

$is = (sis_1, \ldots, sis_n)$,

each of which is assumed to have the same number of sub-input states sis_i as os has components. These input states represent individual machine instructions where sis_i is the function that computes the i-th output state. A sub-input state sis is a term tm built from constants c, variables v, the values of signals sg at time point x, and by applying functions to terms. The input states is (or machine instructions) are organized to form the input state structure iss by applying the let_in "let A in iss" and the conditional "$P \Rightarrow iss \downarrow iss$". This models the control part of the machine cycle in which the right machine instruction is selected and set up for execution. In "let A in iss" the binding $A\ ::=\ v = tm$ declares a simple variable v to replace a term tm used in the input states of iss. The condition structure "$P \Rightarrow iss_1 \downarrow iss_2$" chooses for iss_1 if P is true and for iss_2 if it is false.

As can be seen the specification scheme is a refinement of our general construction model $mp\ [E, E]$ which protrudes intricate structure for the model predicate mp, and also specializes the leaving predicate structure. Note, that the primitive components Inv, Del, Reg, ALU mentioned before are special if not trivial instances of the specification scheme 1. They are what we will call *term transitions* consisting of a single equation of the form $sos = sis$. The goal of our constructions in the next section essentially is to break down an arbitrary instance of scheme 1 into a set of (parallel and sequential) term transitions. But before we get into the details let us make things definite by way of an example.

Example [Gordon's computer]

Gordon's computer is a simple general-purpose computer introduced as a nontrivial exercise in the formal specification and verification of hardware [17, 28]. The top-level specification of the computer at machine instruction level is what an assembler language programmer would need to write a program. At the top level Gordon's computer has a memory and two registers, *viz.* the program counter PC and the accumulator ACC. The specification includes two parts: the front panel operation and the machine instructions.

The front panel of the computer has a stop button, a four-position knob, and data 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 com-

Specification Scheme 1:			*Syntactic Class:*	
$spsc_1$	$::=$	$os = iss\ [E, E]$	$spsc_1$	: specification scheme1
os	$::=$	$(sos, \cdots, sos)$	os	: output state
sos	$::=$	$sg\ y$	sos	: sub-output state
iss	$::=$	is	iss	: input state structure
		$\mid$ let A in iss		
		$\mid P \Rightarrow iss \downarrow iss$		
is	$::=$	$(sis, \cdots, sis)$	is	: input state
sis	$::=$	tm	sis	: sub-input state
A	$::=$	$v = tm$	A	: binding
P	$::=$	$p\ (tm, \cdots, tm)$	P	: Boolean term
		$\mid \neg P$		
		$\mid P \circ P$		
tm	$::=$	c	tm	: term
		$\mid v$	v	: variable
		$\mid sg\ x$	sg	: state signal
		$\mid f\ (tm, \cdots, tm)$	f	: function
$\circ$	$::=$	$\wedge$	$\circ$	: binary relation
		$\mid \vee$		
		$\mid \Rightarrow$		
		$\mid \Leftrightarrow$		
			E	: (entry or leaving) predicate
			x, y	: distinguished time variables
			p	: predicate
			c	: constant

Figure 2: Specification Scheme 1: Machine Instruction Level

puter is idling is determined by the position of the knob according to the following table:

Knob position	Button action
0	Load PC
1	Load ACC
2	Store ACC at PC
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:

Code	Instruction	Effect
0	HALT	Stop execution
1	JMP L	Jump to addr. L
2	JZR L	Jump to addr. L if ACC = 0
3	ADD L	Add contents of addr. L to ACC
4	SUB L	Subtract contents of addr. L from ACC
5	LD L	Load contents of addr. L into ACC
6	ST L	Store contents of ACC in addr. L
7	SKIP	Skip to next instr.

This description for Gordon's computer may be rendered formal by the specification given in Fig. 3, following [28] closely. The meaning of the functions Val_2, Cut_{16_13}, Store_{13}, Opcode, Fetch_{13} is exactly as in [28]. T and F are the Boolean constants for true and false. The specification easily is seen to be an instance of our generic scheme 1.

4 Formal Construction

A step of formal construction is a transformation from one specification to another and the main aspect of this transformation is the decomposition of an abstract specification into less abstract sub-specifications. The process is repeated until concrete and minimal pieces of specification are met that are considered executable or directly implementable. This (well-known) idea can be lifted to specification schemes, so that the formal construction specifies and verifies the design process itself rather than a particular piece of hardware.

In this section we discuss the high stages of the formal construction, the refinement from the machine instruction level to the term transition level. This involves a number of smaller refinement steps via intermediate levels of description, in which a number of different design decisions are introduced and higher-level structures are realized in terms of lower-level structures. Instead of going through all of these steps we will focus on only one intermediate level, which is sufficient to illustrate the basic idea. The complete treatment can be found in [40].

Starting from the specification scheme 1 (Fig. 2), among other less central ones, the following four main refinement steps are to be taken:

(1) Setting State Transition: The global state transition $os = iss$ is distributed into local state transitions, which would correspond to single machine instructions.

$$
\begin{array}{l}
\mathrm{Computer}(\mathit{button}, \mathit{knob}, \mathit{switches}, \mathit{memory},\\
\qquad\qquad \mathit{pc}, \mathit{acc}, \mathit{idle}) \;\equiv\\
((\mathit{memory}\ y, \mathit{pc}\ y, \mathit{acc}\ y, \mathit{idle}\ y) =\\
\ (\mathit{idle}\ x \Rightarrow \mathit{Idle} \downarrow \mathit{Busy}))\ [E, E]\\
\\
\mathit{Idle} \mathrel{\widehat{=}}\\
\quad \mathit{button}\ x \Rightarrow\\
\quad ((\mathrm{Val}_2\ (\mathit{knob}\ x) = 0) \Rightarrow\\
(\mathit{memory}\ x,\ \mathrm{Cut}_{16\_13}\ (\mathit{switches}\ x),\ \mathit{acc}\ x,\ \mathrm{T}) \downarrow\\
\quad (\mathrm{Val}_2\ (\mathit{knob}\ x) = 1) \Rightarrow\\
(\mathit{memory}\ x,\ \mathit{pc}\ x,\ \mathit{switches}\ x,\ \mathrm{T}) \downarrow\\
\quad (\mathrm{Val}_2\ (\mathit{knob}\ x) = 2) \Rightarrow\\
(\mathrm{Store}_{13}\ (\mathit{pc}\ x)\ (\mathit{acc}\ x)\ (\mathit{memory}\ x),\\
\ \mathit{pc}\ x,\ \mathit{acc}\ x,\ \mathrm{T}) \downarrow\\
(\mathit{memory}\ x,\ \mathit{pc}\ x,\ \mathit{acc}\ x,\ \mathrm{F})) \downarrow\\
(\mathit{memory}\ x,\ \mathit{pc}\ x,\ \mathit{acc}\ x,\ \mathrm{T})\\
\\
\mathit{Busy} \mathrel{\widehat{=}}\\
\quad \mathit{button}\ x \Rightarrow\\
(\mathit{memory}\ x,\ \mathit{pc}\ x,\ \mathit{acc}\ x,\ \mathrm{T}) \downarrow\\
\quad \text{let } \mathit{op} =\\
\mathrm{Val}_3\ (\mathrm{Opcode}\ (\mathrm{Fetch}_{13}\ (\mathit{memory}\ x)\ (\mathit{pc}\ x)))\\
\quad \text{in let } \mathit{addr} =\\
\mathrm{Cut}_{16\_13}\ (\mathrm{Fetch}_{13}\ (\mathit{memory}\ x)\ (\mathit{pc}\ x))\ \text{ in}\\
\quad (\ (\mathit{op} = 0) \Rightarrow\\
(\mathit{memory}\ x,\ \mathit{pc}\ x,\ \mathit{acc}\ x,\ \mathrm{T}) \downarrow\\
\qquad (\mathit{op} = 1) \Rightarrow\\
(\mathit{memory}\ x,\ \mathit{addr},\ \mathit{acc}\ x,\ \mathrm{F}) \downarrow\\
\qquad (\mathit{op} = 2) \Rightarrow\\
\qquad ((\mathrm{Val}_{16}\ (\mathit{acc}\ x) = 0) \Rightarrow\\
(\mathit{memory}\ x,\ \mathit{addr},\ \mathit{acc}\ x,\ \mathrm{F}) \downarrow\\
(\mathit{memory}\ x,\ \mathrm{Inc}_{13}\ (\mathit{pc}\ x),\ \mathit{acc}\ x,\ \mathrm{F})) \downarrow\\
\qquad (\mathit{op} = 3) \Rightarrow\\
(\mathit{memory}\ x,\ \mathrm{Inc}_{13}\ (\mathit{pc}\ x),\\
\ \mathrm{Add}_{16}\ (\mathit{acc}\ x)\ \ (\mathrm{Fetch}_{13}\ (\mathit{memory}\ x)\ \mathit{addr}),\\
\ \mathrm{F}) \downarrow\\
\qquad (\mathit{op} = 4) \Rightarrow\\
(\mathit{memory}\ x,\ \mathrm{Inc}_{13}\ (\mathit{pc}\ x),\\
\ \mathrm{Sub}_{16}\ (\mathit{acc}\ x)\ \ (\mathrm{Fetch}_{13}\ (\mathit{memory}\ x)\ \mathit{addr}),\\
\ \mathrm{F}) \downarrow\\
\qquad (\mathit{op} = 5) \Rightarrow\\
(\mathit{memory}\ x,\ \mathrm{Inc}_{13}\ (\mathit{pc}\ x),\\
\ \mathrm{Fetch}_{13}\ (\mathit{memory}\ x)\ \mathit{addr}, \mathrm{F}) \downarrow\\
\qquad (\mathit{op} = 6) \Rightarrow\\
(\mathrm{Store}_{13}\ \mathit{addr}\ (\mathit{acc}\ x)\ (\mathit{memory}\ x),\\
\ \mathrm{Inc}_{13}\ (\mathit{pc}\ x),\ \mathit{acc}\ x,\ \mathrm{F}) \downarrow\\
(\mathit{memory}\ x,\ \mathrm{Inc}_{13}\ (\mathit{pc}\ x),\ \mathit{acc}\ x,\ \ \mathrm{F}))
\end{array}
$$

Figure 3: Specification of Gordon's Computer at Machine Instruction Level

Formally, the global equation $os = iss$ is pushed into the input state structure iss to get local equations $os = is$, and then distributed to individual state components to get a conjunction of *sub-state transitions* $sos = sis$. The conjunction we call a *sub-state transition set.*
For example, in Gordon's computer the global state equation

$$\begin{array}{l} (memory\ y, pc\ y, acc\ y, idle\ y) = \\ \quad idle\ x \Rightarrow (\ button\ x \Rightarrow (\cdots \Rightarrow \cdots \\ \qquad (memory\ x,\ pc\ x,\ switches\ x,\ \mathrm{T}) \\ \qquad\qquad\qquad \downarrow \cdots) \downarrow \cdots) \downarrow Busy \end{array}$$

is translated into

$$\begin{array}{l} idle\ x \Rightarrow (\ button\ x \Rightarrow (\cdots \Rightarrow \cdots \\ \quad (memory\ y = memory\ x\ \wedge \\ \quad\ pc\ y = pc\ x\ \wedge \\ \quad\ acc\ y = switches\ x\ \wedge \\ \quad\ idle\ y = \mathrm{T}) \\ \qquad\qquad\qquad \downarrow \cdots) \downarrow \cdots) \downarrow Busy \end{array}$$

where the resulting conjunction

$$memory\ y = memory\ x\ \wedge\ \cdots\ \wedge\ idle\ y = \mathrm{T}$$

is a sub-state transition set.

(2) Rearranging let_in structure: The let_in binding is our means to break up terms into pieces, *i.e.* undo syntactic substitution. In our later refinements the binding "$A = (v = tm_1)$" in "let A in $sos = tm_2$" is going to be realized by a separate process which sequentially precedes the one generated for "$sos = tm_2$", while the functionally equivalent "$sos = tm_2[tm_1/v]$" results in a single process. Thus, by rearranging the let_ins we can factor out separate subcomputations and control the granularity of primitive computation steps.
For example, in Gordon's computer we may introduce a new let_in such as

let $v = \mathrm{Fetch}_{13}\ (memory\ x)\ (pc\ x)$ in

to replace the two let_in constructs

let $op = \mathrm{Val}_3\ (\mathrm{Opcode}\ (\mathrm{Fetch}_{13}\ \cdots))$ in

let $addr = \mathrm{Cut}_{16\_13}\ (\mathrm{Fetch}_{13}\ \cdots)$ in

and substitute Val_3 (Opcode v) to replace op and $\mathrm{Cut}_{16\_13}\ v$ to replace $addr$. This is sensible because we consider Val_3, Opcode, and Cut_{16_13} as directly implementable, whence no separate computation step is necessary.

(3) Identifying Common Blocks: After setting the state transitions we identify subsets of sub-state transitions which are common to more than one sub-state transition set. These *common blocks* can be factored out and realized by separate processes. This step is the equivalent of rearranging let_in structures at the level of sub-state transitions.
To give an example, in Gordon's computer the sub-state transition
$sst_5 = (pc\ y = \mathrm{Inc}_{13}\ (pc\ x)\ \wedge\ idle\ y = \mathrm{F})$
is shared by a number of machine instructions such as the code for $op = 3, 4, 5, 7$, and thus only needs to be implemented once.

(4) Selection for Parallel and Sequential Execution: In general, the term/subterm relationship does not prejudice the actual execution ordering. In a nested conditional $P_1 \Rightarrow (P_2 \Rightarrow iss_1 \downarrow iss_2) \downarrow iss_3$ for instance, the evaluation of P_1 and P_2 can be either sequential or parallel. In one case both P_1 and P_2 are executed within the same (flexible) time interval, in the other in consecutive intervals. There are many possible ways of distinguishing these cases syntactically. We have chosen to introduce curly brackets { and } for delimiting the borders of parallel execution. Then, the parallel version of the conditional would become $\{P_1 \Rightarrow (P_2 \Rightarrow \{iss_1\} \downarrow \{iss_2\}) \downarrow \{iss_3\}\}$. Similarly we use brackets to select the parallel or sequential execution of nested let_in constructs. In let A_1 in let A_2 in iss the two bindings A_1 and A_2 can be executed in parallel whenever they do not depend on each other. In such a case we

$$
\begin{array}{l}
\mathrm{Computer}(\mathit{button}, \mathit{knob}, \mathit{switches}, \mathit{memory}, \\
\qquad\qquad \mathit{pc}, \mathit{acc}, \mathit{idle}) \;\equiv \\
\quad (\mathit{idle}\ x \Rightarrow \{\mathit{Idle}\} \downarrow \{\mathit{Busy}\})\ [E, E] \\
\\
\mathit{Idle} \;\widehat{=}\; \mathit{button}\ x \Rightarrow \\
\quad \{\ (\mathrm{Val}_2\ (\mathit{knob}\ x) = 0) \Rightarrow \{\mathit{st}_0\} \downarrow \\
\quad\quad (\mathrm{Val}_2\ (\mathit{knob}\ x) = 1) \Rightarrow \{\mathit{st}_1\} \downarrow \\
\quad\quad (\mathrm{Val}_2\ (\mathit{knob}\ x) = 2) \Rightarrow \{\mathit{st}_2\} \downarrow \{\mathit{st}_3\}\}\ \downarrow \\
\quad \{\mathit{st}_4\} \\
\\
\mathit{Busy} \;\widehat{=}\; \mathit{button}\ x \Rightarrow \\
\quad \{\mathit{st}_4\} \downarrow \\
\quad \{\mathrm{let}\ v = \mathrm{Fetch}_{13}\ (\mathit{memory}\ x)\ (\mathit{pc}\ x)\ \mathrm{in} \\
\quad\quad \{\ (\mathrm{Val}_3\ (\mathrm{Opcode}\ v) = 0) \Rightarrow \{\mathit{st}_4\} \downarrow \\
\quad\quad\quad (\mathrm{Val}_3\ (\mathrm{Opcode}\ v) = 1) \Rightarrow \{\mathit{st}_5\} \downarrow \\
\quad\quad\quad (\mathrm{Val}_3\ (\mathrm{Opcode}\ v) = 2) \Rightarrow \\
\quad\quad\quad\quad \{(\mathrm{Val}_{16}\ (\mathit{acc}\ x) = 0) \Rightarrow \{\mathit{st}_5\} \downarrow \\
\qquad\qquad\qquad\qquad\qquad\qquad \{\mathit{sst}_0 \wedge [\mathit{sst}_5]\}\} \\
\quad\quad\quad (\mathrm{Val}_3\ (\mathrm{Opcode}\ v) = 3) \Rightarrow \{\mathit{sst}_1 \wedge [\mathit{sst}_5]\} \downarrow \\
\quad\quad\quad (\mathrm{Val}_3\ (\mathrm{Opcode}\ v) = 4) \Rightarrow \{\mathit{sst}_2 \wedge [\mathit{sst}_5]\} \downarrow \\
\quad\quad\quad (\mathrm{Val}_3\ (\mathrm{Opcode}\ v) = 5) \Rightarrow \{\mathit{sst}_3 \wedge [\mathit{sst}_5]\} \downarrow \\
\quad\quad\quad (\mathrm{Val}_3\ (\mathrm{Opcode}\ v) = 6) \Rightarrow \{\mathit{sst}_4 \wedge [\mathit{sst}_5]\} \downarrow \\
\quad\quad\quad \{\mathit{sst}_0 \wedge [\mathit{sst}_5]\}\}\}
\end{array}
$$

Figure 4: Gordon's Computer at the Intermediate Level

would write $\{\mathrm{let}\ A_1\ \mathrm{in}\ \mathrm{let}\ A_2\ \mathrm{in}\ \{\mathit{iss}\}\}$ while the strictly sequential reading is written $\{\mathrm{let}\ A_1\ \mathrm{in}\ \{\mathrm{let}\ A_2\ \mathrm{in}\ \{\mathit{iss}\}\}\}$. A similar selection is introduced into the specification at the level of sub-state transition sets, with the help of square brackets: while $\mathit{ssts}_1 \wedge \mathit{ssts}_2$ means that the sub-state transition sets ssts_1 and ssts_2 are to executed in parallel, $\mathit{ssts}_1 \wedge [\mathit{ssts}_2]$ separates them to be executed sequentially (in left-to-right order). The state transitions within each ssts_i are executed in parallel, however.

For example, in Gordon's computer the condition predicates $\mathrm{Val}_2\ (\mathit{knob}\ x) = i$ for $i = 0, \ldots, 3$ may be selected for parallel execution because they are mutually disjoint and easily implemented by a simple branch structure. The same is true for the predicates $\mathrm{Val}_3\ (\mathrm{Opcode}\ (\cdots)) = i$ where $i = 0, \ldots, 7$.

A complete specification of Gordon's computer at the intermediate level is given in Fig. 4. It is the result of applying the refinements (1)–(4) to the specification in Fig. 3. The sub-state transition sets arising in Fig. 4 are abbreviated as follows:

$$
\begin{array}{lll}
\mathit{st}_0 & \widehat{=} & (\ \mathit{memory}\ y = \mathit{memory}\ x\ \wedge \\
 & & \quad \mathit{pc}\ y = \mathrm{Cut}_{16\_13}\ (\mathit{switches}\ x)\ \wedge \\
 & & \quad \mathit{acc}\ y = \mathit{acc}\ x\ \wedge \\
 & & \quad \mathit{idle}\ y = \mathrm{T}) \\
\mathit{st}_1 & \widehat{=} & (\ \mathit{memory}\ y = \mathit{memory}\ x\ \wedge \\
 & & \quad \mathit{pc}\ y = \mathit{pc}\ x\ \wedge \\
 & & \quad \mathit{acc}\ y = \mathit{switches}\ x\ \wedge \\
 & & \quad \mathit{idle}\ y = \mathrm{T}) \\
\mathit{st}_2 & \widehat{=} & (\ \mathit{memory}\ y = \\
 & & \quad\ \mathrm{Store}_{13}\ (\mathit{pc}\ x)\ (\mathit{acc}\ x)\ (\mathit{memory}\ x)\ \wedge \\
 & & \quad \mathit{pc}\ y = \mathit{pc}\ x\ \wedge \\
 & & \quad \mathit{acc}\ y = \mathit{acc}\ x\ \wedge \\
 & & \quad \mathit{idle}\ y = \mathrm{T}) \\
\mathit{st}_3 & \widehat{=} & (\ \mathit{memory}\ y = \mathit{memory}\ x\ \wedge \\
 & & \quad \mathit{pc}\ y = \mathit{pc}\ x\ \wedge \\
 & & \quad \mathit{acc}\ y = \mathit{acc}\ x\ \wedge \\
 & & \quad \mathit{idle}\ y = \mathrm{F}) \\
\mathit{st}_4 & \widehat{=} & (\ \mathit{memory}\ y = \mathit{memory}\ x\ \wedge \\
 & & \quad \mathit{pc}\ y = \mathit{pc}\ x\ \wedge \\
 & & \quad \mathit{acc}\ y = \mathit{acc}\ x\ \wedge \\
 & & \quad \mathit{idle}\ y = \mathrm{T}) \\
\mathit{st}_5 & \widehat{=} & (\ \mathit{memory}\ y = \mathit{memory}\ x\ \wedge \\
 & & \quad \mathit{pc}\ y = \mathrm{Cut}_{16\_13}\ (v\ x)\ \wedge \\
 & & \quad \mathit{acc}\ y = \mathit{acc}\ x\ \wedge \\
 & & \quad \mathit{idle}\ y = \mathrm{F}) \\
\mathit{sst}_0 & \widehat{=} & (\ \mathit{memory}\ y = \mathit{memory}\ x\ \wedge \\
 & & \quad \mathit{acc}\ y = \mathit{acc}\ x) \\
\mathit{sst}_1 & \widehat{=} & (\ \mathit{memory}\ y = \mathit{memory}\ x\ \wedge \\
 & & \quad \mathit{acc}\ y = \mathrm{Add}_{16}\ (\mathit{acc}\ x) \\
 & & \qquad\qquad (\mathrm{Fetch}_{13}\ (\mathit{memory}\ x) \\
 & & \qquad\qquad\qquad (\mathrm{Cut}_{16\_13}\ v))) \\
\mathit{sst}_2 & \widehat{=} & (\ \mathit{memory}\ y = \mathit{memory}\ x\ \wedge \\
 & & \quad \mathit{acc}\ y = \mathrm{Sub}_{16}\ (\mathit{acc}\ x) \\
 & & \qquad\qquad (\mathrm{Fetch}_{13}\ (\mathit{memory}\ x) \\
 & & \qquad\qquad\qquad (\mathrm{Cut}_{16\_13}\ v))) \\
\mathit{sst}_3 & \widehat{=} & (\ \mathit{memory}\ y = \mathit{memory}\ x\ \wedge \\
 & & \quad \mathit{acc}\ y = \mathrm{Fetch}_{13}\ (\mathit{memory}\ x) \\
 & & \qquad\qquad (\mathrm{Cut}_{16\_13}\ v)) \\
\mathit{sst}_4 & \widehat{=} & (\ \mathit{memory}\ y = \mathrm{Store}_{13}\ (\mathrm{Cut}_{16\_13}\ v) \\
 & & \qquad\qquad (\mathit{acc}\ x) \\
 & & \qquad\qquad (\mathit{memory}\ x)\ \wedge \\
 & & \quad \mathit{acc}\ y = \mathit{acc}\ x) \\
\mathit{sst}_5 & \widehat{=} & (\ \mathit{pc}\ y = \mathrm{Inc}_{13}\ (\mathit{pc}\ x)\ \wedge\ \mathit{idle}\ y = \mathrm{F}).
\end{array}
$$

Specification Scheme 2:			*Syntactic Class:*
$spsc2$	::=	$sts\ [E, E]$	$spsc2$: specification scheme2
sts	::=	st	sts : state transition structure
		$\mid$ let A in ls	
		$\mid P \Rightarrow cs \downarrow cs$	
ls	::=	let A in ls	ls : let_in structure
		$\mid \{sts\}$	
cs	::=	$P \Rightarrow cs \downarrow cs$	cs : condition structure
		$\mid \{sts\}$	
st	::=	$st\ \wedge\ [ssts]$	st : state transition
		$\mid ssts$	
		$\mid [ssts]$	
$ssts$	::=	sst	$ssts$: sub-state transition set
		$\mid sst \wedge ssts$	
sst	::=	$sos = sis$	sst : sub-state transition

Figure 5: Specification Scheme 2: Intermediate Level

In [40] the refinement steps (1)-(4) discussed above have been formalized rigorously in terms of transformations on specification schemes and their correctness has been verified. Obviously, these refinements are nondeterministic, in general, and need a good deal of user-guidance for the appropriate selections and transformations. To account for this the constructions defined in [40] provide a maximum-degree design-freedom operational framework in which dedicated algorithms and heuristics can be accommodated as special execution strategies.

As mentioned before, the detailed treatment of these construction steps is outside the scope of this paper. Here we only give the abstract syntax of the resulting intermediate specification scheme, see Fig. 5, and focus on the last refinement, from this intermediate level to the term transition level, in the next section.

5 Term Transition Level

At the term transition level a design $spsc_3$ is broken up into a conjunction of construction models $ttm_1 \wedge \cdots \wedge ttm_n$, in which the ttm_i are very simple cases of construction models:

$$ttm_i \;=\; (s_{i1}\ y = tm_{i1}\ \wedge\ \cdots \\ \wedge\ s_{ik_i}\ y = tm_{ik_i})\ [E_{ii}, lps_{io}]$$

The corresponding abstract syntax scheme 3 is given in Fig. 6. The significance of specification scheme 3 is that it embodies a novel high-level design stage, the *term transition level*, which is obtained by exact formal construction. This, so we believe, establishes a rather natural and generic level of abstraction between the machine instruction level above (in other words: specification scheme 1) and the function unit and connection level (in other words: register transfer level) below. The term transition level can be viewed as a generalization of the computational model of UNITY [8] or SYNCHRONIZED TRANSITIONS [34]. A term transition model ttm (*cf.* Fig. 6) in our case corresponds to a basic transition, *i.e.* a multi-assignment, in UNITY or SYNCHRONIZED TRANSITIONS. A set $ttms$ of term transition models, then, corresponds to a basic UNITY or SYNCHRONIZED TRANSITIONS program.

However, whereas in these languages all control is flattened away and effected implicitly through state changes, in our model a rather generic form of control structure is still explicit through the entry and leaving predicates.

We are now going to formalize the construction process that leads from the intermediate level to the term transition level as a transformation Ψ of Prolog style [11] from specification scheme 2 to specification scheme 3. The construction, a simplified variant of the one from [40], is shown in Fig. 7.

A few explanations will be helpful to understand Ψ. A *spsc*2 is always of the form $sts\ [E_i, E_o]$ where the state transition structure *sts* is one of three possibilities, dealt with by Ψ in the equations 4.1, 4.2 and 4.3: if *sts* is a state transition *st* it is passed on to the construction function Ψ_s in 4.1; if it is a let_in structure then 4.2 composes the result as the conjunction of two parts determined by the sub-constructions Ψ_{l1} and Ψ_{l2}, respectively; if it is a condition structure then equation 4.3 builds on the two sub-constructions Ψ_{p1} and Ψ_{p2}. In this way the construction proceeds further by induction on the structure of *spcs*2.

The construction is deterministic except for the generation of variables and the choice of their names. In 4.6 and 4.8 new (existential) signal variables s_v, and in 4.2, 4.4, 4.11, 4.13 new (existential) entry and leaving predicate variables are introduced. The choice of the predicate variable E_m in 4.2, 4.4 is arbitrary except that it must be fresh, *i.e.* not occur already anywhere else in the term (Since they are existential they always can be identified later if necessary). The same holds for variable s_v in 4.6 and 4.8. The choice of s_v, however, must be functional, *i.e.* be uniquely determined by variable v. Similarly, the choice of E_{sts} in 4.11 and 4.13 must be functional, *i.e.* (in some way) uniquely depend on the subterm *sts*. Another remark concerns 4.9: The right-hand side *sts L* denotes the (meta-level) application of the *substitution* L to term *sts*. The substitution L is initiated as the empty list in 4.2 and built up inductively in rule 4.8. The superscripts of $ssts^i, st^o, sts^i$ (4.4, 4.7) are meta-level operations which compute statements about the protection of input and output signals:

Definition [Input and output signal protection]
For a syntactic construct S the input protection is
$S^i \mathrel{\hat=} (r_1\ y = r_1\ x) \wedge \cdots \wedge (r_n\ y = r_n\ x)$,
where r_i $(1 \le i \le n)$ are all the input signals in S. Similarly,
$S^o \mathrel{\hat=} (s_1\ y = s_1\ x) \wedge \cdots \wedge (s_m\ y = s_m\ x)$,
where s_i $(1 \le i \le m)$ are all the output signals, is the output protection for S. In the special case where S does not contain any input or output signals we take the boolean constant T. □

The specification of Gordon's computer at the term transition level is shown in Fig. 8. It is an instance of specification scheme 3 and formally derived by the refinements discussed. It can be obtained by applying Ψ to the specification in Fig. 4 at intermediate level (we also unify appropriate entry and leaving predicates and remove duplicate conjuncts).

From the derived specification of Gordon's computer at term transition level we get a clear picture about how the term transition specification describes a high-level abstract computer architecture. At this level one or a group of term transitions, such as st_2, is taken as a 'primitive' component whose internal structure is left to further construction. Scheme 3 reveals the logical dependency between the term transitions, which may in fact be viewed as an abstract mi-

Specification Scheme 3:			*Syntactic Class:*	
$spsc_3$	::=	$ttms$	$spsc_3$	: specification scheme 3
$ttms$	::=	ttm	$ttms$	: term transition model set
		$\mid ttm \wedge ttms$		
ttm	::=	$tts\ [E_i,\ lps]$	ttm	: term transition model
tts	::=	tt	tts	: term transition set
		$\mid tt \wedge tts$		
tt	::=	$s\ y = tm$	tt	: term transition
			tm	: term
			s	: signal
			lps	: leaving predicate structure

Figure 6: Specification Scheme 3: Term Transition Level

Construction:

4.1 $\Psi\ (st\ [E_i,\ E_o]) \mathrel{\widehat{=}} \Psi_s\ (((\mathrm{T}) \wedge st)\ [E_i,\ E_o])$

4.2 $\Psi\ ((\text{let } A \text{ in } ls)\ [E_i,\ E_o]) \mathrel{\widehat{=}}$
$(\Psi_{l1}\ (\text{let } A \text{ in } ls))\ [E_i,\ E_m] \ \wedge\ \Psi\ ((\Psi_{l2}\ (\text{let } A \text{ in } ls,\ [\,]))\ [E_m,\ E_o])$

4.3 $\Psi\ ((P \Rightarrow cs_1 \downarrow cs_2)\ [E_i,\ E_o]) \mathrel{\widehat{=}}$
$tts\,[E_i,\ \Psi_{p1}\ (P \Rightarrow cs_1 \downarrow cs_2,\ tts)] \ \wedge\ \Psi_{p2}\ (P \Rightarrow cs_1 \downarrow cs_2,\ E_o)$

4.4 $\Psi_s\ (((K) \wedge st \wedge [ssts])\ [E_i,\ E_o]) \mathrel{\widehat{=}}$
$\Psi_s\ (((ssts^i \wedge K) \wedge st)\ [E_i,\ E_m]) \ \wedge\ (st^o \wedge K \wedge ssts)\ [E_m,\ E_o]$

4.5 $\Psi_s\ (((K) \wedge \gamma)\ [E_i,\ E_o]) \mathrel{\widehat{=}} (K \wedge \gamma')\ [E_i,\ E_o]$
where $\gamma' = ssts$ if $\gamma = [ssts]$ and $\gamma' = \gamma$ otherwise

4.6 $\Psi_{l1}\ (\text{let } v = tm \text{ in } ls) \mathrel{\widehat{=}} s_v\ y = tm \wedge \Psi_{l1}(ls)$

4.7 $\Psi_{l1}\ (\{sts\}) \mathrel{\widehat{=}} sts^i$

4.8 $\Psi_{l2}\ (\text{let } v = tm \text{ in } ls,\ L) \mathrel{\widehat{=}} \Psi_{l2}\ (ls,\ [s_v\ x/v] :: L)$

4.9 $\Psi_{l2}\ (\{sts\},\ L) \mathrel{\widehat{=}} sts\ L$

4.10 $\Psi_{p1}\ (P \Rightarrow cs_1 \downarrow cs_2,\ tts_1 \wedge tts_2) \mathrel{\widehat{=}} P\ (\Psi_{p1}\ (cs_1,\ tts_1),\ \Psi_{p1}\ (cs_2,\ tts_2))$

4.11 $\Psi_{p1}\ (\{sts\},\ sts^i) \mathrel{\widehat{=}} E_{sts}$

4.12 $\Psi_{p2}\ (P \Rightarrow cs_1 \downarrow cs_2,\ E_o) \mathrel{\widehat{=}} \Psi_{p2}\ (cs_1,\ E_o) \wedge \Psi_{p2}\ (cs_2,\ E_o)$

4.13 $\Psi_{p2}\ (\{sts\},\ E_o) \mathrel{\widehat{=}} \Psi\ (sts\ [E_{sts},\ E_o])$

Figure 7: Formal Construction of the Refinement

croprogram. A primitive component corresponds to a linear microprogram and what the scheme 3 describes is the connection relation of these linear microprograms. The abstract flowchart for Gordon's computer is seen in Fig. 9, where for conciseness we omit the obvious signal protections. Note that since every 'primitive' term transition set tts is again an instance of scheme 1 the whole refinement process from scheme 1 to scheme

$$
\begin{aligned}
&\mathrm{Computer}(\mathit{button}, \mathit{knob}, \mathit{switches}, \mathit{memory},\\
&\qquad \mathit{pc}, \mathit{acc}, \mathit{idle}) \equiv\\
&\; kss_0\,[E,\ (\mathit{idle}\ x)\ (E_1,\ E_2)] \wedge\\
&\; kss_1\,[E_1,\ (\mathit{button}\ x)\ (E_3,\ E_4)] \wedge\\
&\; kss_2\,[E_2,\ (\mathit{button}\ x)\ (E_4,\ E_5)] \wedge\\
&\; kss_3\,[E_3,\ (\mathrm{Val}_2\ (\mathit{knob}\ x) = 0)\\
&\qquad (E_6,\ (\mathrm{Val}_2\ (\mathit{knob}\ x) = 1)\\
&\qquad\quad (E_7,\ (\mathrm{Val}_2\ (\mathit{knob}\ x) = 2)\\
&\qquad\qquad (E_8,\ E_9)))] \wedge\\
&\; (s_v\ y = \mathrm{Fetch}_{13}\ (\mathit{memory}\ x)\ (\mathit{pc}\ x) \wedge\\
&\qquad kss_4)\,[E_5,\ E_{10}] \wedge\\
&\; kss_5\,[E_{10},\ (\mathrm{Val}_3\ (\mathrm{Opcode}\ (s_v\ x)) = 0)\\
&\quad (E_4,\ (\mathrm{Val}_3\ (\mathrm{Opcode}\ (s_v\ x)) = 1)\\
&\quad\; (E_{11},\ (\mathrm{Val}_3\ (\mathrm{Opcode}\ (s_v\ x)) = 2)\\
&\qquad (E_{12},\ (\mathrm{Val}_3\ (\mathrm{Opcode}\ (s_v\ x)) = 3)\\
&\qquad\; (E_{13},\ (\mathrm{Val}_3\ (\mathrm{Opcode}\ (s_v\ x)) = 4)\\
&\qquad\quad (E_{14},\ (\mathrm{Val}_3\ (\mathrm{Opcode}\ (s_v\ x)) = 5)\\
&\qquad\quad\; (E_{15},\ (\mathrm{Val}_3\ (\mathrm{Opcode}\ (s_v\ x)) = 6)\\
&\qquad\qquad (E_{16},\ E_{17})))))))] \wedge\\
&\; kss_5\,[E_{12}, (\mathrm{Val}_{16}\ (\mathit{acc}\ x) = 0)\ (E_{11},\ E_{18})] \wedge\\
&\; st_0\,[E_6,\ E] \wedge st_1\,[E_7,\ E] \wedge st_2\,[E_8,\ E] \wedge\\
&\; st_3\,[E_9,\ E] \wedge st_4\,[E_4,\ E] \wedge st_5\,[E_{11},\ E] \wedge\\
&\; (sst_1 \wedge kss_6)\,[E_{13},\ E_{19}] \wedge\\
&\; (sst_2 \wedge kss_6)\,[E_{14},\ E_{19}] \wedge\\
&\; (sst_3 \wedge kss_6)\,[E_{15},\ E_{19}] \wedge\\
&\; (sst_4 \wedge kss_6)\,[E_{16},\ E_{19}] \wedge\\
&\; (sst_0 \wedge kss_6)\,[E_{17},\ E_{19}] \wedge\\
&\; (sst_0 \wedge kss_6)\,[E_{18},\ E_{19}] \wedge\\
&\; (sst_5 \wedge kss_7)\,[E_{19},\ E]
\end{aligned}
$$

$$
\begin{aligned}
&kss_0 \mathrel{\hat{=}} \mathit{button}\ y = \mathit{button}\ x \wedge kss_1\\
&kss_1 \mathrel{\hat{=}} \mathit{knob}\ y = \mathit{knob}\ x \wedge kss_3\\
&kss_2 \mathrel{\hat{=}} kss_4\\
&kss_3 \mathrel{\hat{=}} \mathit{switches}\ y = \mathit{switches}\ x \wedge kss_4\\
&kss_4 \mathrel{\hat{=}} kss_6 \wedge kss_7\\
&kss_5 \mathrel{\hat{=}} s_v\ y = s_v\ x \wedge kss_4\\
&kss_6 \mathrel{\hat{=}} \mathit{pc}\ y = \mathit{pc}\ x\\
&kss_7 \mathrel{\hat{=}} \ \mathit{memory}\ y = \mathit{memory}\ x \wedge\\
&\qquad\quad \mathit{acc}\ y = \mathit{acc}\ x
\end{aligned}
$$

Figure 8: Gordon's Computer Refined to Term Transition Level

3 can be reiterated locally for *tts* if this appears appropriate. This means we can break up the terms by introducing and rearranging let_ins, identify common blocks, and select for parallel and sequential execution of subterms.

It is worthwhile to recall that although we have refined the computer to a microprogram level it is abstract in that it still has a large degree of design freedom. Depending on how we decide to interpret (or instantiate) the entry and leaving predicates we can get implementations quite different from a microprogrammed computer such as an abstract synchronous or asynchronous circuit (*cf.* Sec. 2).

In [40] a complete sequence of refinement steps has been worked out along similar lines, which from the microinstruction level leads all the way down to a concrete implementation at register-transfer level involving 10 major levels of specification schemes together with 9 construction steps. We have presented only one of these steps in this paper. All of these construction steps are rigorously defined and formally verified. If Ψ_n is a (nondeterministic) construction from a scheme $\mathcal{S}_n$ to a scheme $\mathcal{S}_{n+1}$, then it is verified that for every instance S_n of scheme $\mathcal{S}_n$ the construction (guided by external choices) will produce a well-formed instance S_{n+1} of scheme $\mathcal{S}_{n+1}$ (= syntactic correctness and completeness), and further it is verified that $\vdash S_{n+1} \Rightarrow S_n$ (= semantical correctness). The framework has been applied to construct several different versions of Gordon's computer.

6 Conclusion

We have presented a framework for the formal design of a class of microprogrammed computers based on the formal refinement of abstract specification schemes. We used Gordon's computer to illustrate our ideas. We believe that this approach offers a feasible path towards the rigorous specification and verification of hardware synthesis systems, which still remains a major challenge [2, 3, 6, 7, 23, 21, 22, 25, 37, 38, 41]. Two

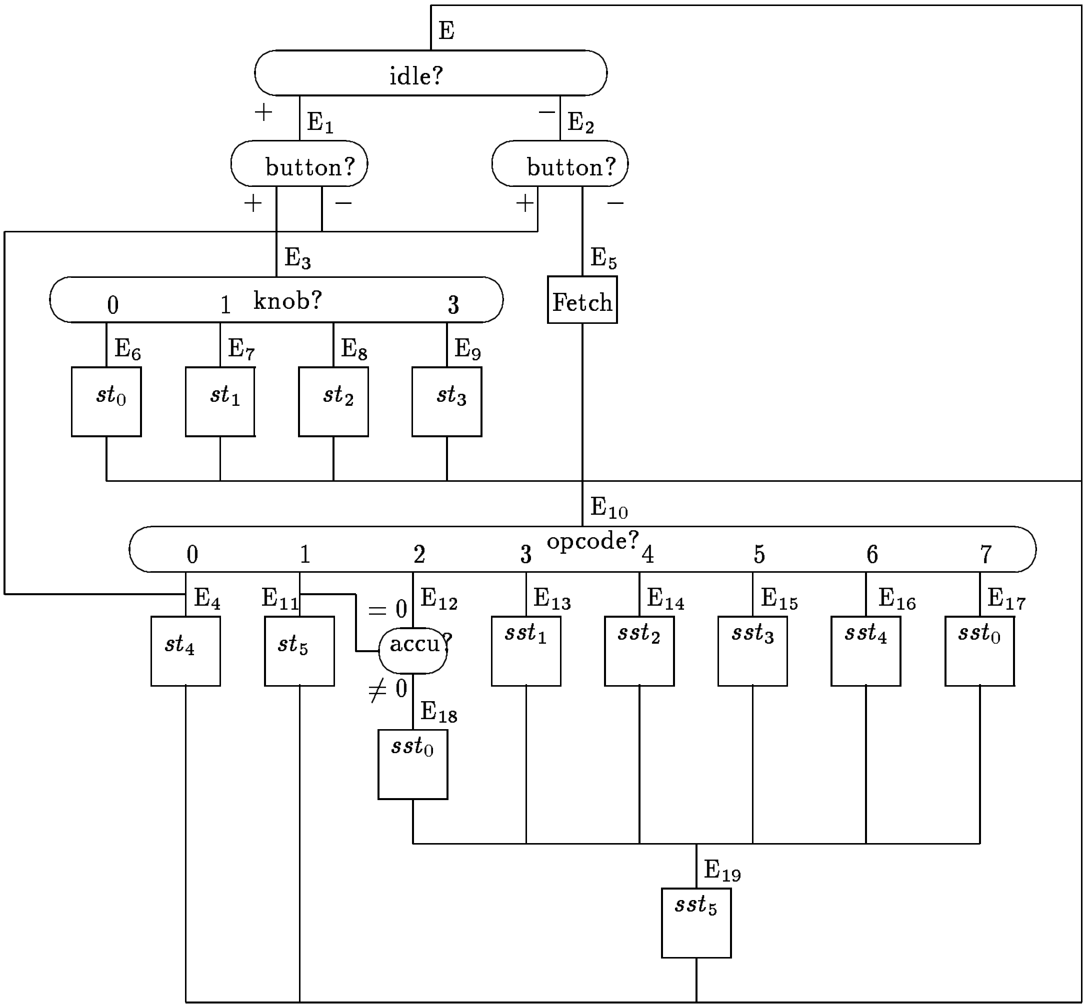


Figure 9: Flowchart of Gordon's Computer at Term Transition Level

technical contributions are made: (1) We introduce a novel construction model of hardware by which the synthesis process can be split into smaller refinement steps compared to the traditional components-as-predicates verification model. (2) With the term transition level our formal construction establishes a new and natural abstraction level of computer architecture, conveniently between microinstruction and register-transfer level.

Our basic methodology stresses the importance of abstract syntax to describe the generic structure of a class of computers. At a more general level the difference between construction (*qua* abstract syntax schemes) and verification (*qua* logic formulas) [41] is less important. What matters is to reveal the fundamental structure of our research objects and their essential relationships.

Further work will be in two directions. We are developing a semi-automatic computer design system which can design a register-transfer level microprocessor from a specification at machine instruction level. The user interaction consists in the selections for high-level allocation and scheduling. Other work aims at the theoretical foundations of our framework, in particular concerning the algebraic and logical properties of the construction model.

Acknowledgements Li-Guo Wang is indebted to Mike Fourman for his encouragement and stimulating supervision of the author's Ph.D. research on which this work is built. We would like to thank Kees Goossens for his comments on a draft of this paper. Thanks are also due to Rocco De Nicola for inviting Michael Mendler to the University of Rome, La Sapienza, where a part of this work was carried out.

References

[1] Simon Bainbridge, Albert Camilleri and Roger Fleming, Theorem Proving as an Industrial Tool for System Level Design. In [36], pp. 253-276.

[2] David A. Basin, Geoffrey M. Brown, and Miriam E. Leeser, Formally verified synthesis of combinational CMOS circuits. In L. J. M. Claesen, editor, *Formal VLSI Specification and Synthesis*, page 197-206, North-Holland, 1990.

[3] D. A. Basin, Extracting Circuits from Constructive Proofs. In 1991 International Workshop on *Formal Verification in VLSI Design.* ACM IFIP WG 10.2, Jan. 1991.

[4] G. Birtwistle and P. Subrahmanyam, eds., *VLSI Specification, Verification and Synthesis*, Kluwer Academic Publishers, Boston, 1988.

[5] J. Bormann, H. Nusser-Wehlan, and G. Venzl, Formal Design in an Industrial Research Laboratory: Lessons and Perspectives. In J. Staunstrup and R. Sharp, eds., Second Workshop on *Designing Correct Circuits*, Lynbgy, Denmark, Jan. 1992 WG10.2, WG 10.5.

[6] D. Borrione, H. Collavizza and C. Le Faou, μSPEED: A Framework for Specifying and Verifying Microprocessors. In Int. Workshop on *Formal Methods in VLSI Design*, 9-11 Jan. 1991, Miami, Springer-Verlag.

[7] B. Bose and S. D. Johnson, DDD-FM9001: Derivation of a Verified Microprocessor. In [32].

[8] K. M. Chandy and J. Mishra, *Parallel Program Design. A Foundation.* Addison Wesley, 1988.

[9] J. Chazarain and H. Collavizza, Combining Symbolic Evaluation and Object Oriented Approach for Verifying Processor-like Architectures at the RT-level. In [32], pp. 109 - 121.

[10] L. Claesen, editor, IMEC-IFIP International Workshop on *Applied Formal Methods for Correct VLSI Design*, Volume 1+2, Elsevier Science Publishers, B.V. North-Holland, Amsterdam, 1989.

[11] W. F. Clocksin and C. S. Mellish, *Programming in Prolog*, third, revised and extended edition, Springer-Verlag, 1987.

[12] Avra Cohn, A Proof of Correctness of the Viper Microprocessor: First Level. In [4], pp. 27-71.

[13] Avra Cohn, Correctness Properties of the Viper Block Model: The Second Level. In G. Birtwistle and P. Subrahmanyam, eds., *Current Trends in Hardware Verification and Automated Theorem Proving*, Springer-Verlag, 1989, pp.1-91.

[14] M. P. Fourman, R. L. Harris, Lambda-Logic and Mathematics Behind Design Automation, 26th ACD/IEEE Design Automation Conference, 1988.

[15] M. P. Fourman, Formal System Design. In [35], pp 191-236.

[16] Gopalakrishnan, Genesh and Fujimoto, Richard, Design and Verification of the Rollback Chip Using HOP: a Case Study of Formal Methods Applied to Hardware

Design. ACM Transactions on Computer Systems, Vo. 11 No. 2 pp. 109–145, May 1993.

[17] M. J. C. Gordon, Proving a Computer Correct with the LCF_LSM Hardware Verification System. Technical Report No. 42, Computer Laboratory, University of Cambridge, 1983.

[18] M. J. C. Gordon, Why higher-order logic is a good formalism for specifying and verifying hardware. in: G. Milne and P. Subrahmanyam, eds., *Formal Aspects of VLSI Design*, Proceedings of the 1985 Edinburgh Conference on VLSI, North-Holland, 1986, pp. 153-177.

[19] M. J. C. Gordon, HOL: A Proof Generating System for Higher-Order Logic. University of Cambridge, Computer Laboratory, Tech Report No. 103, 1987.

[20] F.K. Hanna and N.Daeche, Specification and Verification of Digital Systems using Higher-Order Predicate Logic. IEE Proceedings, Vol. 133, Part E, No. 5, September 1986, pp. 242-254.

[21] F.K. Hanna, M. Longley, and N. Daeche, Formal synthesis of digital systems. In [10], pages 532-548.

[22] F.K. Hanna and N.Daeche. Strongly-Typed Theory of Structure and Behaviors. In [32], pp. 39-54.

[23] N.A. Harman and J. V. Tucker, Algebraic Models and the Correctness of Microprocessors. In [32], pp. 92-108.

[24] Warren A. Hunt, FM8501, *A Verified Microprocessor*. Ph.D. Thesis, Report No. 47, Institute for Computing Science, University of Texas, Austin, December 1985.

[25] P.B. Jackson, Nuprl and its Use in Circuit Design. In [36], pp. 311-336.

[26] G. Jones and M. Sheeran, Circuit Design in RUBY. In [35].

[27] M. Langevin and E. Cerny, Verification of Processor-like Circuits. In Proc. of the *Advanced Work on Correct Hardware Design Methodology*, Turin, 12-14 June 1991.

[28] J. J. Joyce, G. Birtwistle and M. Gordon, Proving a Computer Correct in Higher Order Logic. Report No. 100, Computer Laboratory, Cambridge University, 1986.

[29] Mahmood, M. and Mavaddat, F. and Elmasry, M.I. and Cheng, M.H.M, A Formal Language Model of Logic Microcode Synthesis. In [10], pp. 21–39.

[30] Thomas F. Melham, Abstraction mechanism for hardware verification. In G. Birtwistle and P.A. Subrahmanyam, eds., *VLSI Specification, Verification, and Synthesis,* pages 267-291. Kluwer Academic Publishers, 1988.

[31] G. J. Milne, Design for Verifiability. In Proc. Workshop on *Hardware Specification, Verification and Synthesis: Mathematical Aspects*, Cornell University, USA, July 1989.

[32] G. J. Milne and L. Pierre, eds., *Correct Hardware Design and Verification Methods*, LNCS 683, Springer-Verlag, May 1993.

[33] Lawrence C. Paulson. *Isabelle Tutorial and User's Manual*, 1990.

[34] J. Staunstrup, *A Formal Approach to Hardware Design*. Kluwer Academic Publishers, 1994.

[35] J. Staunstrup, editor, IFIP WG 10.5 *Formal Methods for VLSI Design*, North-Holland, 1990.

[36] V. Stavridou, T.F. Melham, and R.T. Boute, eds., *Theorem Provers in Circuit Design: Theory, Practice and Experience.* IFIP TC10/WG 10.2, North Holland, June 1992.

[37] V. Stavridou, J. A. Goguen, A. Stevens, S. M. Eker, S. N. Alonefits, and K. M. Hobley, FUNNEL and 2OBJ: Towards and Integrated Hardware Design Environment. In [36].

[38] Dany Suk, Hardware Synthesis in Constructive Type Theory. In G. Jones and M. Sheeran, eds., *Designing Correct Circuits*, pp 29-49, Oxford, Springer-Verlag, 1990.

[39] Li-Guo Wang, Deriving a Correct Computer. In L. J. M. Claesen and M. J. C. Gordon, eds., *IFIP Transactions, Higher Order Logic Theorem Proving and its Application,* pages 449-458, Elsevier Science Publisher B.V. North-Holland, Amsterdam 1993.

[40] Li-Guo Wang, *Formal Derivation of A Class of Computers.* PhD Thesis, LFCS, Department of Computer Science, University of Edinburgh, Oct. 1994.

[41] P. J. Windley, A Hierarchical Methodology for the Verification of Microprogrammed Microprocessors. In Proceedings of the IEEE Symposium on *Security and Privacy*, May 1990.