Abstraction of Hardware Construction

August 21, 1995

Li-Guo Wang[1]
Room 1414,
Department of Computer Science,
University of Edinburgh,
JCMB, The King's Buildings,
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 concentrate on the foundation of formal hardware construction and present a new hardware-centered concept and methodology. It is a development from, and also contract to, the traditional logic-centered approach. It introduces a higher-order variable construction model of hardware and the notion of generic construction schemes. We demonstrate how the model and the schemes formalize the construction (and verification as well) for a class of computers, as an example to illustrate our concepts and method.

1 Introduction

1.1 From Application to Theory

The formal method for hardware design presented in this paper was stimulated primarily by application, concretely by the formal derivation of a class of microcomputers [39]. The search for a formal approach that is adequate at a non-academic scale on realistic computers, naturally lead us to reconsider and depart from the rather low level of formalization currently used in the area. While often "formal method" is reduced to the application of a specific calculus of formal logic or specific theorem prover, in our work is it taken in its straight meaning: as the formal manipulation of a class of objects (here computers at different levels of abstraction,) where 'formal' refers to the fact that these objects are represented by abstract syntax; the objects themselves, or rather their behaviours, are given by a well-defined semantics outside of the formal calculus. In this paper we will summarize the central aspects of this work.

1.2 Synthesis versus Verification

The classical application of formal methods to the design of microcomputers is concerned with *formal verification*, *i.e.* the process of proving the consistency between a given implementation

[1]This research is supported by a scholarship from Siemens AG Munich and under SERC GR/F 35890 U.K.
[2]The author was supported by a Human Capital and Mobility fellowship in the *EuroForm* network.

and a given specification. The most well-known examples are the verification of the FM8501 by Hunt [21], of the VIPER by Cohn [9, 10], and of the TAMARACK-3 by Joyce [24]. A number of verification examples followed these pioneering works, such as the SECD [3], Mini Cayuga [31], DLX [36], just to mention a few. The approaches mainly differ in the number of abstraction levels they consider, and what logic or theorem prover they use. However, they all share the essential feature of *post-hoc* formal verification: specification, implementation and all intermediate level descriptions are given *a priori*, while the nontrivial task consists in finding a proof of correctness for each step.

When initially the research focused on the correctness of specific well-defined hardware designs, the importance of *genericity* was soon realized, and the inherent potential of the formal approach to verify whole classes of related hardware was stressed [37, 25, 20]. This trend towards proof genericity is natural not least because of the great efforts required for and the resource-demanding nature of formal verification. A side-effect of the strive for generality is to identify and formalize common hardware structure and abstraction levels along which the verification task can be organized and systematized. Examples of such work for the context of microcomputers is the notion of generic interpreter [41, 42] or the algebraic models of [19].

There are many ways to make formal proofs more generic. However, the ultimate level of genericity is not obtained by formal verification but by *formal synthesis*. In contrast to verification, formal synthesis subjects the very description of a computer to the process of construction not only the correctness proof. This, of course, offers maximal genericity. It also pays pragmatic tribute to the fact that in real life a specification is never completely determined. One usually starts off merely with a first approximation, which is later refined and modified as more details of the implementation become known.
The idea of applying formal methods to the design process itself has been introduced rather early, for instance with the theorem provers Lambda [11, 12] or Veritas [17]. Though quite some work has been done on the formal synthesis of actual hardware, *e.g.* [35, 1, 2, 6], the formal synthesis of realistic (or at least nearly realistic) computers remains an open challenge. A first attempt in this direction is the formal derivation of the DDD-FM9001 [5].

As we see it, the current way of realizing formal synthesis must be seen essentially as a refinement of the tradition of formal verification. Thus, it suffers from being limited into the corset of a particular logic or theorem prover. For instance, [35] is based on Martin-Löf type theory implemented in Isabelle [30], [1, 2] is based on Martin-Löf type theory implemented in Nuprl, [6] is based on higher-order logic implemented in the Lambda prover, and finally [5] uses (a fragment of) first-order logic plus induction implemented the Boyer-Moore theorem prover. We believe that in order for formal synthesis to get off the ground this is the wrong level of formalization. After all, the main concern of synthesis should be the construction of correct computers, not in fact the construction of proofs in some logic on some theorem prover. A large stock of practical experience in formal verification has shown that for a good-structured design, once this structure has been identified, the actual verification may be quite a trivial task. We believe that synthesis in first instance should deal with the formal construction and refinement of hardware structure, and only in second instance with the construction of proofs.

1.3 Hardware-Centered versus Logic-Centered

The previous substantial efforts in applying formal methods to the hardware design process are *logic centered*, typically so are those based on Hol [15], Lambda [11] and Veritas [17]. These tools are all more or less directly motivated by their application to hardware, and yet they are full-fledged general-purpose theorem provers. For example, regarding HOL we have had several international conferences which showed various interesting applications of HOL in very different areas. A similar statement is true for LAMBDA. The first author successfully used LAMBDA to the mechanical synthesis of nondeterministic logic programs [38]. Thus, the traditional applications of formal methods to hardware are mainly centered around very powerful logic systems whose expressiveness is beyond the relatively narrow needs of typical hardware designs. Therefore, we may call these approaches logic-centered.

However, it appears that the strength of the logic-centered approach is at the same time its main weakness: it is far too general to deal with realistically complex hardware designs in an efficient way. This problem is hinted at in [18]:

> *"[...] it is still, 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."*

Concretely, the weakness of the encode-it-in-a-general-purpose-theorem-prover approach lies in the low level of syntactic description and the expensive granularity of elementary design steps. Synthesizing a computer in terms of logical connectives and modus ponens is similar to programming an operating system in an assembler language. More subtly, the generic structure of computers is too rich to be handled by simple formula schemes. For instance, consider a specification of the machine instruction cycle in terms of a global next-state function, such as

$$\begin{array}{rcl} \mathrm{state}\,(t+1) & = & \mathrm{opcode}\,t = c_1 \;\rightarrow\; \mathrm{instruction}_1\;(\mathrm{state}\,t) \;\downarrow \\ & & \mathrm{opcode}\,t = c_2 \;\rightarrow\; \mathrm{instruction}_2\;(\mathrm{state}\,t) \;\downarrow \\ & & \qquad\qquad \cdots \\ & & \mathrm{opcode}\,t = c_n \;\rightarrow\; \mathrm{instruction}_n\;(\mathrm{state}\,t), \end{array}$$

where $P \rightarrow Q \downarrow R$ stands for "if P then Q else R." A simple refinement of this specification might consist in pushing the top-level equation inwards, say in order to separate control and data:

$$\begin{array}{rcl} \mathrm{opcode}\,t = c_1 & \rightarrow & \mathrm{state}\,(t+1) = \mathrm{instruction}_1\;(\mathrm{state}\,t) \;\downarrow \\ \mathrm{opcode}\,t = c_2 & \rightarrow & \mathrm{state}\,(t+1) = \mathrm{instruction}_2\;(\mathrm{state}\,t) \;\downarrow \\ & \cdots & \\ \mathrm{opcode}\,t = c_n & \rightarrow & \mathrm{state}\,(t+1) = \mathrm{instruction}_n\;(\mathrm{state}\,t). \end{array}$$

Now suppose we wanted to implement this step on a theorem prover, as a refinement that is generic in the number n of machine instructions. The most direct solution, namely to exploit the elementary genericity of syntactic substitution, does not work. The syntax of standard logic is not discriminative enough to define the class of all well-formed specifications as the substitution instances of a formula scheme (and the refinement as instances of a rule scheme).
In principle, there are two opposite directions we can take to achieve genericity. The solution usually adopted is to implement the refinement as a tactic. This, however, pushes the formalization to the meta-level at the price of a significant "*correctness leak*": who verifies the tactic? Although we can be sure that tactics are correct relative to the low level of formulas, there is no guarantee that they are actually sensible as refinements of hardware. A tactic must neither be too timid, so that it succeeds in all interesting cases, nor must it be too general, so that it does not mess up the

proof state. Perhaps the most well-known instance of this problem is the so-called *false antecedent problem*: the synthesis tactics may introduce short-circuits or other inconsistencies. To specify and verify a tactic is a nontrivial problem which is rarely, if ever, attempted.
The second possibility to achieve genericity is by encoding the generic structure in terms of inductive data-types, say natural numbers or lists, and recursive functions and induction rules. Such an approach is suggested for example in [20, 18]. This way of going about avoids the correctness leak of tactics. However, by pushing the important structure into the object level it introduces a "*formalization leak*": Explicit structure becomes implicit and the manipulation of this structure (*e.g.* by recursion equations) must be implemented *within* rather than *by* the calculus. In this way the formalization becomes clumsy and in the extreme case may get close to a Gödel numbering of hardware.

When it comes to applying formal methods to hardware design then choosing the right level of formalization is a rather crucial part of the exercise. We believe that to deal with real world designs the current logic-centered approach is too low level and that, therefore, a more refined approach is needed. This new approach should be *hardware-centered* and restricted to the hardware construction area of interest, rather than being general purpose. The purpose of this paper is to present such an approach.

1.4 The New Concepts

The main concepts advanced in this paper are the notions of specification scheme, construction scheme, and construction model. They form the basis for a formal construction framework that has a status very similar to a formal logic, but with a higher level of description, larger degree of genericity, and focus on a particular class of hardware.
Specification schemes are abstract syntax schemes to cover the generic structure of a class of concrete computers, at different levels of abstraction. They are the objects of our constructions just as formulas are the objects manipulated by a theorem prover. The semantics of a specification scheme is induced by a syntactic translation into higher-order logic.
A *construction scheme* is a generic translation of one specification scheme to another. It might formalize a refinement from one level of description to another one, or a semantics-preserving transformation. The application of construction schemes drives the formal synthesis just as rules drive the derivation of formulas in a theorem-prover.
Finally, the *construction model*, in our framework, is the most generic form of a specification scheme. It is meant to capture the behaviour of a hardware component as a combination of a functional aspect and a control aspect. This model applies to the whole construction process of a class of computers as shown in [39, 40]. In contrast to the traditional verification-oriented hardware model, 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.

1.5 The Higher-Order Aspect

The hardware-centered concept and methodology presented in this paper can be implemented in several logic systems such as HOL [15], LAMBDA [11], or ISABELLE [30]. As its purpose is to handle generic structure, a minimum of higher-order features is required for an adequate implementation of (inductive) specification and construction schemes. Also, we use higher-order variables for the logic (entry predicates) and space (signal/port) parameters in the construction model.

2 Specification Schemes

Construction is inherently formal and generic. It is *formal* in the sense that it manipulates syntactic structure. The track of the construction process is a series of syntactic descriptions, called specifications, in our case of hardware, from a behavioural to a structural level. The construction is *generic* in two dimensions. The specification is a scheme that captures the common structure of a class of objects rather than that of a particular instance. This we call the *horizontal* genericity of construction. Moreover, a specification scheme captures the structure only at a particular level of description. It is built up from "primitive" components that are uninterpreted and whose structure is irrelevant at the given level. They may be filled in by later refinements. This we call the *vertical* genericity.

Propositional logic, viewed as a fragment of first-order logic, is a familiar example of a formal and generic calculus. This sub-calculus is sensitive to the structure of logical formulas only in terms of propositional connectives like conjunction and implication. The horizontal genericity lies in the inductive structure of propositional formulas as a subset of all formulas. The propositional formulas are generated from propositional variables as their primitive components. As far as the sub-calculus is concerned the structure of these primitive components is irrelevant. A propositional variable may stand for arbitrary logical sentences, which may be freely substituted for it. This is the vertical genericity of propositional logic (within first-order logic.)

To express specification schemes we use a formal language approach. A specification scheme is a context-free formal grammar presented in Backus-Naur form (BNF). Although BNF does not give us maximal control over syntactic structure, it does extend the possibilities of usual logic formulas considerably. Making use of the technology of abstract syntax BNF works well to keep all constructions and specification schemes generic and exact. To get an instance specification from the scheme we instantiate the uninterpreted primitive components (vertical instantiation) and the generic structure of the scheme (horizontal instantiation). So, the specification scheme is our main interface between formal construction and the real world. It should be generic enough to reflect the real world and exact enough, both in the sense of rigorous and informative, such that the formal construction can work on it.

Let us take a class of computers as an example. Taking a bird's eye view a computer is nothing but a big state transition function working on a number of state signals. We can fix this structure by the following simple specification scheme:

$$
\begin{array}{lcl}
comp & ::= & os\ =\ iss \\
os & ::= & (sg\ y, \ldots, sg\ y) \\
 & \mid & sg\ y
\end{array}
\qquad
\begin{array}{ll}
comp & : \text{computer} \\
os & : \text{output state} \\
iss & : \text{input-state structure} \\
sg & : \text{signal} \\
y & : \text{distinguished time variable.}
\end{array}
$$

The scheme is exact. It says that the output state os of an instruction cycle is computed according to an input-state structure iss. The output state is formed by applying a number of state signals $sg_1, \ldots, sg_n$ to a distinguished time variable y, which represents the completion time of the cycle. Thus, $os = (sg_1\, y, \ldots, sg_n\, y)$. The scheme is generic. It does not restrict how many state signals a computer has and what information is maintained in them. It does not say what the next-state structure is except that it can be evaluated to give a possible state of the computer at a particular point in time.

We may refine the specification scheme by protruding specific structure for iss. Let us adopt a programmer's model according to which a computer is a set of machine instructions. These instructions are executed under different conditions, *e.g.* depending on the value of the operation code (*cf.* Windley's notion of generic interpreters [42].) Let us suppose further that instructions may have sub-instructions in common. For such a class of computers, we may set up the following refinement:

$$
\begin{array}{lcl}
iss & ::= & is \\
 & \mid & \text{let } A \text{ in } iss \\
 & \mid & P \Rightarrow iss \downarrow iss \\
A & ::= & v = comm
\end{array}
\qquad
\begin{array}{ll}
is & : \text{input state} \\
A & : \text{sharing} \\
P & : \text{predicate} \\
comm & : \text{common part} \\
v & : \text{variable.}
\end{array}
$$

Though we have supplied more detail we are still generic: We do not restrict how many instructions a computer has, and in which language they are formulated; we have not said how the instructions are selected, *i.e.* what kind of predicates can be used. These are generic components of the scheme, left to be filled in in different ways by different instances from the real world. Different refinements and design methods may require different kind of structure for instructions and predicates. However, what is described exactly by the scheme is the the relation between these generic components. It is an inductive and recursive structure. Its base case is an instruction and the step case tells us that through let-in and condition structure we can get any computer, no matter how complex it is.

Although this is still quite an abstract scheme, certain construction steps may find enough structure to work on, already at this level of description, without the need to supply further details. Some constructions may push inwards the global equation $os = iss$ to localize the state transitions and separate control structure from data-structure (called *setting state transitions* in [39, 40].) Other constructions may rearrange, massage, or normalize the let-in and condition structure in iss. Yet another type of constructions can try break up the atomic calculation of iss into a collection of sequential and parallel computations.

At some point in the synthesis it will be important to protrude the details of the input state is, predicates P, and common part $comm$. There are many possibilities, depending on how powerful a language we wish to have and on what primitives we wish to build. For our purpose a simple first-order language of terms suffices:

$$
\begin{array}{lcl}
is & ::= & tm \\
 & \mid & (tm, \ldots, tm) \\
P & ::= & tm \\
comm & ::= & tm \\
tm & ::= & c \\
 & \mid & v \\
 & \mid & sg\ x \\
 & & \\
 & \mid & f(tm, \ldots, tm)
\end{array}
\qquad
\begin{array}{ll}
is & : \text{input state} \\
tm & : \text{term} \\
P & : \text{predicate} \\
comm & : \text{common part} \\
c & : \text{constant} \\
v & : \text{variable} \\
sg & : \text{signal} \\
x & : \text{distinguished time variable} \\
f & : \text{function.}
\end{array}
$$

According to this scheme an input state is a tuple of terms, and predicate and common parts are terms. A term is built up from constants, variables, the value of signals at a distinguished point in time (x stands for the start time of the instruction cycle,) and functions applied to terms.

Every construction works on a characteristic syntactic structure, which in an implementation might be established automatically by unification. However, the crucial point is that the specification schemes allow us to specify the effect and purpose of a particular construction in a rigorous way. It allows us to control genericity, which in traditional systems either goes astray in the meta-level of tactics or is obscured in the internals of a clumsy object-level encoding of the structure. The constructions considered in this paper require the structure that is made explicit by the above schemes for *comp*, *iss*, and *is*. We will call this level of description the *machine instruction level* (MIL).

A final remark is in order. Not all constraints we might wish to impose are (naturally) context-free. This is a limitation which sometimes forces us to add extra side-conditions to capture the well-formed specification instances completely. For example, we might want to constrain the variable v of a sharing 'let $v = tm$ in iss' to occur at least twice in iss (otherwise the v is unnecessary,) and further require that no two sharings must use the same variable. Further, we might restrict v not to appear in any *comm*. Though this might be too restrictive in other circumstances, for a computer's specification it is useful to eliminate nested or recursive sharing. Another side-condition requires that the output state os and all input states is have the same number of components. Such restrictions, which will often exist for a syntax scheme, could be handled formally too, are ignored in this paper to keep matters simple.

In order to be able to define (and verify) the correctness of construction steps we need some form of semantics. We adopt the simplest possibility: the meaning of a scheme is induced by a translation into an ambient higher-order predicate logic. This logic will not be formally introduced as it can be assumed, at an intuitive level at least, to be understood throughout. The translation is almost immediate in the sense that instances of our schemes can be viewed directly as formulas in higher-order logic containing some syntactic sugar. This would not be strictly necessary, but it stresses our point that the construction framework has the status of a formal logic.

The primitive elements of the specification scheme above, *e.g.* signals sg, the time variables x, y, and terms tm, are assumed to be part of the higher-order logic. Also, the meaning of composite structures like tuples and equations are taken directly from the logic. The let_in and conditional structures are translated as follows:

$$\begin{array}{rcl} (\text{let } v = A \text{ in } iss)^* & = & \exists v.\, v = A \ \wedge \ iss^* \\ (P \Rightarrow iss_1 \downarrow iss_2)^* & = & (P^* \ \wedge \ iss_1^*) \ \vee \ (\neg P^* \ \wedge \ iss_2^*). \end{array}$$

In this way the semantics of a computer *comp* is a predicate $comp^*$ over a number of free signal variables $sg_1, \ldots, sg_n$ and the two distinguished time variables x, y. Let us assume that time is represented by natural numbers $\mathbf{N}$, and that $\mathbf{N} \rightarrow \alpha_i$, for $i = 1, \ldots, n$, is the type of sg_i. Then $comp^*$ is a relation over $\Sigma \times \mathbf{N} \times \mathbf{N}$ where $\Sigma = (\mathbf{N} \rightarrow \alpha_1) \times \cdots \times (\mathbf{N} \rightarrow \alpha_n)$ is the set of possible state trajectories of the computer. Given $s \in \Sigma$ and $t_1, t_2 \in \mathbf{N}$, $(s, t_1, t_2) \in comp^*$ intuitively means that t_1 is the start time and t_2 the termination time of a machine cycle in trajectory s.

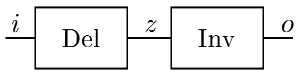
3 The Construction Model

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 [21] 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 [16, 14, 12], and the basis for formal hardware design methodologies such as [23]. 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: (i) Components are represented by predicates with the free variables as their ports, (ii) Internal port connection is realized by existential quantification, and (iii) 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 [28]:

> *"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 [9, 10, 27, 13, 26, 28]. To capture the top-down construction of *incomplete* hardware, in particular for the construction of microprogrammed computers, more sophisticated structure is needed. We have introduced such a new and more general hardware model, the *construction model* [40]. From our point of view the main aspect in the formal design process 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

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

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 constructive framework avoids these fallacies by keeping a rigid discipline 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 $\exists E_i, E_o.\; \mathrm{Inv}\,(z, o)\;[E_i,\; E_o]$.

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.
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 this paper.

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

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.

As mentioned in Sec. refsec:sc, as far as 'meaning' is concerned, we adopt the simplest possible approach, *viz.* a translation into an ambient (higher-order) predicate logic. In this vein, the semantics of the construction model is defined by the following inductive translation:

$$\begin{aligned}(mp\,[E, lps])^* &= \forall x.\exists y.\, E\,x \Rightarrow mp^* \wedge lps^* \\ (P(lps_1, lps_2))^* &= P \Rightarrow lps_1^* \downarrow lps_2^* \\ E^* &= E\,y,\end{aligned}$$

where mp^* is the translation of mp. 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.

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. In our case the model predicates will be instances of the scheme *comp* and conjunctions thereof. 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 [11] ('flexible' variables) or ISABELLE [30] ('meta' variables) and have been shown to be of great advantage in practice. In particular the work based on LAMBDA (e.g. [11]) makes heavy use of existential variables to support formal hardware synthesis.

4 Construction Schemes

Our objects, to be formally manipulated, are not just arbitrary formulas but computers. Just as a theorem prover usually does not derive ground formulas but formula schemes, our framework constructs specification schemes describing classes of computers (at different levels of description), rather than concrete instances.

The specification scheme is the high-level analogon to a formula scheme, the notion of a construction scheme discussed in this section is the high-level analogon to a derivation rule. An ordinary rule like

$$\forall x.\, \phi \wedge \forall x.\, \psi \quad \vdash \quad \forall x.\, (\phi \wedge \psi)$$

in predicate logic is not derived as an entailment relation between two concrete formulas but between formula schemes. The premise might be the schematic description of an implementation, and the conclusion that of a specification:

$$\mathit{Impl} \quad \vdash \quad \mathit{Spec}. \tag{1}$$

Then, the rule can be seen as a *construction scheme*, which allows us to construct from every concrete specification S a concrete implementation I, such that $I \vdash S$. Given S, the construction consists in computing a suitable substitution θ such that $S = \mathit{Spec}[\theta]$, from which the implementation can be obtained as $I = \mathit{Impl}[\theta]$. The computation of θ is usually done by some form of unification.

Our plan is to lift the same idea to the case where *Impl* and *Spec* are specification schemes, such as those considered in the previous sections. A rule, then, becomes a high-level construction. In this way we do not only raise the level of description. In addition to the vertical genericity of syntactic substitution, we get the horizontal genericity of the BNF syntax schemes. Of course, we cannot any longer represent a construction as a finite rule, though in trivial cases such as

$$\forall x_1 .\, \phi_1 \;\wedge\; \cdots \;\wedge\; \forall x_n .\, \phi_n \quad \vdash \quad \forall x_1 .\; \cdots\; \forall x_n .\, (\phi_1 \wedge\; \cdots \;\wedge \phi_n)$$

a pseudo-formal representation using indices and ellipses is possible. We need a different, more general, way of looking at a rule (1) where *Impl* and *Spec* are specification schemes. The most general interpretation is to view it as a function Ψ which translates (or: refines) any instance S of scheme *Spec* into an instance $\Psi(S)$ of scheme *Impl* such that $\Psi(S) \vdash S$ is provable. In other words, we use (1) as an abbreviation for the condition:

$$\Psi : \mathit{Spec} \to \mathit{Impl} \qquad \forall S \in \mathit{Spec}.\, \Psi(S) \vdash S.$$

If Ψ satisfies these condition, then we write

$$\Psi \quad : \quad \mathit{Impl} \vdash \mathit{Spec}.$$

Notice, the correctness involves two aspects: Ψ must be a *total* function from well-formed instances of *Spec* to well-formed instances of *Impl*, and it must be *sound*, *i.e.* the result of applying Ψ must be a provable refinement. Such a construction Ψ may be defined by recursion over the structure of *Spec* and its correctness verified by structural induction.

It is useful to be a bit more general and consider also non-deterministic constructions Ψ, which for a given input specification determine a *set* of refinements. In this case the correctness conditions have to be modified accordingly:

$$\forall S \in \mathit{Spec}.\, \emptyset \neq \Psi(S) \subseteq \mathit{Impl} \qquad \forall S \in \mathit{Spec}.\, \forall I \in \Psi(S).\; I \vdash S.$$

Nondeterministic constructions arise in an ordinary logic calculus naturally as a set of rules with overlapping conclusions, or by rules producing more than one matching for the conclusion. An example of the latter kind is the equality rule $x = e \;\wedge\; \psi \vdash \psi\{e/x\}$. In our framework the nondeterminism is essential, and modelling the potential external selections to steer the synthesis in one direction or other. This can be either by user interaction or by external heuristic algorithms, say for scheduling or allocation.

In this paper we concentrate on the syntactic structure decomposition aspects of construction. Mainly, we will discuss how to construct the logic, space, and time relation of hardware. However,

generally speaking, the construction of hardware includes not only syntactic but also semantical aspects. The main purpose of semantic construction steps, apart from optimization, is to make each function in the specification implementable *w.r.t.* a given implementation level. A syntactic step decomposes the specification to the implementation level.
For an example of a semantical step consider a non-implementable 13 bit counter Inc_{13} as in the model

$$(s\,y = \mathrm{Inc}_{13}\ (r\,x))\ [E_i,\, E_o].$$

It needs to be transformed semantically to the composition of three implementable functions Cut_{16_13}, Inc_{16}, and Pad_{13_16}:

$$(s\,y = \mathrm{Cut}_{16\_13}\,(\mathrm{Inc}_{16}\,(\mathrm{Pad}_{13\_16}\ (r\,x))))\ [E_i,\, E_o].$$

The syntactic aspect of a construction is to divide the specification into sub-specifications by introducing new entry predicates in logic, new signals (port) in space, and new points in time. For example, decomposing the instruction cycle

$$(s\ y = \mathit{execute}(\mathit{fetch}(\mathit{pc}\ x, \mathit{mem}\ x), \mathit{mem}\ x))\ [E, E]$$

into a separate fetch and execution phase

$$(r\ y = \mathit{fetch}(\mathit{pc}\ x, \mathit{mem}\ x)\ \wedge\ \mathit{mem}\ y = \mathit{mem}\ x)\ [E, E_a]\ \wedge\ (s\ y = \mathit{execute}(r\ x, \mathit{mem}\ x))\ [E_a, E]$$

is an example of a syntactic decomposition step. An atomic computation is made non-atomic by breaking it into two pieces. For this to be correct we need to add a *signal protection* constraint: the condition $\mathit{mem}\ y = \mathit{mem}\ x$ makes sure that the *fetch* and the *execute* phase work on an identical *mem* state.

Definition [Input and output signal protection] *For a given syntactic construct S the input signal protection S^i for S is $(r_1\ y = r_1\ x) \wedge \cdots \wedge (r_n\ y = r_n\ x)$, where r_i $(1 \le i \le n)$ are all the input signals in S. Similarly, $S^o \mathrel{\widehat{=}} (s_1\ y = s_1\ x) \wedge \cdots \wedge (s_m\ y = s_m\ x)$, 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.* □

Let us now look at a number of construction schemes that perform relevant syntactic decompositions, taking the specification scheme of the microinstruction level (see Sec. 2) as our starting point:

$$\mathit{spec} \quad ::= \quad \mathit{comp}\,[E, E].$$

It presents a computer as a dynamic process $\mathit{comp}\,[E, E] = (\mathit{os} = \mathit{iss})\ [E, E]$ that executes a global state transition $\mathit{os} = \mathit{iss}$ in a single loop $[E, E]$. A natural refinement goal is to implement the global transition in terms of a set of elementary local state transitions, which are executed partly in sequence and partly in parallel. To synchronize them the single loop will have to be broken up into richer control structure. In other words, we want a refinement

$$\Psi\ :\ \text{simple control} + \text{complex data} \rightarrow \text{complex control} + \text{simple data}.$$

where the domain coincides with the scheme *spec*. But what is the structure of our target specification scheme, capturing the idea of complex control and simple data? The simplest (nontrivial) form of a computer are the instances of the scheme $\mathit{tt}\ ::=\ \mathit{sg}\ y = \mathit{tm}$, *i.e.* those with a single output state signal and the input state structure consisting of a single term. We call the sub-class *tt* of *comp* the *term transitions.* With the term transitions our target scheme now can be specified by

$$impl \quad ::= \quad (tt^+ \; [E, lps])^+,$$

where for a given scheme $\mathcal{S}$, we use the notation $\mathcal{S}^+$ to stand for the finite, nonempty conjunctions of (instances of) $\mathcal{S}$. Thus, an implementation is a conjunction of models

$$\begin{array}{l}(s_{11}\ y = tm_{11}\ \wedge \cdots \wedge\ s_{1k_1}\ y = tm_{1k_1})\ [E_{1i}, lps_{1o}] \\ \quad \wedge \cdots \wedge \\ \qquad (s_{n1}\ y = tm_{n1}\ \wedge \cdots \wedge\ s_{nk_n}\ y = tm_{nk_n})\ [E_{ni}, lps_{no}].\end{array}$$

Intuitively, each model i consist of a set of term transitions $s_{i1}\ y = tm_{i1}\ \wedge \cdots \wedge\ s_{ik_1}\ y = tm_{ik_1}$ executed in parallel in one control step. These sets are synchronized by the $[E_{ii}, lps_{io}]$, which determine how the control steps are linked together. Recall that *lps* is the leaving predicate structure (see Sec. 2.)

The level of description expressed by the scheme *impl* has been called *term transition level* [39, 40]. It establishes, so we believe, a rather natural and generic level of abstraction between the machine instruction level above and the register transfer level below. The term transition level can be viewed as a generalization of the computational model of UNITY [7] or SYNCHRONIZED TRANSITIONS [33]. A term transition model $tt^+\ [E, lps]$ here corresponds to a basic transition, *i.e.* a multi-assignment, there. A conjunction $(tt^+\ [E, lps])^+$ 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.

To get back to our main theme, we are looking for a (nondeterministic) construction scheme Ψ that satisfies the specification $impl \vdash spec$, or more explicitly,

$$(tt^+\ [E, lps])^+\ \vdash\ comp\,[E, E].$$

As explained above, the correctness of Ψ amounts to the two conditions

$$\forall S \in comp\,[E,E].\ \emptyset \neq \Psi(S) \subseteq (tt^+\ [E, lps])^+ \qquad \forall S \in comp\,[E,E].\ \forall I \in \Psi(S).\ I \vdash S.$$

Usually a construction such as Ψ will not be defined in one step. It will be composed of a number of smaller constructions, which realize certain more or less independent aspects of the refinement. The composition is by way of certain combinators, which allow us to obtain new constructions from constructions already defined previously. In the following let us see how this would look like.

A first construction step might consist in transforming some of the data into explicit control structure. Formally, this is done by pushing the global equation inwards into iss. This way we can break up the input state structure iss into smaller pieces while at the same time producing a richer control structure, *i.e.* more equations. Let this construction be called Ψ^a. It can be defined as follows:

$$\begin{array}{ll}4.a.1 & \Psi^a(comp)\ \widehat{=}\ comp\ \mid\ \Psi^a_1(comp) \\ 4.a.2 & \Psi^a_1(os = is)\ \widehat{=}\ os = is \\ 4.a.3 & \Psi^a_1(os = \text{let } A \text{ in } iss)\ \widehat{=}\ \text{let } A \text{ in } \Psi^a(os = iss) \\ 4.a.4 & \psi^a_1(os = (P \Rightarrow iss_1 \downarrow iss_2))\ \widehat{=}\ P \Rightarrow \Psi^a(os = iss_1) \downarrow \Psi^a(os = iss_2).\end{array}$$

Ψ^a is a nondeterministic construction scheme defined by induction on the structure of *comp*. It serves to push the outermost equation across an arbitrary number of let-ins and conditionals. In the first clause, 4.a.1, the vertical bar denotes the choice either to stop the construction or to break out one more control structure (either let-in or conditional) with the sub-construction Ψ_1^a. The new specification scheme describing the result of this construction is as follows:

$$\begin{array}{rcl@{\qquad}ll} \mathit{comps} & ::= & \mathit{comp} & \mathit{comps} & : \text{computer structure} \\ & | & \text{let } A \text{ in } \mathit{comps} & \mathit{comp} & : \text{computer} \\ & | & P \Rightarrow \mathit{comps} \downarrow \mathit{comps} & A & : \text{sharing} \\ & & & P & : \text{predicate.} \end{array}$$

Theorem 1 *The construction Ψ^a satisfies the specification comps $\vdash$ comp, i.e. for all $S \in$ comp, $\emptyset \neq \Psi^a(S) \subseteq$ comps and for all $I \in \Psi^a(S)$, $I \vdash S$.*

We can apply Ψ^a directly, or under an entry and leaving predicate structure $[E, lps]$. In general, if a construction Θ satisfies $\mathcal{S} \vdash \mathcal{T}$, then the construction $\Theta' = \Theta\,[E, lps]$, defined as $\Theta'(T\,[E, lps]) = (\Theta(T))\,[E, lps]$, satisfies $\mathcal{S}\,[E, lps] \vdash \mathcal{T}\,[E, lps]$. For instance,

$$\Psi^a\,[E, lps] : \mathit{comps}\,[E, lps] \vdash \mathit{comp}\,[E, lps].$$

The operation $\Theta \mapsto \Theta\,[E, lps]$ is a first example of a combinator on constructions.

Once we have singled out some control structure with Ψ^a, or $\Psi^a\,[E, lps]$, we might want to decompose the computer so that the control becomes a separate computation. Before we get to the general case let us first discuss two special cases, *viz.* the decomposition of a condition and the decomposition of a let-in structure.

Case 1. [decomposing condition structure]: A condition structure is formed as

$$(P \Rightarrow R \downarrow S)\;[E_i,\ E_o],$$

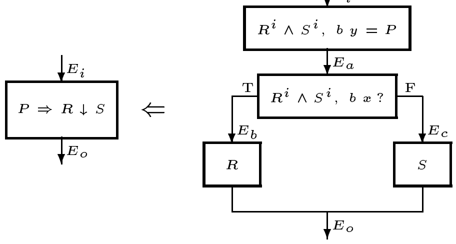
where P is a predicate with time variable x and R and S are predicates with time variables x and y. We can implement the conditional model in three stages:

$$(b\,y = P \wedge R^i \wedge S^i)\;[E_i,\ E_a]\ \wedge \tag{2}$$

$$(R^i \wedge S^i)\;[E_a,\ b\,x\,(E_b,\ E_c)]\ \wedge \tag{3}$$

$$R\;[E_b,\ E_o]\ \wedge\ S\;[E_c,\ E_o]. \tag{4}$$

The first component (2) realizes the computation of the predicate P and supplies its value on a new signal b. The second component (3), then, performs the branching in the leaving predicate structure. The third stage (4) is the computation of R or S, depending on whether control is passed to entry predicate E_b or E_c. Notice, this decomposition separates the *computation* of the condition from from its actual *execution*; the computation is in done in the data, the decision is part of control. The construction introduces predicate variables E_a, E_b and E_c together with their implicit time variables, and port variable b. As mentioned before these variables are existential variables. The decomposition may be illustrated in a flow-graph as follows:



Case 2. [decomposing let-in structure]: A let-in structure is formed as

$$(\text{let } v = tm \text{ in } S)\ [E_i,\ E_o],$$

where tm is a term (see Sec. 2) with time point variable x; v is a variable; S is a predicate with time point variables x and y. By decomposition we get

$$(s_v\, y = tm \wedge S^i)\ [E_i,\ E_a] \quad \wedge \tag{5}$$
$$S\ \{s_v\ x/v\}\ [E_a,\ E_o], \tag{6}$$

where s_v is a fresh signal variable and $S\ \{s_v\ x/v\}$ denotes the syntactic substitution of $s_v\ x$ for v in S. Component (5) first computes the value of tm, which is supplied on signal s_v. In the next control step, S picks up this value from s_v by (6). Notice again that the newly introduced predicate E_a and newly introduced signal s_v appear two times, which preserves the consistency in logic and in space.

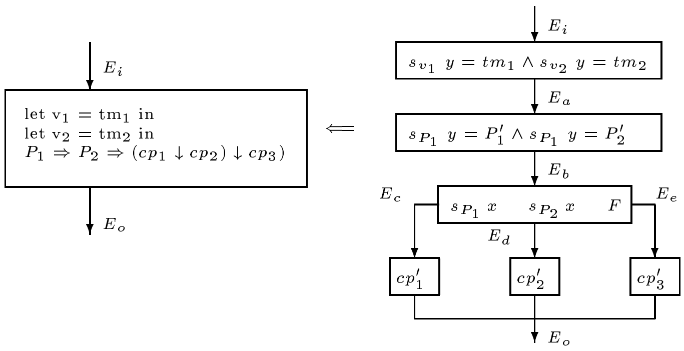
One can generalize the constructions to decompose a nested sequence of let-ins and conditionals such as

$$(\text{let } v_1 = tm_1 \text{ in let } v_2 = tm_2 \text{ in } P_1 \Rightarrow P_2 \Rightarrow (cp_1 \downarrow cp_2) \downarrow cp_3)\ [E_i, E_o]$$

into the models

$$\begin{aligned}
&(s_{v_1}\ y = tm_1 \wedge s_{v_2}\ y = tm_2 \wedge P_1^i \wedge P_2^i \wedge cp_1^i \wedge cp_2^i \wedge cp_3^i)\ [E_i, E_a]\ \wedge \\
&\quad (s_{P_1}\ y = P'_1 \wedge s_{P_2}\ y = P'_2 \wedge {cp'}^i_1 \wedge {cp'}^i_2 \wedge {cp'}^i_3)\ [E_a, E_b]\ \wedge \\
&\quad\quad ({cp'}^i_1 \wedge {cp'}^i_2 \wedge {cp'}^i_3)\ [E_b, (s_{P_1}\ x)((s_{P_2}\ x)(E_c, E_d), E_e)]\ \wedge \\
&\quad\quad\quad {cp'}_1\ [E_c, E_o]\ \wedge\ {cp'}_2\ [E_d, E_o]\ \wedge\ {cp'}_3\ [E_e, E_o],
\end{aligned}$$

where P'_1 abbreviates the substitution $P_1\{s_{v_1}/v_1, s_{v_2}/v_2\}$, and similarly for P'_2, ${cp'}_1, {cp'}_2, {cp'}_3$. The decomposition can be illustrated as seen in the picture below, where for simplicity all signal protections have been omitted:



Note, the nested let_ins and the nested conditionals are each executed for themselves in one control step, whereas the ordering between the let_ins and the conditionals is preserved. They are executed in series.

Let the general construction for this decomposition be called Ψ^b. Then Ψ^b is given by the definition in Fig. 2. A few explanations will be helpful to understand Ψ^b. The input structure is always of the form $comps\ [E_i, E_o]$ where the computer structure $comps$ is one of three possibilities, dealt

4.b.1	$\Psi^b\ ((os = iss)\ [E_i,\ E_o]) \mathrel{\widehat{=}} (os = iss)\ [E_i,\ E_o]$	
4.b.2	$\Psi^b\ ((\text{let } A \text{ in } cs)\ [E_i,\ E_o]) \mathrel{\widehat{=}}$ $(\Psi_{l1}\ (\text{let } A \text{ in } cs))\ [E_i,\ E_m] \ \wedge\ \Psi^b\ ((\Psi_{l2}\ (\text{let } A \text{ in } cs,\ [\,]))\ [E_m,\ E_o])$	
4.b.3	$\Psi^b\ ((P \Rightarrow cs_1 \downarrow cs_2)\ [E_i,\ E_o]) \mathrel{\widehat{=}}$ $(\Psi_{p1}\ (P \Rightarrow cs_1 \downarrow cs_2)\ \wedge\ iprt)\ [E_i, E_m] \wedge$ $iprt\,[E_i,\ \Psi_{p2}\ (P \Rightarrow cs_1 \downarrow cs_2)]\ \wedge\ \Psi_{p3}\ (P \Rightarrow cs_1 \downarrow cs_2,\ E_o)$ where $iprt = (\Psi_{p3}\ (P \Rightarrow cs_1 \downarrow cs_2,\ E_o))^i$.	
4.b.4	$\Psi_{l1}\ (\text{let } v = tm \text{ in } cs) \mathrel{\widehat{=}} s_v\ y = tm \wedge \Psi_{l1}(cs)$	
4.b.5	$\Psi_{l1}\ (os = iss) \mathrel{\widehat{=}} iss^i$	
4.b.6	$\Psi_{l1}\ (P \Rightarrow cs_1 \downarrow cs_2) \mathrel{\widehat{=}} P^i \wedge cs_1^i \wedge cs_2^i$	
4.b.7	$\Psi_{l2}\ (\text{let } v = tm \text{ in } cs,\ L) \mathrel{\widehat{=}} \Psi_{l2}\ (cs,\ [s_v\ x/v] :: L)$	
4.b.8	$\Psi_{l2}\ (os = iss,\ L) \mathrel{\widehat{=}} os = (iss\ L)$	(do substitution L on iss)
4.b.9	$\Psi_{l2}\ (P \Rightarrow cs_1 \downarrow cs_2,\ L) \mathrel{\widehat{=}} (P\ L) \Rightarrow (cs_1\ L) \downarrow (cs_2\ L)$	(do substitution)
4.b.10	$\Psi_{p1}\ (P \Rightarrow cs_1 \downarrow cs_2) \mathrel{\widehat{=}} s_P\ y = P\ \wedge\ \Psi_{p1}\ (cs_1)\ \wedge\ \Psi_{p1}\ (cs_2)$	
4.b.11	$\Psi_{p1}\ (os = iss) \mathrel{\widehat{=}} T$	
4.b.12	$\Psi_{p1}\ (\text{let } A \text{ in } cs) \mathrel{\widehat{=}} T$	
4.b.13	$\Psi_{p2}\ (P \Rightarrow cs_1 \downarrow cs_2) \mathrel{\widehat{=}} (s_P\ x)\ (\Psi_{p2}\ (cs_1),\ \Psi_{p2}\ (cs_2))$	
4.b.14	$\Psi_{p2}\ (os = iss) \mathrel{\widehat{=}} E_{iss}$	
4.b.15	$\Psi_{p2}\ (\text{let } A \text{ in } cs) \mathrel{\widehat{=}} E_{cs}$	
4.b.16	$\Psi_{p3}\ (P \Rightarrow cs_1 \downarrow cs_2,\ E_o) \mathrel{\widehat{=}} \Psi_{p3}\ (cs_1,\ E_o) \wedge \Psi_{p3}\ (cs_2,\ E_o)$	
4.b.17	$\Psi_{p3}\ (os = iss,\ E_o) \mathrel{\widehat{=}} (os = iss)\ [E_{iss},\ E_o]$	
4.b.18	$\Psi_{p3}\ (\text{let } A \text{ in } cs,\ E_o) \mathrel{\widehat{=}} \Psi^b\ ((\text{let } A \text{ in } cs)\ [E_{cs},\ E_o])$	

Figure 2: Construction Scheme Ψ^b

with by Ψ^b in the equations 4.b.1, 4.b.2 and 4.b.3: if *comps* is a computer *comp*, case 4.b.1, then nothing needs to be done. This is the base case of the construction. If it is a let_in structure then 4.b.2 composes the result as the conjunction of two parts determined by the sub-constructions Ψ_{l1} and Ψ_{l2}: first Ψ_{l1} follows a consecutive sequence of nested let_ins and realizes the bindings as a set of parallel transitions, by introducing fresh signals for the bound variables. It also computes the appropriate signal protections; Ψ_{l2} eliminates the let_ins by substitution, the result of which is then continued to be decomposed by Ψ^b. If *comps* is a condition structure then equation 4.b.3 builds on the three sub-constructions Ψ_{p1}, Ψ_{p2}, and Ψ_{p3}. Here Ψ_{p1} follows a consecutive sequence of nested conditions and realizes the evaluation of all predicates in one parallel step. The second conjunct in 4.b.3 is the decision tree in the form of a leaving predicate structure. This is done by Ψ_{p2}. Finally, Ψ_{p3} deals with the decomposition of the continuation processes that are invoked by the decision tree.

The construction is deterministic except for the generation of fresh variables and the choice of their names. In 4.b.4, 4.b.7, 4.b.10, and 4.b.13 new (existential) signal variables s_v, s_P, and in 4.b.2/3, 4.b.14/15, and 4.b.17/18 new (existential) entry and leaving predicate variables, E_m, E_{iss}, E_{cs}, are introduced. The choice of E_m in 4.b.2/3 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.b.4 and 4.b.7, and s_P in 4.b.10 and 4.b.13. The choice of s_v and s_P, however, must be functional, *i.e.* be uniquely determined by variable v and predicate P, respectively. Similarly, the choices of E_{iss} and E_{cs} in 4.b.14/15 and 4.17/18 must be functional, *i.e.* (in some way) uniquely depend on the sub-terms iss and cs, respectively.

Theorem 2 *Assume that the trivial Boolean T is generally omitted in a conjunction. Then,*

$$\Psi^b \;:\; ((tt^+ \cup comp)\,[E, lps])^+ \;\vdash\; comps\,[E, lps],$$

where $\cup$ denotes the ordinary set-theoretic union of sets (of languages.)

We can now combine our two constructions Ψ^a and Ψ^b to obtain a new construction Ψ^c, that first applies $\Psi^a\,[E, lps]$ and then Ψ^b. Formally,

$$\Psi^c = \Psi^b \,\circ\, (\Psi^a\,[E, lps]),$$

such that

$$\Psi^c \;:\; ((tt^+ \cup comp)\,[E, lps])^+ \vdash comp\,[E, lps].$$

Here $\circ$ is the *functional composition* of constructions. In general, if $\Theta_1 : \mathcal{R} \vdash \mathcal{S}$ and $\Theta_2 : \mathcal{S} \vdash \mathcal{T}$, then $(\Theta_1 \circ \Theta_2) : \mathcal{R} \vdash \mathcal{T}$ is defined by $(\Theta_1 \circ \Theta_2)(T) = \Theta_1(\Theta_2(R))$. Here, $T \in \mathcal{T}$ and $R \in \mathcal{R}$.

The effect of Ψ^c as a refinement of the specification $comp\,[E, lps]$ is to reach into $comp$ up to a certain depth and to break out the appropriate control structure (by $\Psi^a\,[E, lps]$.) This control structure is then implemented in terms of term transitions (by Ψ^b.) In the resulting specification, which is an instance of $((tt^+ \cup comp)\,[E, lps])^+$, there may still be decomposable conjuncts of form $comp\,[E, lps]$. On these we could apply Ψ^c again, until there are no more of them left. We then end up in $(tt^+\,[E, lps])^+$. Let Ψ^e be this construction. Ψ^e can also be expressed using general combinators. The first combinator we need is the *union operation*: if $\Theta : \mathcal{S}_1 \vdash \mathcal{S}_2$ and $\mathcal{T}$ a specification scheme, then $\mathcal{T} \cup \Theta : \mathcal{T} \cup \mathcal{S}_1 \vdash \mathcal{T} \cup \mathcal{S}_2$, where $\mathcal{T} \cup \Theta$ is defined in the obvious way as

$$\begin{array}{lcll} (\mathcal{T} \cup \Theta)(X) & \widehat{=} & \Theta(X) & \text{if } X \in \mathcal{S}_2 \\ (\mathcal{T} \cup \Theta)(X) & \widehat{=} & X & \text{if } X \in \mathcal{T}. \end{array}$$

Notice, if $\mathcal{S}_2$ and $\mathcal{T}$ overlap, then we have a choice. This means that $\mathcal{T} \cup \Theta$ may be nondeterministic even if Θ is not. Another combinator that is relevant is the *conjunctive mapping*: if $\Theta : \mathcal{S} \vdash \mathcal{T}$, then $\Theta^+ : \mathcal{S}^+ \vdash \mathcal{T}^+$, where Θ^+ is defined by

$$\begin{array}{rcll} \Theta^+(X) & \widehat{=} & \Theta(X) & \text{if } X \in \mathcal{S} \\ \Theta^+(X \wedge R) & \widehat{=} & \Theta(X) \wedge \Theta^+(R) & \text{otherwise.} \end{array}$$

With these two combinators we can now form the construction $\Psi^d = (tt^+\,[E, lps] \cup \Psi^c)^+$, by which we can apply Ψ^c to every conjunctive element of the form $comp\,[E, lps]$ in an instance of $((tt^+ \cup comp)\,[E, lps])^+$. For Ψ^d we obtain the specification

$$\Psi^d : (tt^+\,[E, lps] \cup ((tt^+ \cup comp)\,[E, lps])^+)^+ \vdash (tt^+\,[E, lps] \cup comp\,[E, lps])^+.$$

It is now easy to verify that for the lhs $(tt^+\,[E, lps] \cup ((tt^+ \cup comp)\,[E, lps])^+)^+ = ((tt^+ \cup comp)\,[E, lps])^+$, and also that for the rhs $(tt^+\,[E, lps] \cup comp\,[E, lps])^+ = ((tt^+ \cup comp)\,[E, lps])^+$. Thus, we get

$$\Psi^d : ((tt^+ \cup comp)\,[E, lps])^+ \vdash ((tt^+ \cup comp)\,[E, lps])^+.$$

All we need to do to get Ψ^e is to iterate Ψ^d until we reach a *fixed-point*, *i.e.* until there is no control structure left to be implemented. Formally, $\Psi^e = \mu\Psi^d$, where

$$\begin{array}{lll} Y \in (\mu\Psi^d)(X) & \textit{iff} & \exists X_0, \ldots, X_n.\; X = X_0 \;\wedge\; X_n = Y \;\wedge \\ & & \qquad (\forall i < n.\; X_{i+1} \in \Psi^d(X_i)) \;\wedge\; (\forall Y' \in \Psi^d(X_n).\; Y' = X_n). \end{array}$$

For arbitrary constructions $\Theta : \mathcal{S} \vdash \mathcal{S}$, $\mu\Theta$ need not be total, *i.e.* for some X we may have $(\mu\Theta)(X) = \emptyset$. However, for Ψ^d it can be shown that $\mu\Psi^d$ is indeed total. Viewing Ψ^d as a rewriting system, totality of $\mu\Psi^d$ corresponds to the condition of weak normalization. One can further show that Ψ^e satisfies the specification

$$\Psi^e : ((tt^+ \cup os = is)\ [E, lps])^+ \vdash ((tt^+ \cup comp)\ [E, lps])^+.$$

To round off our tool set we might come up with a further construction Ψ^f, using the logical equivalence

$$(sg_1\ y = tm_1)\ \wedge\ (sg_2\ y = tm_2) \quad \equiv \quad (sg_1\ y,\ sg_2\ y) = (tm_1,\ tm_2),$$

that implements the decomposition

$$\Psi^f \quad : \quad tt^+ \vdash os = is.$$

Now finally we can assemble our desired refinement Ψ from the machine instruction level to the term transition level as follows:

$$\begin{aligned}
\Psi &= ((tt^+ \cup \Psi^f)\ [E, lps])^+ \ \circ\ \Psi^e \\
&= ((tt^+ \cup \Psi^f)\ [E, lps])^+ \ \circ\ (\mu\Psi^d) \\
&= ((tt^+ \cup \Psi^f)\ [E, lps])^+ \ \circ\ (\mu(tt^+\ [E, lps] \cup \Psi^c)^+) \\
&= ((tt^+ \cup \Psi^f)\ [E, lps])^+ \ \circ\ \mu(tt^+\ [E, lps] \cup (\Psi^b\ \circ\ (\Psi^a\ [E, lps])))^+.
\end{aligned}$$

Here Ψ^a, Ψ^b, and Ψ^f are given directly, though they could be composed systematically from smaller pieces as well. For Ψ we derive the specification

$$\Psi : (tt^+\ [E, lps])^+ \vdash ((tt^+ \cup comp)\ [E, lps])^+,$$

or by restriction, if we observe that $comp\,[E, lps]$ is a sub-specification (subset of languages) of $((tt^+ \cup comp)\ [E, lps])^+$:

$$\Psi : (tt^+\ [E, lps])^+ \vdash comp\,[E, lps]$$

as desired.

We should point out that the constructions presented above are just examples. Many more constructions are necessary to refine a computer from machine instruction level to, say, the register transfer level. Some other examples are mentioned later in Sec. 6. The gist of this section is to show that constructions can be treated in a way similar to functions in a functional language; and that specification schemes can be used to specify constructions rigorously, similar to the way types specify functions. However, since BNF specification schemes have a richer structure than ordinary functional types the expressibility is considerably higher.
In this context it is useful to contrast our constructions with the tactics in a typical theorem prover: The only specification usually available for a tactic is that it has the type *theorem* $\rightarrow$ *theorem*, where the function arrow $\rightarrow$ means "partial function". This is not extremely informative and the reason for the correctness leak mentioned in the introduction.

5 Constructing Logic, Space and Time Relations

Logic, space and time are the three primitive dimensions of hardware. Constructing hardware means constructing these three dimensions at different levels of abstraction. For hardware components the relation in logic is fundamental and can be reduced to the three basic operations:

sequence, condition, and loop. The relation in space can be reduced to the three basic components: port-connection, bus, and register. The relation in time is accompanied by the relation in logic: through entry predicates we measure and define time points, time units, time clocks, time layers, and time refinements. We give a short summary, further details can be found in [39].

5.1 Constructing Logic Relation

Logic relation is constructed by introducing fresh entry predicate variables and dividing a specification into, for example, two sub-specifications. To preserve logic consistency, the variable always is made to appear in one sub-specification as the leaving and in the other one as the entry predicate. This leads to the logic structure of construction as follows.

1. Sequence: Through E_m compose $\alpha\,[E_i,\, E_m]$ and $\beta\,[E_m,\, E_o]$
2. Condition: Through leaving predicate structure, $\alpha\,[E_a,\ \beta\,(E_b,\, E_c)]$
3. Loop: Through a sequence of entry/leaving predicates with common start predicate E, $\alpha\,[E,\, E_a] \wedge \cdots \wedge \gamma\,[E_z,\ E]$.

Thus, our extension of the traditional verification model by the entry predicate and leaving predicate structure, captures an elementary form of sequencing control. It allows us to separate control from function, and consequently to refine both aspects independently wherever appropriate. The notion of control here is still fairly abstract, however. It depends on the specific instantiation of the entry and leaving predicates; it can stand for synchronous or asynchronous control, single-clocked or multi-phase, pipelined or un-pipelined. This genericity justifies using the more general term 'logic relation' rather than 'control'.

5.2 Constructing Space Relation

In a very general sense, the design process is stimulated by the aim to establish the consistency of a specification. This also applies to the construction in space. On the other hand, under the condition ensuring consistency there are various possible selections for the construction in space. We present a generic way based on the *connection model* to guarantee the consistency, while providing maximal flexibility to construct varied possible space relations.
When a space relationship is needed we introduce a pair of ports: one, say r, as the output of a component and the other one, say s, as the input of another component. In this case we introduce a new *connection* formed as

$$(s\,y = r\,x)\ [E_i,\, E_o],$$

where s is the output and r the input of the connection. Like in introducing entry predicates, we always introduce *fresh* ports (signals). However, when allocating hardware we possibly assign two or more functions, *e.g.* Add and Sub, to be implemented by the same device. This leads to the unification of the corresponding ports. When allocation is done we distinguish three cases for a connection with output s:

1. Port-Connection: Signal s only appears once. In this case we can unify s with r and E_i with E_o and use the weakest implementation, the Boolean constant T:

$$(s\,y = s\,x)\ [E_i,\, E_i]\ \equiv\ T.$$

2. Bus: s is the common output of two or more connections.

$$(s\,y = r_1\,x)\ [E_{i1},\, E_{o1}]\ \wedge\ \cdots\ \wedge\ (s\,y = r_k\,x)\ [E_{ik},\, E_{ok}] \quad (k \geq 2)$$

In this case a bus is introduced to realize signal s.

3. Register: If a time constraint is added, like in

$(s\,y = r\,x \wedge y > x)\ [E_i,\, E_o]$

we can realize the connection by a register with input r and output s. Depending on the time level, a register may also be a simple delay.

5.3 Constructing Time Relation

We cannot directly catch time in hardware, it always is measured through its effect. In the construction model it is the entry predicates that measure time. When we construct the entry predicates we also construct time. The construction of time is always accompanied by the construction of entry predicates.

Definition 5.3.1 [time point]: A *time point* for entry predicate E is a t such that $E\ t$ is true.□

Definition 5.3.2 [time clock & time layer]: If E_i, $(i = 1, \ldots, k)$, are entry predicates of a circuit specification at a given level of description, then the disjunction $\vee_{k=1}^{n} E_k$ is called a(n) (abstract) *clock* of the circuit. The clock *ticks* if and when one and only one predicate in the disjunction becomes true. We call the set of all entry predicates of a circuit its *time layer.* Time clock and layer are relative to the specification level. The same circuit may have different time clocks and time layers at different specification levels. □

Definition 5.3.3 [time refinement]: The construction from a high-level specification, whose time layer is L_i, to the adjacent low-level specification, whose time layer is L_{i+1}, is through introducing a set Δ of new middle entry predicates. Therefore the time construction is $L_i \cup \Delta = L_{i+1}$. The *time refinement* of a circuit from top specification 1 to bottom implementation n is given as the sequence $L_1 \subset L_2 \subset \cdots \subset L_n$ of time layers. As the Δ for each step is non-empty, this is a proper hierarchy. □

Definition 5.3.4 [time unit]: We can introduce a specific time unit into a model by refining $S\,(x,\, y)\ [E_i,\, E_o]$ into $\forall x.\ E_i\, x \Rightarrow S\,(x, x+\delta) \wedge E_o\,(x+\delta)$. The δ is called the *time unit* of the model. □

In our methodology the process of introducing a time clock is a process of imposing more and more external constraints. So, we can have a various specific selections for the clocks of a circuit:

1. Non-Clock: Do not introduce clocks at all, use other forms of control.
2. Variable-Clock: Introduce clocks and time units.
3. Constant-Clock: Introduce clocks and time units, and moreover the constraint that the time units of all models are identical.

6 Case Study

In this section we describe briefly the main steps in the construction of a microprogrammed computer, Gordon's computer [13], which was taken as a case study in [39]. It includes four main levels:

1. Machine Instruction (MIL): The programmer's view of a computer as a global next-state transition. The specification scheme capturing MIL was introduced in Sec. 2. This level is characterized by complex data (next-state function) but simple control (single-transition loop) structure.

2. State Transition (STL): At this level the control structure has been separated from the data part, but not as yet implemented. Each instruction is decomposed into a set of concurrent state transitions. There is still one global control loop.

3. Term Transition (TTL): This is the mixed behaviour-structural specification at term transition level. The control structure is realized by entry/leaving predicates as an abstract microprogram, while the data functions appear as synchronized term transitions.

In contrast to the machine instruction level, the term transition level is characterized by a complex control structure of synchronized local state transformations, each of which is governed by a simple function term.

4. Register Transfer (RTL): A model in the register transfer specification is either a function, which is considered implementable at the register transfer level, a port-connection, bus, or register. The control has been flattened into a linear microprogram, microinstruction and microcode.

All description levels MIL, STL, TTL, RTL have been formalized rigorously as specification schemes [39, 40]. Let us illustrate the two main steps, *viz.* the construction from state transition level STL to term transition level TTL, and from TTL to register transfer level RTL, through rather simplified examples. The complete construction of Gordon's computer, which applies our framework of specification and construction schemes, has been formalized in detail in [39].

Example 6.1. [from STL to TTL]: Our top-level specification describes a simple generic microcomputer:

Specification at STL:

$$
\begin{array}{l}
(\ \text{let } v_1 = tm_1 \text{ in} \\
\quad \text{let } v_2 = tm_2 \text{ in} \\
\qquad (P \Rightarrow (\alpha \wedge \Delta) \downarrow (Q \Rightarrow (\beta \wedge \Delta) \downarrow \gamma)) \\
)\ [E,\ E]
\end{array}
$$

This computer has three machine instructions described through state transitions $\alpha \wedge \Delta$, $\beta \wedge \Delta$ and γ respectively. Δ is the common block in the state transition of the first two instructions. For instance, Δ might contain the incrementation of the program counter, $pc\,y = pc\,x + 1$. The three machine instructions are executed under different condition combinations led by P_1 and P_2. Two let_in sentences describe the common parts tm_1 and tm_2 bound to variables v_1 and v_2, respectively. For instance, tm_1 might stand for the opcode of the current instruction and tm_2 for the operands. The entry and leaving predicate of the specification are the same, which describes an infinite loop for the repeated execution of the machine cycle. By applying refinement constructions similar to those in Sec. 4, we derive the following implementation at term transition level.

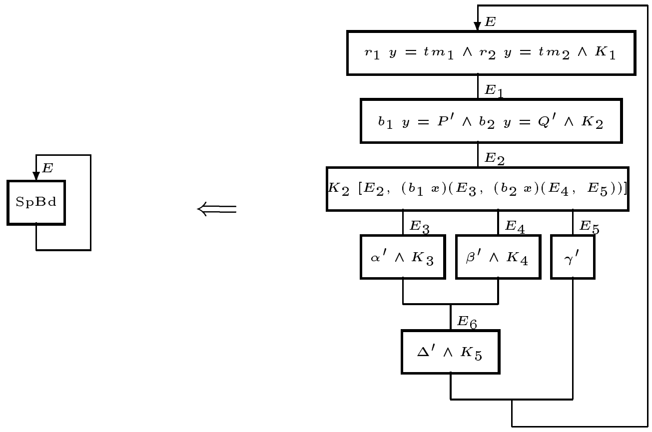
Implementation at TTL:

$$
\begin{array}{l}
(r_1\ y = tm_1 \wedge r_2\ y = tm_2 \wedge K_1)\ [E,\ E_1]\ \wedge\ (b_1\ y = P' \wedge b_2\ y = Q' \wedge K_2)\ [E_1,\ E_2] \wedge \\
K_2\ [E_2,\ (b_1\, x)(E_3,\ (b_2\, x)(E_4,\ E_5))]\ \wedge\ (\alpha' \wedge K_3)\ [E_3,\ E_6]\ \wedge\ (\beta' \wedge K_4)\ [E_4,\ E_6] \wedge \\
(\Delta' \wedge K_5)\ [E_6,\ E]\ \wedge\ \gamma'\ [E_5,\ E],
\end{array}
$$

where

$$
\begin{array}{ll}
K_1 = P^i \wedge Q^i \wedge \alpha^i \wedge \beta^i \wedge \gamma^i \wedge \Delta^i & K_3 = K_4 = \Delta'^i \\
K_2 = \alpha'^i \wedge \beta'^i \wedge \gamma'^i \wedge \Delta'^i & K_5 = \alpha'^o \wedge \beta'^o
\end{array}
$$

are the signal protections arising from decomposing atomic transitions. We write α' for $\alpha[r_1\, x/v_1,\ r_2\, x/v_2]$, and similarly for P', Q', β', γ', and Δ'. The refinement can be depicted graphically as follows, where "SpBd" abbreviates the specification body of the initial STL specification:



The logical structure of the computer at term transition level gives us a clear picture of the computer's global structure at a high level — each box corresponds to a set of parallel term transitions which are uninterpreted and abstract. 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. We call the implementation the *abstract microprogram*. 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 a synchronous or asynchronous circuit.

Example 6.2. [from TTL to RTL]: We now consider a part of the computer at TTL, for example the γ' in the previous example 6.1.

Specification at TTL:

$$(s_1\ y = f\ (g\ (r_1\ x))\ \wedge\ s_2\ y = h\ (r_2\ x))\ [E_i,\ E_o].$$

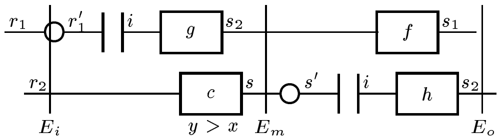
We select a device D_f for function f and D_{gh} for both functions g and h. Assuming that r_1 and r_2 are distinct signals we need to introduce a bus or a multiplexer to feed D_{gf}. Let us say we use the bus, together with tristate gates to drive it. The devices are then defined as

$\mathrm{D}_{gh}\ (\mathrm{c}_g,\ \mathrm{c}_h,\ \mathrm{i},\ \mathrm{s}_2)\ \equiv\ \forall t.\ \mathrm{s}_2\ (t+1) = (\mathrm{c}_g\ t \Rightarrow g\ (\mathrm{i}\ t) \downarrow (\mathrm{c}_h\ t \Rightarrow h\ (\mathrm{i}\ t) \downarrow \mathrm{i}\ t))$
$\mathrm{D}_f\ (\mathrm{c}_f,\ \mathrm{s}_2,\ \mathrm{s}_1)\ \equiv\ \forall t.\ \mathrm{s}_1\ (t+1) = (\mathrm{c}_f\ t \Rightarrow f\ (\mathrm{s}_2\ t) \downarrow \mathrm{s}_2\ t))$
$\mathrm{Reg}\ (\mathrm{c}_r,\ \mathrm{r}_2,\ \mathrm{s})\ \equiv\ \forall t.\ \mathrm{s}\ (t+1) = (\mathrm{c}_r\ t \Rightarrow \mathrm{r}_2\ t \downarrow \mathrm{s}\ t)$
$\mathrm{Gate}\ (\mathrm{c}_{b1},\ \mathrm{r}_1,\ \mathrm{r}_1')\ \equiv\ \forall t.\ \mathrm{r}_1'\ t = (\mathrm{c}_{b1}\ t \Rightarrow \mathrm{Mk\_Tri}\ (\mathrm{r}_1\ t) \downarrow \mathrm{Float})$
$\mathrm{Gate}\ (\mathrm{c}_{b2},\ \mathrm{s},\ \mathrm{s}')\ \equiv\ \forall t.\ \mathrm{s}'\ t = (\mathrm{c}_{b2}\ t \Rightarrow \mathrm{Mk\_Tri}\ (\mathrm{s}\ t) \downarrow \mathrm{Float})$
$\mathrm{Bus}\ (\mathrm{r}_1',\ \mathrm{s}',\ \mathrm{i})\ \equiv\ \forall t.\ \mathrm{i}\ t = \mathrm{Dest\_Tri}\ (\mathrm{r}_1'\ t \cup \mathrm{s}'\ t).$

We could also work with more abstract versions of these devices, *viz.* simple instances of our scheme *comp* $[E, E]$. For example, instead of D_f as shown here we could use the model $(s_1\, y = (\mathrm{c}_f\ x \Rightarrow f\ (\mathrm{s}_2\ x) \downarrow \mathrm{s}_2\ x))\ [E_i, E_o]$. To reflect the allocation we refine our TTL specification to

$$(r_1'\ x = r_1\ x \wedge i\ x = r_1'\ x \wedge s_2\ y = g\ (i\ x)\ \wedge\ s\ y = r_2\ x \wedge y > x)\ [E_i,\ E_m] \wedge$$
$$(s_1\ y = f\ (s_2\ x) \wedge s'\ x = s\ x\ \wedge\ i\ x = s'\ x \wedge s_2\ y = h\ (i\ x))\ [E_m,\ E_o].$$

Using appropriate symbols to distinguish the individual sub-state transitions, we get the following abstract control flow:



The control is effected by the three abstract control steps E_i, E_m, E_o. This can be viewed as a linear microprogram, which can be turned into a concrete microprogram, *viz.* through instantiating the entry predicates by the state of a microinstruction counter: $E_i\ x\ \equiv$ addr x = 11, $E_m\ x\ \equiv$ addr x = 12, and $E_o\ x\ \equiv$ addr x = 13. We then obtain a concrete RTL structure.

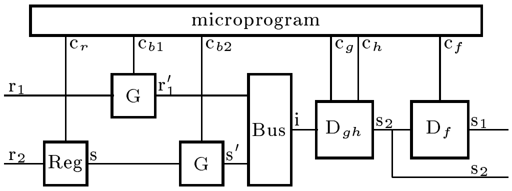
Implementation at RTL:

Microprogram:

$(\forall x.\ \mathrm{addr}\ x = 11 \Rightarrow\ \mathrm{c}_{b1}\ x \wedge \mathrm{c}_g\ x \wedge \mathrm{c}_r\ x\ \wedge\ \mathrm{addr}\ (x+1) = 12) \wedge$
$(\forall x.\ \mathrm{addr}\ x = 12 \Rightarrow\ \mathrm{c}_{b2}\ x \wedge \mathrm{c}_h\ x \wedge \mathrm{c}_f\ x\ \wedge\ \mathrm{addr}\ (x+1) = 13)$

Datapath:

$\mathrm{Reg}\ (\mathrm{c}_r,\ \mathrm{r}_2,\ \mathrm{s}) \wedge \mathrm{Gate}\ (\mathrm{c}_{b1},\ \mathrm{r}_1,\ \mathrm{r}_1')\ \wedge\ \mathrm{Gate}\ (\mathrm{c}_{b2},\ \mathrm{s},\ \mathrm{s}') \wedge \mathrm{Bus}\ (\mathrm{r}_1',\ \mathrm{s}',\ \mathrm{i}) \wedge$
$\mathrm{D}_{gh}\ (\mathrm{c}_g,\ \mathrm{c}_h,\ \mathrm{i},\ \mathrm{s}_2) \wedge \mathrm{D}_f\ (\mathrm{c}_f,\ \mathrm{s}_2,\ \mathrm{s}_1)$



In [39] a complete sequence of generic refinement steps has been worked out, which from the microinstruction level leads all the way down to a concrete implementation at register-transfer level. The refinements involve 10 major levels of specification schemes together with 9 major construction steps. All of these construction steps are rigorously defined and formally verified, using the framework of BNF specification schemes and constructions schemes described in this paper. The framework has been applied to construct several different versions of Gordon's computer.

7 Conclusion and Further Work

The work reported in this paper is concerned with the formal synthesis of hardware, with the particular goal of achieving the rigour of formal verification while considerably increasing the amount of genericity and the level of formalization. The proposed framework is centered around the notion of a specification scheme, construction scheme, and construction model.

We argue that the traditional logic-centered approach to formal synthesis (based on theorem-proving) in terms of formulas and rules employs too low a level of formalization. Instead of arbitrary formula schemes our calculus constructs specification schemes, each of which captures the generic structure of a class of hardware objects at a specific level of abstraction. This increases the level of formalization but also the amount of genericity: where formula schemes only exhibit vertical genericity a specification scheme also exploits the horizontal genericity of inductive BNF syntax. Construction schemes, then, generalize ordinary rule schemes in a natural way. They are logically correct refinements of one specification scheme to another one.

Yet another source of genericity in our approach lies in the use of the construction model. Besides the obvious signal parameters of the traditional verification model (components-as-relations paradigm) the construction model allows components to communicate via additional entry and leaving predicates. Through this extra dimension of logical communication we can separate functional from control aspects, and refine both independently whenever appropriate. In particular, we can concentrate on functional decomposition, even perform steps of abstract scheduling and allocation, without fixing the particular mode of control. We may instantiate to synchronous or asynchronous, micro-programmed or hard-wired control, at a fairly late stage in the design. We believe that the construction model is a natural and useful generalization of the traditional verification model.

The rather informal character of the presentation in this paper might be misleading. It should be stressed that the proposed construction framework is a rigorous calculus based on formally specified and verified construction schemes. How this can be done is demonstrated in [39], where a suite of concrete specification and construction schemes is presented to derive a class of computers. In this paper we concentrated essentially on motivating the basic ideas from a more general perspective.

Further work will be in two directions. We plan to develop a semi-automatic computer design system which can design a register-transfer level microprocessor from a specification at machine instruction level. It will be based on the ideas discussed in this paper. The user interaction consists in the selections for high-level allocation and scheduling, for which standard algorithms will be integrated. Exploiting the genericity of our approach the tool will be extendible to realize various alternative forms of control, such as multi-phase synchronous, pipelined, or asynchronous handshake. This is a long term goal, however. Other work aims at the theoretical foundations of our framework, in particular concerning the algebraic and logical properties of the construction model.

References

[1] 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.

[2] 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.

[3] G. Birtwistle and B. Graham, Verifying the SECD in HOL. In [32].

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

[5] B. Bose and S. D. Johnson, DDD-FM9001: Derivation of a Verified Microprocessor. In [29].

[6] H. Busch, Proof-based transformation of formal hardware models. In [22], pp. 271–296.

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

[8] L. Claesen, editor, IMEC-IFIP International Workshop on *Applied Formal Methods for Correct VLSI Design*, Volume 1+2, Elsevier/North-Holland, 1989.

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

[10] 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.

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

[12] M. P. Fourman, Formal System Design. In [32], pp 191-236.

[13] 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.

[14] 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*, North-Holland, 1986, pp. 153-177.

[15] M. J. C. Gordon and T. F. Melham, *Introduction to HOL*. Cambridge University Press, 1993.

[16] 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.

[17] F.K. Hanna, M. Longley, and N. Daeche, Formal synthesis of digital systems. In [8], pages 532-548.

[18] F.K. Hanna and N.Daeche, Strongly-Typed Theory of Structure and Behaviors. In [29], pp. 39-54.

[19] N.A. Harman and J. V. Tucker, Algebraic Models and the Correctness of Microprocessors. In [29], pp. 92-108.

[20] J. M. J. Herbert, Incremental Design and Formal Verification of Microcoded Microprocessors. In [34], pp. 157–174.

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

[22] G. Jones and M. Sheeran, *Designing Correct Circuits*. Springer, 1991.

[23] G. Jones and M. Sheeran, Circuit Design in RUBY. In [32].

[24] J. J. Joyce, *Multi-Level Verification of Microprocessor-Based Systems*, Ph.D. Thesis, Computer Laboratory, Cambridge University, December 1989 (Technical Report No. 195, May 1990).

[25] J. J. Joyce, Generic Specification of Digital Hardware. In [22], pp. 68–91.

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

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

[28] 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.

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

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

[31] M. Srivas and M. Bickford, *Formal Verification of a Pipelined Microprocessor*, In IEEE Software, September 1990, pp. 52–64.

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

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

[34] 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.

[35] 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.

[36] S. Tahar and R. Kumar, Towards a Methodology for the Formal Verification of RISC Processors. In Proceedings IEEE International Conference on Computer Design (ICCD'93), 1993, pp. 58–62.

[37] D. Verkest and L. Claesen and H. De Man, On the use of the Boyer-Moore theorem prover for correctness proofs of parametrized hardware modules. In [8], pp.405–422.

[38] Li-Guo Wang, Synthesis of Nondeterministic Logic Programs, Chinese Journal of Software, Vo. 1, No. 1, ISSN 1000-9825, CN 11 - 2560, January 1990.

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

[40] Li-Guo Wang and M. Mendler, Formal Derivation of A Class of Computers: Its high stage — abstract microprogramming. To appear in H. Eveking and P. Camurati, eds., *Correct Hardware Design and Verification Methods* (CHARME'95), Springer, 1995.

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

[42] P. J. Windley, A Theory of Generic Interpreters. In [29], pp. 122–134.