, , 1–24 ()

Modeling a Hardware Synthesis Methodology in Isabelle

DAVID BASIN AND STEFAN FRIEDRICH {basin,friedric}@informatik.uni-freiburg.de
Institut für Informatik
Albert-Ludwigs-Universität Freiburg

Editor:

Abstract. *Formal Synthesis* is a methodology developed at the University of Kent for combining circuit design and verification, where a circuit is constructed from a proof that it meets a given formal specification. We have reinterpreted this methodology in ISABELLE's theory of higher-order logic so that circuits are incrementally built during proofs using higher-order resolution. Our interpretation simplifies and extends Formal Synthesis both conceptually and in implementation. It also supports integration of this development style with other proof-based synthesis methodologies and leads to techniques for developing new classes of circuits, e.g., recursive descriptions of parametric designs.

Keywords: Hardware verification and synthesis, theorem proving, higher-order logic, higher-order unification.

1. Introduction

Verification by formal proof is time intensive and this is a burden in bringing formal methods into software and hardware design. One approach to reducing the verification burden is to combine development and verification by using a calculus where development steps either guarantee correctness or, since proof steps are in parallel to design steps, allow early detection of design errors. Such approaches have been actively investigated for software development, e.g., the use of constructive logics [4, 9] and the many approaches to deductive synthesis [5, 6, 20, 22]. Of particular relevance to our work here are the approaches to synthesis where resolution is used to construct programs during proofs. This methodology goes back to Green's use of first-order resolution to automatically synthesize programs during proofs [15]. More recently, within ISABELLE, interactive and tactic guided proof by higher-order resolution has been used to construct programs in type theory [23], develop functional programs using classical logics [1, 8], and synthesize logic programs [2]. There is also a growing body of work on proof based hardware synthesis [21, 24, 25, 26] including synthesis based on higher-order resolution using the LAMBDA system [10, 12, 13].

We present work here on combining two different development methodologies: synthesis based on higher-order resolution and a particular methodology for the top-down deductive refinement of specifications into circuits. The latter is a novel approach to hardware synthesis, called *Formal Synthesis*, proposed by the VERITAS group at Kent [18]. In Formal Synthesis, one starts with a *design goal*, which specifies the behavioral properties of a circuit to be constructed, and interactively refines the design using a small but powerful set of *techniques*, which allows the developer to hierarchically decompose specifications and introduce subdesigns and library components in a natural way. Internally, each technique consists of a pair of functions: a *subgoaling function* and a *validation function*. The former decomposes a specification $Spec$ into subspecifications $Spec_i$. The latter takes proofs that some Imp_i achieve $Spec_i$, and constructs an implementation Imp with a proof that Imp achieves $Spec$. When the refinement is finished, the system composes the validation functions that were applied and this constructs a theorem that a (synthesized) circuit satisfies the original design goal.

To make the above clearer, consider one of the simpler techniques, called *Split*. The subgoaling function reduces a design goal $Spec_1 \wedge Spec_2$ to two new design goals, one for each conjunct. The validation function is based on the rule

$$\frac{\vdash Imp_1 \rightarrow Spec_1 \quad \vdash Imp_2 \rightarrow Spec_2}{\vdash Imp_1 \wedge Imp_2 \rightarrow Spec_1 \wedge Spec_2}, \tag{1}$$

which explains how to combine implementations that achieve the subgoals into an implementation that achieves the original goal. This kind of top-down problem decomposition is in the spirit of (LCF style) tactics and designing a good set of techniques is analogous to designing an appropriate set of tactics for a problem domain. However, tactics decompose a goal into subgoals from which a validation proves the original goal, whereas with techniques, the validation proves a different theorem altogether. Formal Synthesis separates design goals and techniques from ordinary theorem-proving goals and tactics. This is a conceptual separation: they are different sorts of entities with different semantics. Moreover they are treated differently in the VERITAS implementation too and Formal Synthesis required extending the VERITAS theorem prover itself.

Our contribution is to show how to reinterpret Formal Synthesis as deductive synthesis based on higher-order resolution and to explain why this interpretation is not only natural but also advantageous. The reinterpretation starts with the final theorem that the Formal Synthesis validations should deliver: a circuit achieves a stated specification. However the circuit is not given up front; instead it is named by a metavariable. Proof proceeds by applying rules in ISABELLE that correspond to VERITAS validation rules like (1). Because rule application uses resolution, the metavariable is incrementally instantiated to the synthesized circuit. Let us illustrate this with the Split technique. Our goal (simplifying slightly) is to prove a theorem like $?Imp \rightarrow Spec$, where the question mark means that Imp is a metavariable. If $Spec$ is a conjunction, $Spec_1 \wedge Spec_2$, then we can resolve $?Imp \rightarrow Spec_1 \wedge Spec_2$ with the rule in (1). The result is a substitution, $?Imp = ?Imp_1 \wedge ?Imp_2$, and the subgoals $?Imp_1 \rightarrow Spec_1$ and $?Imp_2 \rightarrow Spec_2$. Further refinement of these subgoals will generate instances of the $?Imp_i$ and hence $?Imp$.

There are a number of advantages to this reinterpretation. It is conceptually simple. We do away with the subgoal and validation functions and let resolution construct circuits. It is simple to implement. No changes are required to the Isabelle system; instead, we derive rules and program tactics in an appropriate extension of Isabelle's theory of higher-order logic (HOL). Moreover, the reinterpretation makes it easy to implement new techniques that are compatible with the Formal

Synthesis style of development. In the past, we have worked on rules and tactics for the synthesis of logic programs [2] and these programs are similar to circuits: both can be described as conjunctive combinations of primitive relations where existential quantification is used to 'pass values'. By moving Formal Synthesis into a similar theorem proving setting, we could adapt and apply many of our tactics and rules for logic programming synthesis to circuit synthesis. For example, rules for developing recursive programs yield new techniques for developing parameterized classes of circuits. We also exhibit (cf. Section 6) that the kind of rules used for resolution-based development in the LAMBDA system are compatible with our reinterpretation.

The remainder of this paper is organized as follows. In Section 2 we review some essentials concerning VERITAS and ISABELLE. In Section 3 we present the Formal Synthesis techniques and in Section 4 our abstraction, interpretation, and extension of them. We provide two examples in Section 5: a comparator cell, which was developed in VERITAS, and a parameterized carry lookahead adder, which was developed using our extension with induction techniques. Finally we compare with other design methodologies and draw conclusions.

2. Background

We assume some familiarity with ISABELLE [23] and higher-order logic and review only notation and a few essentials. ISABELLE is an interactive tactic-based theorem prover. Its metalogic is a fragment of intuitionistic higher-order logic based on the typed lambda-calculus which allows polymorphism but not dependent types. Object logics are encoded in ISABELLE's metalogic by declaring a *theory*, which is a signature and set of axioms. For example, propositional logic might have a type $Form$ of formulae, with constructors like conjunction, $\wedge$, and a typical proof rule would be

$$!!\, A, B : Form.\, A \Longrightarrow (B \Longrightarrow A \wedge B)\,,$$

which formalizes the conjunction-introduction rule of natural deduction. Here $\Longrightarrow$ and !! are implication and universal quantification in ISABELLE's metalogic, and should not be confused with encoded operators, like $\wedge$, $\rightarrow$, or $\forall$, of the formalized object logic. Outermost (meta-)quantifiers are often omitted and iterated implication is written in a more readable list notation, e.g., $[\![\ A; B\ ]\!] \Longrightarrow A \wedge B$. These implicitly quantified variables are treated as metavariables, which can be instantiated when applying the rule using higher-order unification. In our work we use ISABELLE's theory of higher-order logic, which is based on Gordon's HOL [14], extended with theories of sets, well-founded recursion, natural numbers, and the like.

Within ISABELLE the proof state is represented by a theorem $[\![\ \psi_1, \ldots, \psi_m\ ]\!] \Longrightarrow \psi$ of the meta-theory. Proof construction is done by higher-order resolution: Given a subgoal ψ_i and a proof rule $[\![\ \phi_1; \ldots; \phi_n\ ]\!] \Longrightarrow \phi$, we unify ϕ with ψ_i. If this succeeds, then unification yields a substitution σ and the proof state is updated by applying σ to it and replacing ψ_i with the subgoals $\sigma(\phi_1), \ldots, \sigma(\phi_n)$; since unification is used to apply rules, the proof state itself may contain metavariables. We exploit this to synthesize circuits during proofs. Note that a proof rule can be read as an intuitionistic sequent where the ϕ_i are the hypotheses. Isabelle's resolution tactics apply rules in a way that maintains this illusion of working with sequents and we will often refer to the ϕ_i as assumptions.

VERITAS [17, 16] is a tactic based theorem prover similar to ISABELLE and the HOL system, but its higher-order logic is augmented with constructions from type theory, e.g., standard type constructors such as (dependent) function space, product, and subtype.

In VERITAS, every term and every theorem is associated with a support signature that describes the theory to which the term or theorem belongs. The signature consists of constant definitions, declarations (typing-constraints) etc. and can be extended dynamically with new definitions declarations or even assumptions during proof, where it serves as a proof-context.

When used to reason about hardware, one proves theorems that relate implementations of circuits, *Imp*, to their specifications, *Spec*, e.g.,

$$\forall p_1 \ldots p_n.\ \mathit{Imp}(p_1, \ldots, p_n) \to \mathit{Spec}(p_1, \ldots, p_n)\,,$$

where $p_1 \ldots p_n$ are the external port names. As is common, hardware is represented relationally, where primitive constructors (e.g., transistors, gates, etc.) are relations combined with conjunction and 'wired together' using existential quantification [7]. The variables in $p_1 \ldots p_n$ are typed, and the primary difference between VERITAS and similar systems is that one can use richer types in such specifications.

For example, if we are defining circuits operating over 8-bit words in VERITAS, we would formalize this requirement on word-length in the types. A subtype of natural numbers could be defined by $N \equiv \lambda\, n{:}nat.\, \{m{:}nat \mid m < n\}$ and a type of vectors of length l by $V \equiv \lambda\, t{:}\mathit{Type}.\, \lambda\, l{:}nat.\, (N\, l \to t)$. ($\mathit{Type}$ is a universe of 'small' types.) The elements of the type $(V\ bit\ 8)$ are then vectors of length 8 whose elements are of type bit, defined by $N\,2$. Note that type-checking in VERITAS is not decidable.

3. Formal Synthesis

The Formal Synthesis design methodology consists of starting with a specification and building an implementation that satisfies it. To do this, the user interactively applies techniques (which are also called methods) whose subgoaling functions decompose the specification into hopefully simpler specifications, each of which must be implemented to implement the original specification. Refinement by method application may also produce auxiliary proof obligations, verification conditions that must be proven by the user. A successful proof concludes when all specifications can be reduced to specifications that are implemented by library components (or, as we will see, subdesigns) and all verification conditions are discharged. At this point, the validations associated with the applied methods are executed to build a proof that a circuit achieves a specification.

Before giving the techniques we introduce some notation used by Formal Synthesis. A design specification *spec*, which is to be implemented, is written $\Box spec$ ($\Box$ is just a symbol; there is no relationship to modal logics). A formula *Thm* to be proved is written as $\vdash Thm$. A term Tm to be demonstrated well-typed is written as $\diamond Tm$. A proof in VERITAS is associated with a support signature, which contains the theory used (e.g., datatype definitions), definitions of predicates, and the like. In VERITAS, the signature can be extended dynamically during proof. If a technique extends the signature of a goal $Spec$, then this is written as $[\![extension]\!]Spec$. Examples are the Reveal and the Subdesign technique (see below); the former adds typing constraints of internal signals to the signature and the latter extends the signature with the definition of a new design. Finally, the initial design goal must be a definition of the form $Circ \mathrel{\widehat{=}} \lambda dec.\ Spec$; $Circ$ is the name assigned to the circuit and the right hand side is the specification to be implemented and is given as an abstraction over the input and output signals, which are declared in dec.

Table 1. VERITAS design techniques

Name	Subgoaling rule	Validation rule
Claim	$\dfrac{\Box Spec}{\Box Spec' \quad \vdash Spec' \to Spec}$	$\dfrac{\vdash Imp \to Spec' \quad \vdash Spec' \to Spec}{\vdash Imp \to Spec}$
Split	$\dfrac{\Box(Spec_1 \wedge Spec_2)}{\Box Spec_1 \quad \Box Spec_2}$	$\dfrac{\vdash Imp_1 \to Spec_1 \quad \vdash Imp_2 \to Spec_2}{\vdash Imp_1 \wedge Imp_2 \to Spec_1 \wedge Spec_2}$
Reveal	$\dfrac{\Box \exists dec.\ Spec}{[\![dec]\!] \Box Spec}$	$\dfrac{[\![dec]\!] \vdash Imp \to Spec}{\vdash (\exists dec.\ Imp) \to (\exists dec.\ Spec)}$
Inside	$\dfrac{\Box\, Circ \mathrel{\widehat{=}} \lambda dec.\ Spec}{[\![dec]\!] \Box Spec}$	$\dfrac{[\![dec]\!] \vdash Imp \to Spec}{[\![Circ \mathrel{\widehat{=}} \lambda dec.\ Spec]\!] \vdash \forall dec.\ Imp \to Circ\ ports}$
Library	$\dfrac{\Box library\_part\ args}{}$	$\dfrac{}{\vdash library\_part\ args \to library\_part\ args}$
Subdesign	$\dfrac{\Box Spec}{[\![c \mathrel{\widehat{=}} \lambda dec.\ S']\!] \Box Spec \quad \Box c \mathrel{\widehat{=}} \lambda dec.\ S'}$	$\dfrac{[\![c \mathrel{\widehat{=}} \lambda dec.\ S']\!] \vdash Imp \to Spec}{\vdash (\mathsf{let}\ c \mathrel{\widehat{=}} \lambda dec.\ S'\ \mathsf{in}\ Imp) \to Spec}$
Design	$\dfrac{\Box subdesign\ args}{}$	$\dfrac{}{\vdash subdesign\ args \to subdesign\ args}$
Proof	$\dfrac{\vdash Thm}{} \quad \dfrac{\diamond Tm}{}$	$\dfrac{}{\vdash Thm} \quad \dfrac{}{\vdash Tm}$

There are eight techniques and these are adequate to develop any combinational circuit in a structured hierarchical way. The subgoaling and validation functions are both based on rules, given in Table 1. The subgoaling rules should be read top-down: to implement the goal above the line, solve all subgoals below the line. The validation rules operate in the opposite direction: the goal below the line is established from proofs of the goals above the line. These rules behave as follows:

Claim: The subgoaling function replaces the design goal $\Box Spec$ with a new one $\Box Spec'$, which is supplied by the user. This raises a proof obligation that $Spec'$ suffices for $Spec$. The user has to solve this proof obligation. The validation is based on the transitivity of implication, $\to$. It builds the correctness proof for $\Box Spec$ from the proof for $\Box Spec'$ and the proof given by the user.

Split: The subgoaling function decomposes a conjunctive specification into the problem of implementing each conjunct. The validation constructs a design from designs for the two subparts.

Reveal: If a design goal contains existentially quantified variables (expressing internal wiring), we must remove the quantifiers for further refinement (make the wiring explicit). This is done by the Reveal technique, which adds the typing information of internally quantified ports to the signature. The declarations added to the signature may later be used to establish the correctness of the implementation.

Inside: Since the initial design goal is given as a lambda abstraction, a technique is needed to remove this binding. To implement the circuit *Circ* means to implement the specification *Spec*. Similar to Reveal, the typing information of the input and output signals, which is declared in the declaration *dec*, is added to the support signature. The validation rule unfolds the definition of *Circ*; it states that the implementation is correct for all port-values (which are represented by the list of variables *port* and declared in the declaration *dec*) that can appear at the ports.

Library: The subgoaling function imports components from a database of predefined circuits and thus no new subgoals are generated by this technique. The validation corresponds to the use of a lemma, that is, we may assume the library component to be correct.

Subdesign: Subdesigns, similar to subroutines in programming languages, facilitate hierarchical development. A subdesign is submitted as a lambda abstraction by the user and the signature is extended with the given definition. The subdesign must be implemented (second subgoal) and may be used in proving the original design goal (first subgoal). The corresponding validation rule unfolds the definition in the correctness proof.

Design: Like the Library technique, this enables the user to apply a design that has already been implemented. In this case, the implementation is a subdesign introduced during the design process. As with the library technique, the validation theorem is a kind of assumption rule.

Proof: VERITAS is used to prove a theorem. The user can discharge a verification condition that arose during refinement or prove that a term is well-typed (recall, that type-checking in VERITAS is not decidable).

4. Implementation and Extension

We now describe our reinterpretation of Formal Synthesis in ISABELLE. We divide our presentation in several parts, which roughly correspond to ISABELLE theories that we implemented: circuit abstractions and 'concrete technology', technique reinterpretation, and new techniques.

4.1. Circuit Abstractions and Concrete Technology

We represent circuits as relations over port-values. We model kinds of signals as sets and we use ISABELLE's set theory (in HOL) to imitate the dependent types of the VERITAS logic; hence quantifiers used in our encoding are the bound quantifiers of ISABELLE's set theory. We name the connectives that serve as constructors for circuits with the following definitions.

$$C \ \mathsf{Sat}\ S \equiv C \rightarrow S \qquad \mathsf{Port}\, p : P.\ \phi(p) \equiv \forall p : P.\ \phi(p)$$
$$C_1\ \mathsf{Join}\ C_2 \equiv C_1 \wedge C_2 \qquad \mathsf{Wire}\, w : P.\ \mathit{Imp}(w) \equiv \exists\, w : P.\ \mathit{Imp}(w)$$

We will say more about the type P shortly. After definition, we formally derive rules which characterize these operators and use these rules in subsequent proofs (rather than expanding definitions). For example, using properties of conjunction and implication we derive (hence we are guaranteed of their correctness) in ISABELLE the following rules which relate Join to Sat.

$$\mathit{Imp}_1\ \mathsf{Sat}\ \mathit{Spec} \Longrightarrow (\mathit{Imp}_1\ \mathsf{Join}\ \mathit{Imp}_2)\ \mathsf{Sat}\ \mathit{Spec}$$
$$\mathit{Imp}_2\ \mathsf{Sat}\ \mathit{Spec} \Longrightarrow (\mathit{Imp}_1\ \mathsf{Join}\ \mathit{Imp}_2)\ \mathsf{Sat}\ \mathit{Spec}$$

Associating definitions with such characterization theorems increases comprehensibility and provides some abstraction: if we change definitions we can reuse our theories provided the characterization theorems are still derivable. Definitions have a second function, which is perhaps more subtle: they distinguish circuit constructors from propositional connectives and this restricts resolution (e.g., Join does not unify with $\wedge$) and makes it easier to write tactics that automate design.

Using the above abstractions, we can express the correctness of a (to be synthesized) circuit with respect to a specification as

$$\mathsf{Port}\, p_1 : P_1 \ \ldots\ p_n : P_n.\ ?\mathit{Imp}(p_1, \ldots, p_n)\ \mathsf{Sat}\ \mathit{Spec}(p_1, \ldots, p_n)\,.$$

This is not quite adequate though to simulate Formal Synthesis style proofs. A small problem is that the VERITAS Subdesign technique extends the signature with a new definition. In our setting we regard the new definition as a condition necessary for the validity of the correctness theorem and we express this condition using ordinary implication. Of course, when stating the initial goal we do not yet know which definitions will be made, so again we use a metavariable to leave this as an unknown. Thus the theorems we prove initially have the form

$$\mathsf{Def}(?\mathit{Definitions}) \rightarrow \mathsf{Port}\, p_1 : P_1 \ \ldots\ p_n : P_n.\ ?\mathit{Imp}(p_1, \ldots, p_n)\ \mathsf{Sat}\ \mathit{Spec}(p_1, \ldots, p_n)\,.$$

Analogous to the Formal Synthesis design goals, we call such a theorem a *design theorem*. The constant Def is simply the identity function and Def($?\mathit{Definitions}$) serves as a kind of 'definition context'.[1] Each time a definition is made, $?\mathit{Definitions}$ is instantiated to a conjunction of the definition and a fresh metavariable, which can be instantiated further with definitions made later.

In the design theorem, wires are bound to types P_i; these types are defined in theories about 'concrete technologies', e.g., representations of voltage, signals, and the like. A simple instance is where port values range over a datatype bin where voltage takes either the value Lo or Hi. We use ISABELLE's HOL set theory to define a *set* bin containing these two elements. Afterwards, we derive standard rules for bin, e.g., case-analysis over binary values.

We extend our theory with tactics that automate most common kinds of reasoning about binary values. For example, we have tactics that perform exhaustive analysis on Port values quantified over bin and tactics that combine such case analysis with ISABELLE's simplifier and heuristic based proof procedure (`fast_tac`) for HOL. These tactics automate almost all standard kinds of reasoning about combinational circuits.

We extend our theory of voltage with a theory of gates that declares the types of gates and axiomatizes their behavior. For example, the and gate is a predicate of type $[\mathsf{bin}, \mathsf{bin}, \mathsf{bin}] \Rightarrow \mathsf{bool}$ whose behaviour is axiomatized as

$$\mathsf{Port}\, a{:}\mathsf{bin}\ b{:}\mathsf{bin}\ c{:}\mathsf{bin}.\ \mathsf{and}(a,b,c)\ \mathsf{Sat}\ (c = \mathsf{Hi}) \leftrightarrow (b = \mathsf{Hi}) \wedge (a = \mathsf{Hi})\,.$$

Such axioms form part of a library of circuit specifications and are used to synthesize parts of circuits.

4.2. Reinterpreting Techniques

Table 2. ISABELLE Techniques (Name, Tactic, and Rule)

Claim: `claim_tac` *str i*
$[\![\ \mathit{Imp}\ \mathsf{Sat}\ \mathit{Spec}';\ \mathit{Spec}' \rightarrow \mathit{Spec}\ ]\!] \Longrightarrow \mathit{Imp}\ \mathsf{Sat}\ \mathit{Spec}$

Split: `split_tac` *i*
$[\![\ \mathit{Imp}_1\ \mathsf{Sat}\ \mathit{Spec}_1;\ \mathit{Imp}_2\ \mathsf{Sat}\ \mathit{Spec}_2\ ]\!] \Longrightarrow (\mathit{Imp}_1\ \mathsf{Join}\ \mathit{Imp}_2)\ \mathsf{Sat}\ (\mathit{Spec}_1 \wedge \mathit{Spec}_2)$

Reveal: `reveal_tac` *i*
$[\![\ !!w.\ w{:}T \Longrightarrow \mathit{Imp}(w)\ \mathsf{Sat}\ \mathit{Spec}(w)\ ]\!] \Longrightarrow (\mathsf{Wire}\, w{:}T.\ \mathit{Imp}(w))\ \mathsf{Sat}\ (\exists w{:}T.\mathit{Spec}(w))$

Inside: `inside_tac` *i*
$[\![\ !!p.\ p{:}T \Longrightarrow \mathit{Imp}(p)\ \mathsf{Sat}\ \mathit{Spec}(p)\ ]\!] \Longrightarrow \mathsf{Port}\, p{:}T.\ \mathit{Imp}(p)\ \mathsf{Sat}\ \mathit{Spec}(p)$

Library: `library_tac` *dels elims thm i*
$[\![\ \mathsf{Def}(H) \rightarrow P;\ \mathsf{Def}(H);\ P \Longrightarrow R\ ]\!] \Longrightarrow R$

Subdesign: `subdesign_tac` *str i*
$[\![\mathsf{Def}(\mathsf{Port}\, p_1{:}P_1\ \ldots\ p_n{:}P_n.\ \mathit{Name}(p_1,\ldots,p_n) = \mathit{Imp}'(p_1,\ldots,p_n));$
$\mathsf{Port}\, p_1{:}P_1\ \ldots\ p_n{:}P_n.\ \mathit{Imp}'(p_1,\ldots,p_n)\ \mathsf{Sat}\ \mathit{Spec}'(p_1,\ldots,p_n);$
$\mathsf{Port}\, p_1{:}P_1\ \ldots\ p_n{:}P_n.\ \mathit{Name}(p_1,\ldots,p_n)\ \mathsf{Sat}\ \mathit{Spec}'(p_1,\ldots,p_n) \Longrightarrow \mathit{Imp}\ \mathsf{Sat}\ \mathit{Spec}$
$]\!] \Longrightarrow \mathit{Imp}\ \mathsf{Sat}\ \mathit{Spec}$

Design: `design_tac` *dels elims i*
$[\![\ \mathsf{Port}\, x : T.\ P(x);\ P(x) \Longrightarrow R;\ x{:}T\ ]\!] \Longrightarrow R$

We have implemented tactics that simulate the first seven techniques (for the Proof technique we simply call the prover). Table 2 lists both the tactics and the derived rules corresponding to the VERITAS techniques. The tactics are based on the derived rules, which correspond to the validation rule associated with the VERITAS technique; their function is mostly self explanatory. In all tactics the parameter i specifies the subgoal the tactic is applied to.

Claim and **Split**: The tactics apply rules that are direct translations of the corresponding VERITAS validation rules. The specification Spec' is supplied as a string *str* to `claim_tac`.

Reveal and **Inside**: These are identical to the VERITAS techniques except that internal wiring, or quantification over ports, is converted to quantification at the metalevel (rather than extending the signature with declarations). The type constraints (expressed in ISABELLE's HOL set-theory) become assumptions. The rules state that if an implementation satisfies its specification, then we can wire a signal to a port with Inside or we can hide it as an internal wiring with Reveal. `reveal_tac` and `inside_tac` apply their respective rule as many times as possible to the specified goal.

Library: `library_tac` solves the design goal R by using a previously implemented component, supplied by the user as the component's design theorem *thm*. The rule is a form of implication elimination. The first subgoal is instantiated with the theorem *thm*. The second is solved by discharging it and thus extending the definition context of the overall design with the definitions of the component. The third establishes the correctness of the design using the specification of the library component. This involves type checking for the ports, which depends on the concrete technology of the designs; hence we supply the tactic with additional 'elimination' and 'deletion' rules, which solve the type checking goals.

Subdesign: We have given an informal rule schema (the others are formally derived) that represents infinitely many rules (there are arbitrarily many quantifiers, indicated by the ellipses). `subdesign_tac` simulates the effect of applying such a rule. The user gives the design goal of the subdesign to be implemented in the same form as the initial goal. Three new subgoals are generated by the tactic. The first corresponds to the subdesign definition and the tactic discharges this by adding it to the definition context of the main design. The second subgoal commits us to implement the subdesign. Here, the implementation is represented by the same metavariable as in the body of the definition and hence instantiations will be propagated to the definition. The third subgoal allows us to use the subdesign when proving the original goal.

Design: This solves a design goal by using a previously introduced subdesign. The subdesign has been given as a design goal, which is part of the assumptions of the goal to be solved (cf. the third subgoal of the Subdesign-method). The tactic removes all port quantifiers for the assumption by repeatedly applying the associated rule. For each port a new subgoal is generated, which concerns the type of that port, and is solved as in the Library technique; hence we provide lists of type checking rules to the tactic.

Proof: General proof goals arise from the application of the Claim rule. These are undecidable in general and must be proven by the user.

4.3. Extension with Induction

The techniques defined by Formal Synthesis are effective for developing combinational circuits. However, nontrivial circuits are often best developed as instances of parametric designs. For example, rather than designing a 16-bit adder it is preferable to develop one parameterized by word-length and afterwards to compute particular instances. We have developed new techniques that are compatible with our reinterpretation of Formal Synthesis and construct such parameterized circuits. Structural induction is used to build parameterized linear circuits and more generally n-dimensional grids, and course-of-values induction (also called Noetherian Induction) is used to build general recursively defined designs. We present the more general course-of-values method below and apply it afterwards to build a tree-structured addition circuit.

The idea for such extensions is motivated by previous work of ours on calculi in ISABELLE for synthesizing recursive logic [2] and functional programs [1]. There we developed rules and tactics based on induction that extend definition contexts with templates (a function or predicate name with a metavariable standing in for the body) for recursive definitions and leave the user with a goal to prove where use of the induction hypothesis builds a recursive program. This past work has much in common with our technique-based calculus for circuits. Syntactically, logic programs and circuits are similar: both can be described as conjunctive combinations of primitive relations where existential quantification 'passes values'. It turns out that we could (with very minor adaptation) directly use the rules and tactics we developed for synthesizing logic programs to build recursive circuits. We find this kind of reuse, not just of concepts but also of actual rules and tactics, an attractive advantage of interpreting different synthesis methodologies in a common framework. We will address this point again in Section 6 where we consider how techniques developed for the LAMBDA system can also be applied in the Formal Synthesis setting.

We construct a parameterized circuit by proving a parameterized design theorem, which is a design theorem where the outermost quantifier (or quantifiers) ranges over a subset of an inductively defined datatype like the natural numbers, e.g.,

$$\forall\, n{:}N.\ \mathsf{Port}\, p_1 : P_1\ \ldots\ p_m : P_m.\ \mathit{Imp}(n,\, p_1, \ldots, p_m)\ \mathsf{Sat}\ \mathit{Spec}(n,\, p_1, \ldots, p_m)$$

specifies an implementation whose size depends on the number n.

We use induction to instantiate $?\mathit{Imp}$ to a recursively specified design. ISABELLE's HOL comes with a theory of natural numbers (given as an inductive definition) from which it is easy to derive the following course-of-values induction rule.

$$[\![\ !!\, n.\ [\![\ n{:}N;\ \forall\, k{:}N.\ k < n \rightarrow P(k)\ ]\!] \Longrightarrow P(n)\ ]\!] \Longrightarrow \forall\, n{:}N.\ P(n)$$

We use this rule as the basis for a tactic, `cov_induct_tac`, which functions as if an instance of the following rule schema were applied.

$$
\begin{array}{l}
[\![\ \mathsf{Def}(\ \forall\, n{:}N.\ \mathsf{Port}\, p_1 : P_1\ \ldots\ p_m : P_m.\ \mathit{Imp}(n, p_1, \ldots, p_m) = \mathit{Definition}(n, p_1, \ldots, p_m)\ );\\[1ex]
\quad !!\, n.[\![n{:}N;\\
\qquad \forall\, k{:}N.\, k < n \rightarrow \mathsf{Port}\, p_1 : P_1\ \ldots\ p_m : P_m.\ \mathit{Imp}(k, p_1, \ldots, p_m)\ \mathsf{Sat}\ \mathit{Spec}(k, p_1, \ldots, p_m)\\
\quad\ \ ]\!] \Longrightarrow\quad \mathsf{Port}\, p_1 : P_1\ \ldots\ p_m : P_m.\ \mathit{Definition}(n, p_1, \ldots, p_m)\ \mathsf{Sat}\ \mathit{Spec}(n, p_1, \ldots, p_m)\\[1ex]
]\!] \Longrightarrow \forall\, n{:}N.\ \mathsf{Port}\, p_1 : P_1\ \ldots\ p_m : P_m.\ \mathit{Imp}(n, p_1, \ldots, p_m)\ \mathsf{Sat}\ \mathit{Spec}(n, p_1, \ldots, p_m)
\end{array}
$$

When an instance of this rule is applied by higher-order resolution, *Spec* will unify with the specification of the design theorem, and *Imp* with the metavariable standing for the circuit. The two assumptions of this rule generate two subgoals to be proven. The first subgoal sets up a parameterized circuit definition: an equality for *Imp* is defined, where it is equal to *Definition*, which will be a metavariable. Our tactic discharges this subgoal by adding it to the definitions of the main design. This leaves us with the second goal, which is the design goal. However, now we build a circuit named *Definition* and instantiating this in subsequent proof steps will instantiate our definition for *Imp*. Moreover, we now have a new assumption (the induction hypothesis),which states that *Imp* achieves *Spec* for smaller values of k. If we can reduce the problem to implementing a smaller design, then we can resolve with the induction hypothesis and this will build a recursive design by instantiating the definition body of *Imp* with an instance of *Imp*. How this works in practice will become clearer in Section 5.2. Course-of-values induction is usually used in conjunction with a rule for case-splitting (to create base and step cases) and in our setting this corresponds to the following rule.

$$[\![\ P \Longrightarrow \mathit{Imp}\ \mathsf{Sat}\ \mathit{Spec};\ \neg P \Longrightarrow \mathit{Imp}'\ \mathsf{Sat}\ \mathit{Spec}\ ]\!] \Longrightarrow \mathsf{if}(P, \mathit{Imp}, \mathit{Imp}')\ \mathsf{Sat}\ \mathit{Spec}$$

The tactic `if_tac` applies this rule instantiating P with a condition given as a string by the user.

Parameterized specifications require parameterized port types, e.g., rather than having input ports, we have parameterized input busses. To support this we develop a theory of busses encoded as lists of specified lengths (there are other possibilities, e.g. finite functions, but this allows us to directly use ISABELLE's list theory). Some of our definitions are as follows.

$$\begin{aligned}
B\ \ \mathsf{bus}\ (n) &\equiv \{b.\ \mathsf{length}(b) = n\ \wedge\ (\mathsf{Alls}\, x{:}b.\ x{:}B)\}\\
\mathsf{upper}(n,b) &\equiv \epsilon u.\ (\mathsf{length}(u) = n) \wedge (\exists\, l.\ (l@u) = b)\\
\mathsf{lower}(n,b) &\equiv \epsilon l.\ (\mathsf{length}(l) = n) \wedge (\exists\, u.\ (l@u) = b)\\
b`n &\equiv (\mathsf{nat_rec}(n,\ \mathsf{hd},\ \lambda\, u\, v\, r.\ v(\mathsf{tl}(r)))b)
\end{aligned}$$

A bus b of type $(B\ \ \mathsf{bus}\ (n))$ is a list of length n and whose members are from the set B. The functions upper and lower return the upper and lower n bits (when they exist) from b, and $b`n$ returns the nth element of b. We have proven many standard facts about these definitions, e.g., we can decompose busses

$$[\![\ k = n + m;\ b{:}(B\,\mathsf{bus}\,(k))\ ]\!] \Longrightarrow b = \mathsf{lower}(n,b)@\mathsf{upper}(m,b)$$

and we have derived typing rules, which can be used in the context of the Library and Design method (cf. Section 4.2).

$$\begin{aligned}
[\![\ b{:}(B\,\mathsf{bus}\,n);\ b'{:}(B\,\mathsf{bus}\,m)\ ]\!] &\Longrightarrow b@b'{:}(B\,\mathsf{bus}\,(n+m))\\
[\![\ k = n + m;\ b{:}(B\,\mathsf{bus}\,k)\ ]\!] &\Longrightarrow \mathsf{lower}(n,b){:}(B\,\mathsf{bus}\,n)\\
[\![\ k = n + m;\ b{:}(B\,\mathsf{bus}\,k)\ ]\!] &\Longrightarrow \mathsf{upper}(m,b){:}(B\,\mathsf{bus}\,m)
\end{aligned}$$

Data abstraction is provided by functions that interpret port values as natural numbers. The function vl maps bits to numbers and val extends this function to busses (i.e. lists, where [] is the empty list and # is the infix 'cons' operator on lists).

$$\mathsf{vl}(b) \;\equiv\; \mathsf{case}\ b\ \mathsf{of}\ \mathsf{Lo}\ \Rightarrow\ 0\ |\ \mathsf{Hi}\ \Rightarrow\ 1$$
$$\mathsf{val}([]) \;\equiv\; 0$$
$$\mathsf{val}(b\#bs) \;\equiv\; \mathsf{vl}(b) + 2 * \mathsf{val}(bs)$$

5. Examples

We present two examples. Our first example is a comparator developed using Formal Synthesis in [18]. It illustrates how we can directly mimic Formal Synthesis to create hierarchical designs. Our second example uses induction to construct a parameterized tree-shaped adder.

For both examples we give skeleton derivation trees (see figures 1 and 5) that show which techniques are applied in our ISABELLE derivations. We also show steps from the derivations themselves; all text in `typewriterfont` is taken, with minor pretty printing, directly from an Isabelle session. The steps in the derivation trees are labeled with numbers, which we refer to when describing the corresponding steps.

Note that there is not a fixed order in which subgoals must be solved in ISABELLE. Moreover, subdesigns may be introduced at any time in the design process. Thus a designer has considerable freedom in organizing proofs and the derivation trees for the developed circuits can vary, considerably, from those we present here.

5.1. Comparator Cell

A comparator takes as input two words A and B, representing numerals, and determines their relative order, i.e., which of the three cases $A < B$, $A = B$ or $A > B$ holds. Such a circuit can be built in a ripple-carry fashion from comparator cells. These cells (see the left-hand figure in Figure 2) compare a bit from A and B and also have three input bits (one for each case), which are the result of previous comparisons (*grin*, *eqin*, and *lsin*), and generate three output bits (*grout*, *eqout*, *lsout*). The behavioral specification is:

```
CompCellS(a, b, grin, eqin, lsin, grout, eqout, lsout) ==
    (vl(a) < vl(b) --> grout = Lo   & eqout = Lo   & lsout = Hi) &
    (vl(a) = vl(b) --> grout = grin & eqout = eqin & lsout = lsin) &
    (vl(b) < vl(a) --> grout = Hi   & eqout = Lo   & lsout = Lo)
```

As previously indicated, vl is defined in our theory of binary values: $\mathsf{vl}(\mathsf{Lo}) = 0$ and $\mathsf{vl}(\mathsf{Hi}) = 1$. We can now submit the following design theorem to ISABELLE.

```
Def(?H) -->
  (Port a:bin b:bin grin:bin eqin:bin lsin:bin grout:bin eqout:bin lsout:bin.
    ?Imp(a, b, grin, eqin, lsin, grout, eqout, lsout) Sat
    CompCellS(a, b, grin, eqin, lsin, grout, eqout, lsout))
```

We start the proof by applying an initialization tactic, which sets up the design goal by implication introduction (which moves the definition context into the assumption list) and applies the Inside tactic (step 1). Isabelle responds with the following proof state (we elide some information about the typing of the ports).

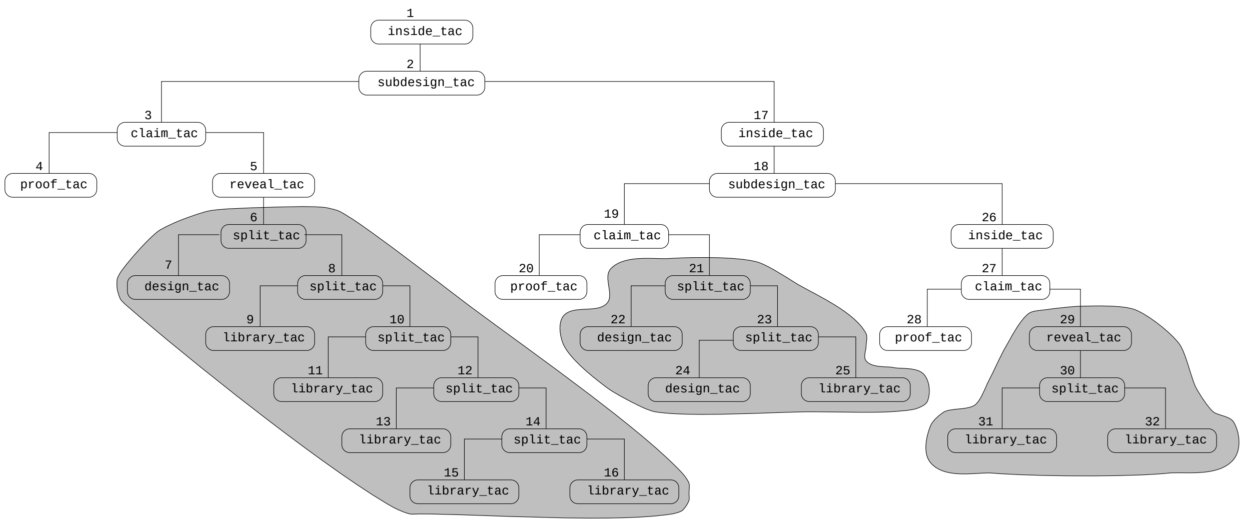

Figure 1. Derivation tree for comparator cell

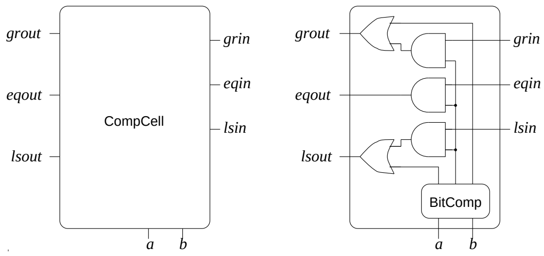

Figure 2. Comp-cell and its claimed implementation

```
Def(?H) --> (Port a:bin b:bin grin:bin eqin:bin lsin:bin grout:bin eqout:bin lsout:bin.
    ?Imp(a, b, grin, eqin, lsin, grout, eqout, lsout) Sat
    CompCellS(a, b, grin, eqin, lsin, grout, eqout, lsout))

 1. !! a  b grin eqin lsin grout eqout lsout.
       [| a:bin; ...; lsout:bin; Def(?H) |] ==>
       ?Imp(a, b, grin, eqin, lsin, grout, eqout, lsout) Sat
       CompCellS(a, b, grin, eqin, lsin, grout, eqout, lsout)
```

The original goal is given on the first-three lines; it contains two metavariables $?Imp$ and $?H$ standing for the implementation of CompCellS and the definitions that will be made, respectively. These are also present in the following lines (which contain the subgoal that must be established to prove the original goal) and will be instantiated when proving this subgoal. Next we introduce a new subdesign *BitComp* that we will use as a component. The idea is that we first compare the two bits a and b representing the current digit. Then we combine the result of this comparison with information coming from comparisons of less significant bits to give the result. The specification of the *BitComp* subdesign is the following.

```
BitCompS(a, b, gr, eq, ls) ==
   (vl(a) < vl(b) --> gr = Lo & eq = Lo & ls = Hi) &
   (vl(a) = vl(b) --> gr = Lo & eq = Hi & ls = Lo) &
   (vl(b) < vl(a) --> gr = Hi & eq = Lo & ls = Lo)
```

We apply `subdesign_tac` with this specification (step 2) and this yields two subgoals. At the top, we see our original goal where the definition context $?H$ was extended with a definition for *BitComp* and there is a new metavariable $?G$ for further definitions. The first subgoal is a design theorem for the subdesign. The second is the original design theorem but now with an additional assumption that there is an implementation of the subdesign that satisfies the specification BitCompS.

```
Def(?G &
    (Port a:bin b:bin gr:bin eq:bin ls:bin. ?BitComp(a,b,gr,eq,ls) = ?C(a,b,gr,eq,ls)))  -->
  Port a:bin b:bin grin:bin eqin:bin lsin:bin grout:bin eqout:bin lsout:bin.
    ?Imp(a, b, grin, eqin, lsin, grout, eqout, lsout) Sat
    CompCellS(a, b, grin, eqin, lsin, grout, eqout, lsout)

  1. Def(?G) ==>
```

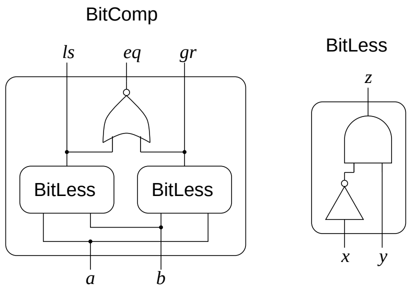

Figure 3. Claimed implementations of bit-comp and bit-less

```
    Port a:bin b:bin gr:bin eq:bin ls:bin. ?C(a,b,gr,eq,ls) Sat BitCompS(a,b,gr,eq,ls)

  2. !! a  b grin eqin lsin grout eqout lsout.
    [| a:bin; ...; lsout:bin; Def(?G);
       Port a:bin b:bin gr:bin eq:bin ls:bin.
          ?BitComp(a, b, gr, eq, ls) Sat BitCompS(a, b, gr, eq, ls) |] ==>
    ?Imp(a, b, grin, eqin, lsin, grout, eqout, lsout) Sat
    CompCellS(a, b, grin, eqin, lsin, grout, eqout, lsout)
```

Given this subdesign we use `claim_tac` (step 3) to state that the following specification entails the original goal (Figure 2).

```
EX gr:bin eq:bin ls:bin x:bin y:bin.
    BitCompS(a, b, gr, eq, ls) & andS(eq, grin, x) & orS(gr, x, grout) &
    andS(eq, eqin, eqout) & andS(eq, lsin, ls) & orS(ls, y, lsout)
```

The proof continues with a number of further refinement steps (steps 4 to 16). Let us sketch some of the remaining steps. First we show that the claimed specification entails the original one; we prove this automatically using a tactic that performs case-analysis and simplification (step 4). After, we implement it by using `reveal_tac` (step 5) to strip existential quantifications (and introduce internal wires) and then use `split_tac` to break up the conjunctions and Join together subcircuits (steps 6 to 16). The components are implemented using `library_tac`, which accesses appropriate library parts, and using `design_tac`, to apply the subdesign BitComp; this leaves us with the following proof state.

```
Def(?G &
    (Port a:bin b:bin gr:bin eq:bin ls:bin. ?BitComp(a,b,gr,eq,ls) = ?C(a,b,gr,eq,ls)))  -->
  Port a:bin b:bin grin:bin eqin:bin lsin:bin grout:bin eqout:bin lsout:bin.
    ( Wire gr:bin eq:bin ls:bin x:bin y:bin.
        ?BitComp(a, b, gr, eq, ls) Join and(eq, grin, x) Join or(gr, x, grout) Join
        and(eq, eqin, eqout) Join and(eq, lsin, ls) Join or(ls, y, lsout))Sat
    CompCellS(a, b, grin, eqin, lsin, grout, eqout, lsout)

  1. Def(?G) ==>
     Port a:bin b:bin gr:bin eq:bin ls:bin. ?C(a,b,gr,eq,ls) Sat BitCompS(a,b,gr,eq,ls)
```

To implement the subdesign BitCompS we proceed in the same manner as earlier and we introduce a new subdesign *BitLess* (step 18). After, we build the above *BitComp* using *BitLess* twice as shown in Figure 3 (steps 19 to 25). Finally, BitLessS is so simple that we can claim a direct implementation consisting of components from the library. Solving the proof obligation and revealing the internal wiring (steps 28 and 29) results in the following proof state.

```
Def(?G &
    (Port a:bin b:bin gr:bin eq:bin ls:bin.
       ?BitComp(a,b,gr,eq,ls) = ?BitLess(a,b,ls) Join ?BitLess(b,a,gr) Join nor(ls,gr,eq)) &
    (Port x:bin y:bin z:bin. ?BitLess(x, y, z) = Wire l. ?C(x, y, z, l))) -->
  Port a:bin b:bin grin:bin eqin:bin lsin:bin grout:bin eqout:bin lsout:bin.
    ( Wire gr:bin eq:bin ls:bin x:bin y:bin.
        ?BitComp(a, b, gr, eq, ls) Join and(eq, grin, x) Join or(gr, x, grout) Join
        and(eq, eqin, eqout) Join and(eq, lsin, ls) Join or(ls, y, lsout))
    Sat CompCellS(a, b, grin, eqin, lsin, grout, eqout, lsout)

  1. !! x y z l. [|Def(?G); x:bin; y:bin; z:bin; l:bin|] ==>
        ?C(x, y, z, l) Sat invS(x, l) & andS(l, y, z)
```

We split this design-goal (step 30) into two new design-goals that can be implemented by the inverter and the and-gate from the library using `library_tac` (steps 31 and 32). After these steps, the design theorem that we have proved is the following.

```
Def((Port a:bin b:bin gr:bin eq:bin ls:bin.
       ?BitComp(a,b,gr,eq,ls) = ?BitLess(a,b,ls) Join ?BitLess(b,a,gr) Join nor(ls,gr,eq)) &
    (Port x:bin y:bin z:bin.
       ?BitLess(x, y, z) = Wire l. inv(x, l) Join and(l, y, z))) -->
  Port a:bin b:bin grin:bin eqin:bin lsin:bin grout:bin eqout:bin lsout:bin.
    ( Wire gr:bin eq:bin ls:bin x:bin y:bin.
        ?BitComp(a, b, gr, eq, ls) Join and(eq, grin, x) Join or(gr, x, grout) Join
        and(eq, eqin, eqout) Join and(eq, lsin, ls) Join or(ls, y, lsout))
    Sat CompCellS(a, b, grin, eqin, lsin, grout, eqout, lsout)
```

The metavariable $?H$ has become instantiated with the conjunction of the definitions of the subdesigns used in the implementation of the main design goal, i.e., *BitComp* and *BitLess*. In the main goal, the unknown $?Imp$ has become a predicate that represents the structure of the desired circuit and is built from these subdesigns and additional and and or gates. Overall, our proof builds the same circuit and uses the identical sequence of technique applications as that presented in [18].[2]

The difference between our derivation and the Formal Synthesis one is not so much in the techniques applied, but rather the underlying conceptualization and implementation: in the VERITAS system, the implementation is constructed at the end of the proof by the validation functions, whereas in our setting, the design takes shape incrementally during the proof. We find this advantageous since we can directly see the effects of each design decision taken.

5.2. Carry Lookahead Adder

Our second example illustrates how our well-founded induction technique synthesizes parameterized designs: We synthesize a carry lookahead adder (henceforth,

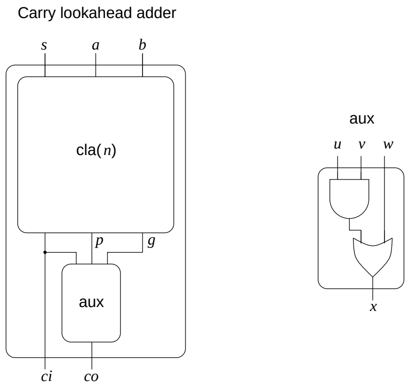

Figure 4. Implementation of a cla-adder from two components

cla-adder) that is parametric in its bit-width n. For n-bit numbers, such an adder has a height that is proportional to $\log(n)$ and thus computes the sum s and a carry c_o from two numbers a, b and an incoming carry c_i in $O(\log(n))$ time. Instead of propagating the carry from digit to digit as it is done in a ripple-carry adder, we compute more detailed information (cf. [19]). A *generate bit* g indicates when a carry is generated by adding the digits of a and b and a *propagate bit* p indicates if an incoming carry is handed through. From this information we obtain the *carry bit* c_o for the adder in the following way: it is Hi if $g = \mathsf{Hi}$ or if both $p = \mathsf{Hi}$ and $c_i = \mathsf{Hi}$. A carry lookahead adder is implemented, roughly speaking, by recursively decomposing it into two adders, each half the size of the original. The propagate and generate bits for the overall adder are obtained by combining the corresponding bits of the subparts with the incoming carry c_i. In the case of adding single digits of a and b (the base case of the recursion), the propagate bit corresponds to the logical or and the generate bit corresponds to the logical and of the digits.

The adder we synthesize is built from two components (Figure 4). The first, *cla*, computes the sum s, the propagate bit p and the generate bit g from the numbers a and b and the incoming carry c_i. The second is an auxiliary component *aux*, which is used to combine the propagate bit, the generate bit and the incoming carry to the outgoing carry c_o. This component consists of two gates and can be derived in a few steps. We focus here on the development of the more interesting component *cla*. We can specify its behavior using data abstraction by an arithmetic expression.

```
claS(n,a,b,s,p,g,ci) ==
   case g of
     Lo => val(s) + 2^n*vl(p)*vl(ci) = val(a) + val(b) + vl(ci)
   | Hi => val(s) +              2^n = val(a) + val(b) + vl(ci)
```

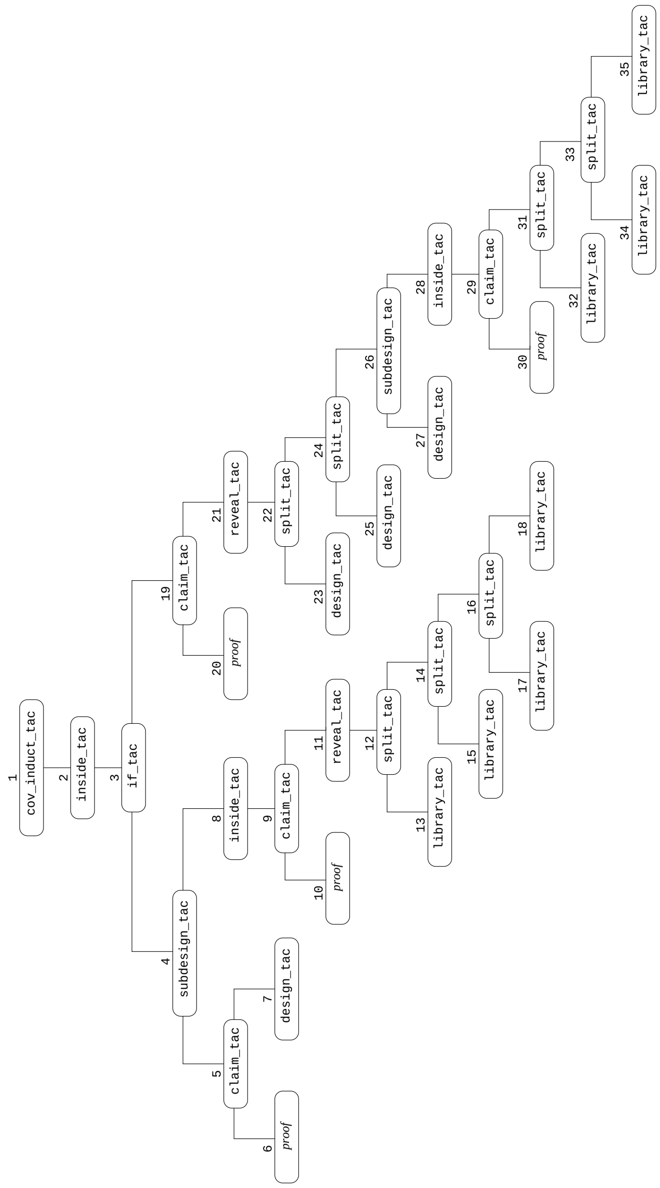

Figure 5. Derivation tree for cla unit

Note that numbers are represented by busses (i.e. bit vectors) and the value of a bus as a natural number is given by val. We assume busses to have a nonzero length. Hence in the following design theorem, we restrict induction to the set nnat(1) of natural numbers greater than zero.

```
Def(?H)-->
  ( !n:nnat(1). Port a:(bin bus n) b:(bin bus n) s:(bin bus n) p:bin g:bin ci:bin.
    ?cla(n,a,b,s,p,g,ci) Sat claS(n,a,b,s,p,g,ci))
```

As before, we begin by shifting the definition environment Def($?H$) to the assumptions and applying the course-of-values induction tactic (step 1), which yields the following proof state.

```
Def((ALL n:nnat(1). Port a:(bin bus n) b:(bin bus n) s:(bin bus n) p:bin g:bin ci:bin.
           ?cla(n, a, b, s, p, g, ci) = ?D16(n, a, b, s, p, g, ci)) &
    ?Q2) -->
(ALL n:nnat(1). Port a:(bin bus n) b:(bin bus n) s:(bin bus n) p:bin g:bin ci:bin .
       ?cla(n, a, b, s, p, g, ci) Sat claS(n, a, b, s, p, g, ci))

 1. !!n. [| Def(?Q2); n:nnat(1);
            ALL k:nnat(1). k < n -->
               (Port a:(bin bus k) b:(bin bus k) s:(bin bus k) p:bin g:bin ci:bin .
                   ?cla(k, a, b, s, p, g, ci) Sat claS(k, a, b, s, p, g, ci)) |] ==>
         Port a:(bin bus n) b:(bin bus n) s:(bin bus n) p:bin g:bin ci:bin .
            ?D16(n, a, b, s, p, g, ci) Sat claS(n, a, b, s, p, g, ci)
```

As previously described, $?H$ is extended with a definition template and further definitions are collected in $?Q_2$. The metavariable at the left-hand side of the definition serves as the name for the design being defined. The metavariable on the right-hand side will be instantiated with the implementation when we prove subgoal 1. The induction hypothesis is added to the assumptions of that subgoal: We assume that we have an implementation for all k less than n and we can use this to build a circuit of size n.

We proceed by performing a case analysis on n. We type `by(if_tac "n=1" 1)`, which resolves subgoal 1 with the rule

$$[\![\, P \Longrightarrow \mathit{Imp}\ \mathsf{Sat}\ \mathit{Spec};\ \neg P \Longrightarrow \mathit{Imp}'\ \mathsf{Sat}\ \mathit{Spec}\,]\!] \Longrightarrow \mathsf{if}(P, \mathit{Imp}, \mathit{Imp}')\ \mathsf{Sat}\ \mathit{Spec}\,,$$

where P is instantiated with $n = 1$ (step 3). ISABELLE responds:

```
Def((ALL n:nnat(1). Port a:(bin bus n) b:(bin bus n) s:(bin bus n) p:bin g:bin ci:bin .
        ?cla(n, a, b, s, p, g, ci) =
           if(n = 1, ?C23(n, a, b, s, p, g, ci), ?C'23(n, a, b, s, p, g, ci))) & ?Q2) -->
(ALL n:nnat(1). Port a:(bin bus n) b:(bin bus n) s:(bin bus n) p:bin g:bin ci:bin .
       ?cla(n, a, b, s, p, g, ci) Sat claS(n, a, b, s, p, g, ci))
 1. !!n a b s p g ci.
       [| Def(?Q2); n:nnat(1);
          ALL k:nnat(1). k < n -->
             (Port a:(bin bus k) b:(bin bus k) s:(bin bus k) p:bin g:bin ci:bin .
                 cla(k, a, b, s, p, g, ci) Sat claS(k, a, b, s, p, g, ci));
          a:bin bus n; b:bin bus n; s:bin bus n; p:bin; g:bin; ci:bin; n = 1 |] ==>
       ?C23(n, a, b, s, p, g, ci) Sat claS(n, a, b, s, p, g, ci)

 2. !!n a b s p g ci.
       [| Def(?Q2); n:nnat(1);
```

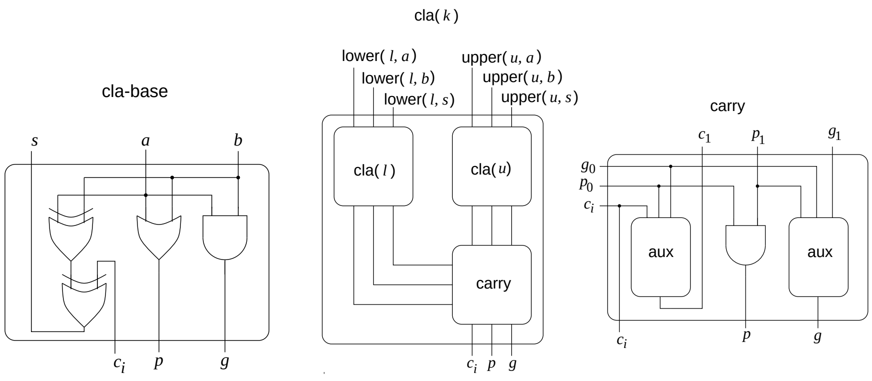

Figure 6. Base case (left) and recursive decomposition (middle/right) where $l = k\,\mathsf{div}\,2$ and $u = k\,\mathsf{div}\,2 + k\,\mathsf{mod}\,2$

```
        ALL k:nnat(1). k < n -->
           (Port a:(bin bus k) b:(bin bus k) s:(bin bus k) p:bin g:bin ci:bin .
               cla(k, a, b, s, p, g, c) Sat claS(k, a, b, s, p, g, ci));
        a:bin bus n; b:bin bus n; s:bin bus n; p:bin; g:bin; ci:bin; n ~= 1 |] ==>
     ?C'23(n, a, b, s, p, g, ci) Sat claS(n, a, b, s, p, g, ci)
```

In the overall design goal, the right-hand side of the definition has been instantiated with a conditional whose alternatives $?C_{23}$ and $?C'_{23}$ are the respective implementations for the base and the step case. The former is to be implemented by proving subgoal 1 under the assumption $n = 1$ the latter by proving subgoal 2 under the assumption $n \neq 1$. The base case is solved by the subdesign *cla_base* (Figure 6) in a few simple steps (steps 4 to 18).

In the step case we build the adder from two smaller adders of half the size: We decompose the busses a, b and s each in a segment $\mathsf{lower}(n\,\mathsf{div}\,2, \_)$ containing the inferior $n\,\mathsf{div}\,2$ bits of the bus and a segment $\mathsf{upper}(n\,\mathsf{div}\,2 + n\,\mathsf{mod}\,2, \_)$, containing the remaining bits. The lower (upper) segments of a and b can be added by an adder of bit-width $n\,\mathsf{div}\,2$ $(n\,\mathsf{div}\,2 + n\,\mathsf{mod}\,2)$ yielding the lower (upper) segment of the sum s. The propagate, generate and carry bit for the overall adder and the carry flowing from the lower to the upper part of the adder are computed by some additional circuitry collected in a new subdesign. Accordingly we can reformulate the specification for the component *cla* as follows (step 19).

```
by( claim_tac
" EX p0:bin. EX g0:bin. EX p1:bin. EX g1:bin. EX c1:bin.
 claS(n div 2, lower(n div 2, a), lower(n div 2, b), lower(n div 2, s), p0, g0, c) &
 claS(n div 2 + n mod 2, upper(n div 2 + n mod 2, a), upper(n div 2 + n mod 2, b),
      upper(n div 2 + n mod 2, s), p1, g1, c1 ) &
 carryS(p0, g0, p1, g1, c1, p, g, ci)" 1 );
```

The proof that this new specification entails the original is accomplished automatically by a tactic we implemented that performs exhaustive case analysis on the values of the carry, propagate and generate bits and performs simplification of arithmetic expressions (step 20).

We decompose the new specification into its subparts and implement the recursive occurrences of the specification claS by using the induction hypothesis. This is done by applying design_tac twice (steps 23 and 25). The remaining specification carryS is solved by a new subdesign, which is implemented as shown in Figure 6. Note that we reuse the formerly developed *aux* here. Thus, all design goals are solved and after 35 steps, completing the derivation.

```
Def((ALL n:nnat(1). Port a:(bin bus n) b:(bin bus n) s:(bin bus n) p:bin g:bin ci:bin.
      ?cla(n, a, b, s, p, g, ci) =
      if(n = 1, ?cla_base(a`0, b`0, s`0, p, g, ci),
         Wire p0:bin g0:bin p1:bin g1:bin c1:bin.
         ?cla(n div 2, lower(n div 2,a), lower(n div 2,b), lower(n div 2,s), p0, g0, ci) Join
         ?cla(n div 2 + n mod 2,upper(n div 2 + n mod 2, a),upper(n div 2 + n mod 2, b),
              upper(n div 2 + n mod 2,s),p1,g1,c1) Join ?carry(p0,g0,p1,g1,c1,p,g,ci))) &
    (Port a:bin b:bin s:bin p:bin g:bin ci:bin.
      ?cla_base(a, b, s, p, g, ci) =
      (Wire w:bin. xor(a, b, w) Join xor(w, ci, s) Join or(a, b, p) Join and(a, b, g))) &
    (Port p0:bin g0:bin p1:bin g1:bin c1:bin p:bin g:bin  ci:bin .
      ?carry(p0, g0, p1, g1, c1, p, g, ci) =
      (?aux(p0, ci, g0, c1) Join ?aux(g0, p1, g1, g) Join and(p0, p1, p))) &
    (Port u:bin v:bin w:bin x:bin .
      ?aux(u, v, w, x) = (Wire wa:bin. and(u, v, wa) Join or(wa, w, x)))) -->
(ALL n:nnat(1). Port a:(bin bus n) b:(bin bus n) s:(bin bus n) p:bin g:bin ci:bin .
      ?cla(n, a, b, s, p, g, ci) Sat claS(n, a, b, s, p, g, ci))
```

In our definition context, $?H$ has become instantiated by four definitions. There is a one to one correspondence between the implementations shown in Figures 4 and 6 and the predicates defining them.

6. Comparison and Conclusion

We have combined two development methodologies: Formal Synthesis as implemented in VERITAS and resolution based synthesis in ISABELLE. The result is a simple realization of Formal Synthesis that is compatible with other approaches to resolution based synthesis. Moreover, our implementation supports structural, behavioral, and data-abstraction as well as independence from the concrete circuit technology. Our implementation is based on extensions to higher-order logic and we were able to directly utilize standard Isabelle theories (e.g., inductive definitions) in our work as well as Isabelle's simplification tactics. Most of our derived rules, (e.g., those implementing techniques) were proven quickly in one step by ISABELLE's classical prover, a tactic that automates the construction of many proofs in higher-order logic.

The idea of using first-order resolution to build programs goes back to Green in the 1960s [15]. More recently, within systems like ISABELLE, interactive proof by higher-order resolution has been used to construct verified programs and hardware designs [2, 8, 23]. The work most closely related to ours is that of Mike Fourman's group based on the LAMBDA system, which is a proof development system that supports synthesis based on second-order resolution [10, 11]. Motivated by

ISABELLE, they too use rules in order to represent the design state. The difference lies in the particular approach they use in proof construction;[3] instead of using general purpose techniques as in Formal Synthesis, they derive (introduction) rules for each component from its definition. These rules are applied to the proof state in order to simplify the specification and thereby refine the implementation. The specialized form of their rules supports a higher degree of automation than general purpose techniques. Conversely, the generality of the Formal Synthesis techniques provides a more abstract view of the design process and better supports hierarchical development.

Just as we have extended Formal Synthesis with techniques for induction, it is possible to adapt the LAMBDA methodology within our setting. We have carried out some experiments that indicate both that we can use our Formal Synthesis techniques to synthesize LAMBDA style design rules and that such rules can be combined with techniques in circuit development. As a simple illustration, suppose we have an axiom for an adder circuit given by

$$\mathsf{Port}\, a{:}\mathsf{nat}\, b{:}\mathsf{nat}\, s{:}\mathsf{nat}.\ \mathsf{addc}(a,b,s)\ \mathsf{Sat}\ \ s = a + b\,.$$

We can apply our techniques to synthesize a partial implementation for a schematic specification *Spec*, which contains a subexpression of the form $a + b$.

```
Port a:nat b:nat. ?Imp(a,b) Sat Spec(a+b)
```

After applying the techniques Inside, Claim, Reveal and Split, and solving the proof obligation from Claim, we arrive at the following intermediate proof state.

```
Port a:nat b:nat.
   (Wire s:nat. addc(a, b, s) Join ?C2(a, b, s)) Sat Spec(a + b)
 1. !!a b s. [| a:nat; b:nat; s:nat |] ==> ?C2(a, b, s) Sat Spec(s)
```

After discharging the first subgoal by assuming it and removing the Port quantifiers, we arrive at a LAMBDA-style design rule.

$$\begin{aligned}&[\![\ !!\ a\, b\, s.[\![\ a{:}\mathsf{nat};\ b{:}\mathsf{nat};\ s{:}\mathsf{nat}\ ]\!] \Longrightarrow\ ?C(a,b,s)\ \mathsf{Sat}\ ?\mathit{Spec}(s);\\ &\quad ?a{:}\mathsf{nat};\ ?b{:}\mathsf{nat}\\ &]\!] \Longrightarrow\ (\mathsf{Wire}\, s{:}\mathsf{nat}.\ \mathsf{addc}(?a,?b,s)\ \mathsf{Join}\ ?C(?a,?b,s))\ \mathsf{Sat}\ ?\mathit{Spec}(?a{+}?b)\end{aligned}$$

Explaining this as a technique, it says that we can reduce a specification involving the addition of $a{+}b$ to one instead involving s. The validation tells us that the circuit built for $\mathit{Spec}(s)$, when hooked up appropriately to an adder, builds a circuit for the original specification. It would not be difficult to build tactics that enable us to integrate such LAMBDA techniques with the others we developed, taking advantage of the different strengths of these two approaches.

We conclude with a brief mention of deficiencies and future work. Currently, the amount of information present during proof can be overwhelming. A short-term solution is to instruct Isabelle to elide information; however, a graphical interface, like that in LAMBDA, would be of tremendous value both in displaying designs and giving specifications. Another weakness is automation. We have automated many simple kinds of reasoning by combining Isabelle's simplifiers with case-analysis over binary values. The resulting tactics are effective, but their execution is slow. There are decision procedures based on BDDs that can more effectively solve many of these problems. We have started integrating one of these with our synthesis environment, namely, a decision procedure for a decidable monadic logic that is well-suited for modeling hardware [3]. We hope that this is a step towards a synthesis framework in which different verification methodologies may be integrated.

Notes

1. Not only does Def 'mark' which part of a formula or a proof state consists of the definitions, it also prevents higher-order unification from trivially solving goals, i.e., instantiating *?Definitions* with the antecedent of the design goal.
2. We have actually automated it considerably more though. For instance, in the claimed specification for the comparator cell, the subparts consisted of conjoined specifications of simple gates. We have a tactic, which takes a library (list of design theorems for components), and automatically proves these kinds of goals. In the derivation-tree those parts are highlited gray.
3. In the end, there are only so many (700?) proof development systems, but there are many more strategies for constructing proofs.

References

1. A. Ayari and D. Basin. Generic system support for deductive program development. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS-96)*, volume 1055 of *LNCS*, pages 313–328, Passau, Germany, March 1996. Springer-Verlag.
2. D. Basin. Logic frameworks for logic programs. In *4th International Workshop on Logic Program Synthesis and Transformation, (LOPSTR'94)*, volume 883 of *LNCS*, pages 1–16, Pisa, Italy, June 1994. Springer-Verlag.
3. D. Basin and N. Klarlund. Hardware verification using monadic second-order logic. In *Computer-Aided Verification (CAV '95)*, volume 939 of *LNCS*, pages 31–41, Pisa, Italy, June 1995. Springer-Verlag.
4. J. L. Bates and R. L. Constable. Proofs as programs. *ACM Transactions on Programming Languages and Systems*, 7(1):113–136, January 1985.
5. W. Bibel. Syntax-directed, semantics-supported program synthesis. *Artificial Intelligence*, 14(1):243–261, 1980.
6. R. Bird. Lectures on constructive functional programming. In *Constructive Methods in Computing Science*. Springer-Verlag, 1989.
7. A. J. Camilleri, M. J. C. Gordon, and T. F. Melham. Hardware verification using higher-order logic. In D. Borrione, editor, *From HDL Descriptions to Guaranteed Correct Circuit Designs*. North Holland, September 1986.
8. M. D. Coen. Interactive program derivation. Technical Report 272, Cambridge University Computer Laboratory, Cambridge, 1992.
9. T. Coquand and G. Huet. The Calculus of Constructions. *Information and Computation*, pages 95–120, 1988.
10. S. Finn, M. P. Fourman, M. Francis, and R. Harris. Formal system design — interactive synthesis based on computer-assisted formal reasoning. In D. Claesen, editor, *IMEC-IFIP International Workshop on Applied Formal Methods for Correct VLSI Design*, volume 1, pages 97–110, Amsterdam, 1989. Elsevier Science publishers B.V. (North-Holland).
11. M. P. Fourman. Proof and synthesis. In *International Conference on Circuit Design*, pages 600–603, Rye, New York, 1988.
12. M. P. Fourman. Formal methods for modelling design. In *Conference on Modelling the Innovation: Communications, Automation and Information Systems*, Rome, Italy, 1990.
13. M. P. Fourman and E. M. Mayger. Formally based system design — interactive hardware scheduling. In *Proceedings of VLSI '89*, pages 101–112, Munich, Germany, 1989.

14. M. Gordon and T. Melham. *Introduction to HOL: A Theorem Proving Environment for Higher Order Logic.* Cambridge University Press, 1993.
15. C. Green. Application of theorem proving to problem solving. In *Proceedings of the IJCAI-69*, pages 219–239, 1969.
16. F. Hanna, N. Daeche, and G. Howells. Implementation of the Veritas Design Logic. In V. Stavridon, T.F. Melham, and R.T. Boute, editors, *Proc. of the International Conference on Theorem Provers in Circuit Design: Theory, Practice and Experience*, pages 77–94, Nijmegen, June 1992. IFIP TC10/WG 10.2, North-Holland.
17. F. Hanna, N. Daeche, and M. Longley. VERITAS+: A specification language based on type theory. In *Hardware Specification, Verification and Synthesis: Mathematical Aspects*, Ithaca, New York, 1989. Springer-Verlag.
18. F. K. Hanna, N. Daeche, and M. Longley. Formal synthesis of digital systems. In *IMEC-IFIP International Workshop on: Applied Formal Methods For Correct VLSI Design*, volume 2, pages 532–548, Leuven, Belgium, 1989.
19. J. Hennessy and D. Patterson. *Computer Architecture, a Quanitative Approach.* Morgan Kaufmann, 1990.
20. C. Hogger. Derivation of logic programs. *J. ACM*, 28(2):372–392, April 1981.
21. G. Jones and M. Sheeran. Designing arithmetic circuits by refinement in Ruby. In M. Bird and Woodcock, editors, *Mathematics of Program Construction*, volume 669 of *LNCS*. Springer-Verlag, 1992.
22. Z. Manna and R. Waldinger. Fundamentals of the deductive program synthesis. *IEEE Transactions on Software Engineering*, January 1992.
23. L. C. Paulson. *Isabelle : a generic theorem prover; with contributions by Tobias Nipkow*, volume 828 of *LNCS*. Springer-Verlag, 1994.
24. O. Rasmussen. Formalising Ruby in Isabelle ZF. In L. Paulson, editor, *Proceedings of the First Isabelle Users Workshop*, pages 246–265, Cambridge, England, 1995. University of Cambridge, Computer Laboratory.
25. L. Rossen. Proving (facts about) Ruby. In G. Birtwistle, editor, *IV Higher Order Workshop, Banff 1990*, Workshops in Computing, pages 265–284, Heidelberg, 1991. Springer-Verlag.
26. L. Wang and M. Mendler. Formal design of a class of computers. In P. Camurati and H. Eveking, editors, *Correct Hardware Design and Verification Methods*, volume 987 of *LNCS*, pages 84–102, Heidelberg, 1995. Springer-Verlag.