Modeling a Hardware Synthesis Methodology in Isabelle

David Basin and Stefan Friedrich

Max-Planck-Institut für Informatik
Im Stadtwald, D-66123, Saarbrücken, Germany

Abstract. *Formal Synthesis* is a methodology developed at Kent for combining circuit design and verification. We have reinterpreted this methodology in ISABELLE's theory of higher-order logic so that circuits are synthesized 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 synthesis methodologies and leads to techniques for developing new classes of circuits, e.g., recursive descriptions of parameterized circuits.

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. We present such an approach here, using resolution to synthesize circuit designs during proofs of their correctness. Our approach is based on modeling a particular methodology for hierarchical design development within a synthesis by resolution framework.

Our starting point is a novel methodology for hardware synthesis, called *Formal Synthesis*, proposed by the VERITAS group at Kent [7]. 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 $Circ_i$ achieve $Spec_i$, and constructs an implementation *Circ* with a proof that *Circ* 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 which 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 implementation too and Formal Synthesis required extending the VERITAS theorem prover itself.

We show how to reinterpret Formal Synthesis as deductive synthesis based on higher-order resolution. To do this, we begin 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 which 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 $?Circ \rightarrow Spec$, where the question mark means that *Circ* is a metavariable. If *Spec* is a conjunction, $Spec_1 \wedge Spec_2$, then we can resolve $?Circ \rightarrow Spec_1 \wedge Spec_2$ with the rule in (1). The result is a substitution, $?Circ = ?Imp_1 \wedge ?Imp_2$, 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 $?Circ$.

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 [1] 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.

2 Background

We assume familiarity with ISABELLE [10] and higher-order logic. Space limitations restrict us to reviewing only notation and a few essentials.

ISABELLE is an interactive tactic-based theorem prover. 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 $\wedge$ and a typical proof rule would be $!!A, B : Form.\, A \Longrightarrow (B \Longrightarrow A \wedge B)$. Here $\Longrightarrow$ and $!!$ are implication and universal quantification in ISABELLE's metalogic. 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, extended with theories of sets, well founded recursion, natural numbers, and the like.

Isabelle supports proof construction by higher-order resolution. Given a proof state with subgoal ψ and a proof rule $[\![\phi_1; \ldots; \phi_n]\!] \Longrightarrow \phi$, we unify ϕ with ψ. If this succeeds, then unification yields a substitution σ and the proof state is updated by applying σ to it and replacing ψ 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 use 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 [8] is a tactic based theorem prover similar to ISABELLE and HOL, 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. When used to reason about hardware, one proves theorems that relate circuits, *Circ*, to specifications, *Spec*, e.g.,

$$\forall p_1 \ldots p_n.\ \mathit{Circ}(p_1, \ldots, p_n) \rightarrow \mathit{Spec}(p_1, \ldots, p_n)\,,$$

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 [3]. 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, we would formalize this requirement on word-length in the types.

3 Formal Synthesis

Formal Synthesis is based on *techniques*. As previously indicated, each technique combines a subgoaling function with a validation. The validations are executed when a derivation is completed to build a proof that a circuit achieves an implementation. Before giving the techniques we introduce some notation used by Formal Synthesis. A design specification *spec* which is to be implemented is written $\Box \mathit{spec}$ ($\Box$ is just a symbol; there is no relationship to modal logics). A formula *Thm* to be proved is written as $\vdash \mathit{Thm}$. A term *Tm* to be demonstrated well typed is written as $\Diamond \mathit{Tm}$. A proof in VERITAS is associated with a support

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

Table 1. VERITAS design techniques

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 $[\![extension]\!]\,Spec$. Finally, the initial design goal must be of the form $Circ \mathrel{\widehat{=}} \lambda dec.\ Spec$ where *Circ* is the name assigned to the circuit.

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. 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 yields a new design goal *Spec'* and a subgoal that this suffices for *Spec*. The validation is based on the transitivity of $\rightarrow$.

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: Shifts internally quantified ports into the signature, allowing further refinement.

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*. The validation theorem states that the implementation is correct for all port-values that can appear at the ports *port*, which are declared in the declaration *dec*.

Library: The subgoaling function imports components from a database of predefined circuits. The validation corresponds to the use of a lemma.

Subdesign: A subdesign, submitted as a lambda abstraction, may be introduced. It must be implemented (second subgoal) and may be used in proving the original design goal (first subgoal).

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.

Proof: VERITAS is used to prove a theorem.

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.

$$\begin{array}{ll} C\ \mathsf{Sat}\ S \equiv C \rightarrow S & \mathsf{Port}\, p\!:\!P.\ \phi(p) \equiv \forall p\!:\!P.\ \phi(p) \\ C_1\ \mathsf{Join}\ C_2 \equiv C_1 \wedge C_2 & \mathsf{Wire}\, w\!:\!W.\ \mathit{Circ}(w) \equiv \exists\, w\!:\!W.\ \mathit{Circ}(w) \end{array}$$

We will say more about the types P and W shortly. After definition, we 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.

$$\begin{array}{l} \mathit{Circ}_1\ \mathsf{Sat}\ \mathit{Spec} \Longrightarrow (\mathit{Circ}_1\ \mathsf{Join}\ \mathit{Circ}_2)\ \mathsf{Sat}\ \mathit{Spec} \\ \mathit{Circ}_2\ \mathsf{Sat}\ \mathit{Spec} \Longrightarrow (\mathit{Circ}_1\ \mathsf{Join}\ \mathit{Circ}_2)\ \mathsf{Sat}\ \mathit{Spec} \end{array}$$

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: 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

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

This is not quite adequate though to simulate Formal Synthesis style proofs. A small problem is that the application of the techniques Reveal, Inside, and Subdesign extend the VERITAS signature. As there is no direct analog of dynamic signature extension in ISABELLE, we model Reveal and Inside using bounded quantification and achieve an effect similar to signature extension by adding declaration information to the assumptions of the proof state according to the rules of set theory. Subdesign, which in VERITAS extends the signature with a new definition is slightly trickier to model; 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 a theorem that we prove takes the initial form

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

Analogous to the Formal Synthesis design goals we call such a theorem a *design theorem*. The constant Def is simply the identity function and $\mathsf{Def}(?\mathit{Definitions})$ serves as a kind of 'definition context'. 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 voltage theory 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

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

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

4.2 Reinterpreting Techniques

We have implemented tactics that simulate the first seven techniques (for the Proof technique we simply call the prover). Table 2 lists the tactics and the derived rules corresponding to the VERITAS techniques. The tactics are based

Claim: `claim_tac` *str i*

$[\![\ \mathit{Circ}\ \mathsf{Sat}\ \mathit{Spec}';\ \mathit{Spec}' \rightarrow \mathit{Spec}\ ]\!] \Longrightarrow \mathit{Circ}\ \mathsf{Sat}\ \mathit{Spec}$

Split: `split_tac` *i*

$[\![\ \mathit{Circ}_1\ \mathsf{Sat}\ \mathit{Spec}_1;\ \mathit{Circ}_2\ \mathsf{Sat}\ \mathit{Spec}_2\ ]\!] \Longrightarrow (\mathit{Circ}_1\ \mathsf{Join}\ \mathit{Circ}_2)\ \mathsf{Sat}\ (\mathit{Spec}_1 \wedge \mathit{Spec}_2)$

Reveal: `reveal_tac` *i*

$[\![\ !!\, w.\ w{:}T \Longrightarrow \mathit{Circ}(w)\ \mathsf{Sat}\ \mathit{Spec}(w)\ ]\!] \Longrightarrow (\mathsf{Wire}\, w{:}T.\ \mathit{Circ}(w))\ \mathsf{Sat}\ (\exists w{:}T.\mathit{Spec}(w))$

Inside: `inside_tac` *i*

$[\![\ !!\, p.\ p{:}T \Longrightarrow \mathit{Circ}(p)\ \mathsf{Sat}\ \mathit{Spec}(p)\ ]\!] \Longrightarrow \mathsf{Port}\, p{:}T.\ \mathit{Circ}(p)\ \mathsf{Sat}\ \mathit{Spec}(p)$

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*

$[\![\mathsf{Def}(\mathsf{Port}\, p_1 : P_1\ \ldots\ p_n : P_n.\ \mathit{Name}(p_1, \ldots, p_n) = \mathit{Circ}'(p_1, \ldots, p_n));$
$\mathsf{Port}\, p_1 : P_1\ \ldots\ p_n : P_n.\ \mathit{Circ}'(p_1, \ldots, p_n)\ \mathsf{Sat}\ \mathit{Spec}'(p_1, \ldots, p_n);$
$\mathsf{Port}\, p_1 : P_1\ \ldots\ p_n : P_n.\ \mathit{Name}(p_1, \ldots, p_n)\ \mathsf{Sat}\ \mathit{Spec}'(p_1, \ldots, p_n) \Longrightarrow \mathit{Circ}\ \mathsf{Sat}\ \mathit{Spec}$
$]\!] \Longrightarrow \mathit{Circ}\ \mathsf{Sat}\ \mathit{Spec}$

Design: `design_tac` *dels elims i*

$[\![\ \mathsf{Port}\, x : T.\ P(x);\ P(x) \Longrightarrow R;\ x{:}T\ ]\!] \Longrightarrow R$

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

on the derived rules, which correspond to the validation rule associated with the VERITAS technique; their function is mostly self explanatory.

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 and 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 design, supplied by the user. The rule is a form of implication elimination. The first subgoal is instantiated with the component's design theorem. The second is solved by extending the definition context of the overall design. 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. The third allows us to use the subdesign when proving the original subgoal.

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. 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 Extensions of the Calculus

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 is used to build general recursively defined designs. We will consider course-of-values induction below and later apply it 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 programs [1]. There we developed rules and tactics based on induction which 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 an inductively defined datatype like the natural numbers, e.g.,

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

specifies an implementation whose size depends on the number n. We use induction to instantiate *Circ* 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-value 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 the following rule schema were applied.

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

When this rule is applied by higher-order resolution, *Spec* will unify with the specification of the design theorem, and *Circ* with the metavariable standing for the circuit. The first subgoal sets up a parameterized circuit definition: an equality for *Circ* 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 *Circ*. Moreover, we now have a new assumption (the induction hypothesis),which states that *Circ* 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 *Circ* with an instance of *Circ*. How this works in practice will become clearer in Section 5.2.

Parameterized specifications require parameterized input 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, but this allows us to directly use ISABELLE's list theory). Some of our definitions are as follows.

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

A bus (B 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 a bus b and $l`n$ returns the nth element of a bus. We have proven many standard facts about these definitions, e.g., we can decompose busses.

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

5 Examples

We present two examples. Our first example is a comparator developed using Formal Synthesis in [7]. 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.

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 (left-hand figure in Figure 1) compare a bit from A and B and also have as input three bits (one for each case) which are the result of previous comparisons (*grin*, *eqin*, and *lsin*), and as output three 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)
```

The function 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.
    ?Circ(a, b, grin, eqin, lsin, grout, eqout, lsout) Sat
    CompCellS(a, b, grin, eqin, lsin, grout, eqout, lsout))
```

We apply 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; this yields the following proof state (we elide some information about the typing of the ports).

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

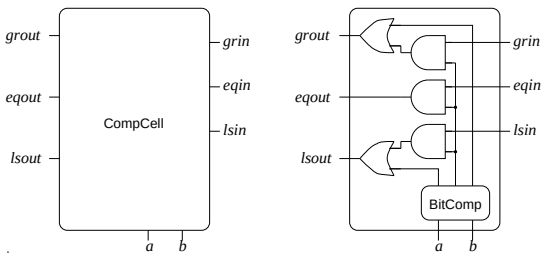

Fig. 1. Comp-cell and its claimed implementation

The original goal is given on the first-three lines; it contains two metavariables ?*Circ* 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` 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 which 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.
    ?Circ(a, b, grin, eqin, lsin, grout, eqout, 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)

  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) |] ==>
     ?Circ(a, b, grin, eqin, lsin, grout, eqout, lsout) Sat
     CompCellS(a, b, grin, eqin, lsin, grout, eqout, lsout)
```

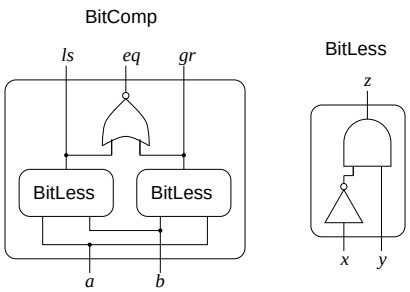

Fig. 2. Claimed implementations of bit-comp and bit-less

Given this subdesign we use `claim_tac` to state that the following specification entails the original goal (Figure 1).

```
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)
```

Due to space limitations we will just sketch the remaining proof. First we show that the claimed specification entails the original one; we prove this automatically using a tactic that performs case-analysis and simplification. After, we implement it by using `reveal_tac` to strip existential quantifications (and introduce internal wires) and then use `split_tac` to break up the conjunctions (and Join together subcircuits). The components are each implemented either by introducing simpler subdesigns and implementing those (see below), or using `library_tac`, which accesses appropriate library parts, and using `design_tac`, to apply developed subdesigns.

Let us sketch one of these remaining tasks: implementing the subdesign BitCompS. We proceed in the same manner as earlier and we introduce a new subdesign *BitLess*; then we build the above *BitComp* using *BitLess* twice as shown in Figure 2. Finally, BitLessS is so simple that we can claim a direct implementation consisting of components from the library. 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 $?Circ$ 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 [7]. The difference 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, 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 (c.f. [9]). 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 $g =$ Hi or if both $p =$ Hi and $c_i =$ Hi. A carry lookahead adder is implemented, roughly speaking, by recursively decomposing it in 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 3). 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)
```

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))
```

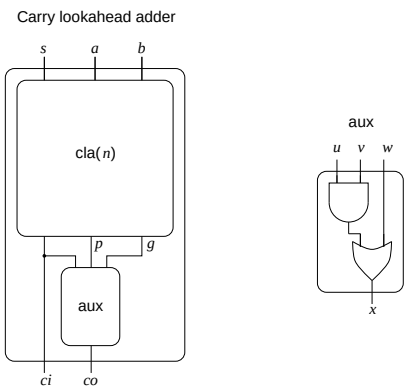

Fig. 3. Implementation of a cla-adder from two components

As before, we begin by shifting the definition environment $\mathsf{Def}(?H)$ to the assumptions. After, we apply the course-of-values induction tactic 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 may assume that we have an implementation for all k less than n and that 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

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

where P is instantiated with $n = 1$. 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) -->
```

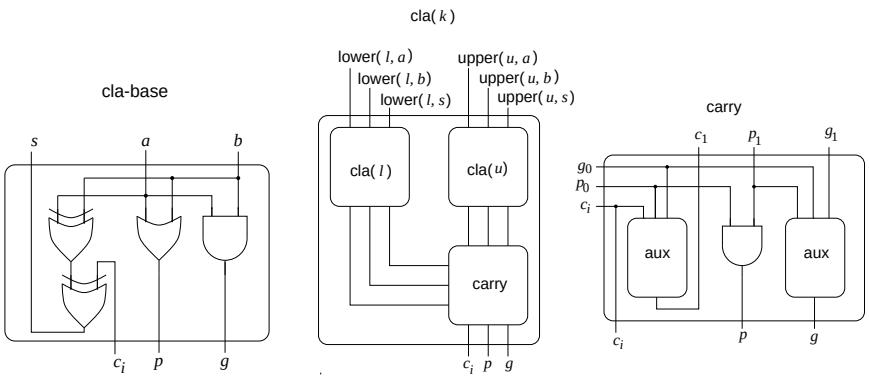

Fig. 4. Base case (left) and recursive decomposition (middle/right) where $l = k \operatorname{div} 2$ and $u = k \operatorname{div} 2 + k \bmod 2$

```
(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);
          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 4) in a few simple steps.

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 $\mathsf{lower}(n \operatorname{div} 2, _)$ containing the inferior $n \operatorname{div} 2$ bits of the bus and a segment $\mathsf{upper}(n \operatorname{div} 2 + n \bmod 2, _)$, containing the remaining bits. The lower (upper) segments of a and b can be added by an adder of bit-width $n \operatorname{div} 2$ ($n \operatorname{div} 2 + n \bmod 2$) 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.

```
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.

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. The remaining specification carryS is solved by a new subdesign, which is implemented as shown in Figure 4. Note that we reuse the formerly developed *aux* here. Thus, all design goals are solved and after 39 steps we are finished.

```
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 3 and 4 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 a series of extensions to higher-order logic and we were able to directly utilize standard Isabelle theories in our

work as well as Isabelle's simplification tactics. Most of our derived rules were proven in one step proofs by ISABELLE's classical prover.

The idea of using first-order resolution to build programs goes back to Green in the 1960s [6]. More recently, within systems like ISABELLE, interactive proof by higher-order resolution has been used to construct verified programs and hardware designs [1, 4, 10]. 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 [5]. 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;[1] 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 their methodology within our setting. We have carried out some initial experiments which 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. ?Circ(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.

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

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

[1] In the end, there are only so many (700?) proof development systems, but there are many more strategies for constructing proofs.

circuit built for $Spec(s)$, 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 [2]. We hope that this is a step towards a synthesis framework in which different verification methodologies may be integrated.

References

1. David A. 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.
2. David A. Basin and Nils Klarlund. Hardware verification using monadic second-order logic. In *Computer-Aided Verification (CAV '95)*, volume 939 of *LNCS*, pages 31–41. Springer-Verlag, 1995.
3. 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.
4. Martin David Coen. Interactive program derivation. Technical Report 272, Cambridge University Computer Laboratory, Cambridge, November 1992.
5. S. Finn, M. P. Fourman, M. Francis, and R. Harris. Formal system design — interactive synthesis based on computer-assisted formal reasoning. In Dr. Luc 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).
6. Cordell Green. Application of theorem proving to problem solving. In *Proceedings of the IJCAI-69*, pages 219–239, 1969.
7. F. Keith Hanna, Neil Daeche, and Mark 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.
8. F.K. 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.
9. John L. Hennessy and David A. Patterson. *Computer Architecture, a Quanitative Approach*. Morgan Kaufmann, 1990.
10. Lawrence C. Paulson. *Isabelle : a generic theorem prover; with contributions by Tobias Nipkow*, volume 828 of *LNCS*. Springer, Berlin, 1994.