Relational STE and Theorem Proving for Formal Verification of Industrial Circuit Designs

John O'Leary and Roope Kaivola
Intel Corporation
{john.w.oleary, roope.k.kaivola}@intel.com

Tom Melham
University of Oxford
Tom.Melham@cs.ox.ac.uk

***Abstract*—Model checking by symbolic trajectory evaluation, orchestrated in a flexible functional-programming framework, is a well-established technology for correctness verification of industrial-scale circuit designs. Most verifications in this domain require decomposition into subproblems that symbolic trajectory evaluation can handle, and deductive theorem proving has long been proposed as a complement to symbolic trajectory evaluation to enable such compositional reasoning. This paper describes an approach to verification by symbolic simulation, called *Relational STE*, that raises verification properties to the purely logical level suitable for compositional reasoning in a theorem prover. We also introduce a new deductive theorem prover, called *Goaled*, that has been integrated into Intel's Forte verification framework for this purpose. We illustrate the effectiveness of this combination of technologies by describing a general framework, accessible to non-experts, that is widely used for verification and regression validation of integer multipliers at Intel.**

I. Introduction and Motivation

Forte [1] is a formal verification environment, based on symbolic circuit simulation, that is well-established as an effective solution to large-scale, datapath correctness verification at Intel Corporation [2], [3], [4], [5], [6]. Two prominent successes are the verification of the entire execution cluster of the Intel Core 2 Duo [7] and Core i7 processors [8]. Some challenging control-dominated designs have also been verified [9].

The foundation for verification of circuit properties in Forte is *symbolic trajectory evaluation* [10]. Symbolic trajectory evaluation (STE) is a model-checking method powered by symbolic circuit simulation: it computes expressions for circuit outputs in terms of variables that stand for inputs, and checks that the circuit behaviours obtained satisfy temporal logic formulas, computing the exact region of any disagreement. These features give a seamless connection between simulation and verification, as well as comprehensive feedback on failed properties—two key elements of an effective methodology for large-scale formal verification [1], [11].

To control complexity, STE adds a flexible mechanism for partitioned abstraction [12], [13]. But, like any model checker, STE still has limited capacity. Forte therefore complements model checking with a higher-order logic theorem prover of similar design to the HOL system [14]. Theorem proving bridges the gap between big, industrially-important verification tasks and tractable model checking problems. At Intel, Forte is commonly used to provide assurance of functional correctness—rather than 'bug hunting'. (Other tools are used for assertion-based verification.) The verifications tackled are therefore large and highly complex, and almost always require some form of problem decomposition into tractable model-checking cases. A typical high-level correctness statement may decompose in complex ways into tens or even hundreds of individual STE properties. Theorem proving helps assure the verification engineer that these do indeed join up to imply the overall correctness result. Problem decompositions also commonly spin out side conditions that can't be checked by STE itself, but which yield to validation by theorem proving.

The Forte approach is to integrate model checking and theorem proving within the single framework of a functional programming language and its runtime system. A highly engineered implementation of STE is built into the core of the language, with numerous entry points into the internals of the model-checking algorithm provided as user-visible functions. Classically, verification in Forte has been viewed as *programming* activity, in which the functional language is used directly by verification engineers to orchestrate proofs and customize the tools to meet complex verification challenges [1].

More recently, however, the focus at Intel has been shifting to a higher level approach, enabled by two key technical developments: a new theorem prover called *Goaled* and a higher level formulation of property verification by symbolic simulation called *Relational STE*. Goaled is a complete replacement for Forte's original theorem prover, ThmTac [1], [15]. Goaled has a much more complete logical basis than ThmTac and is fully integrated with *reFLect* [16], a principled redesign and implementation of the system's original functional programming language. Relational STE liberates the user from the low-level temporal logic of primitive STE. It allows properties to be expressed in terms of purely logical constraints, which are suitable for compositional reasoning with the Goaled theorem prover.

In this paper, we give the first detailed account of Goaled and Relational STE, and of the higher level approach to verification enabled by these technical developments. We illustrate the approach by describing a general framework for integer multiplier verification that puts the power of Forte into the hands of non-experts, and which is widely used for verification and regression validation of multipliers at Intel.

Intel's deployments of STE and Forte are among the most substantial and sustained formal verification efforts in industry; and, for some time, they were distinctive in general approach. It is therefore encouraging to see the emergence of some

ISBN 978-0-9835678-3-7/13. Copyright owned jointly by the authors and FMCAD Inc.

impressive results obtained at Centaur Technology [17], [18] using a framework, based around the ACL2 theorem prover, that has many parallels with Forte—as well as some significant differences. We discuss this work in some detail in Section V.

II. The Integrated Goaled Theorem Prover

Theorem proving as a complement to STE model-checking has a long history. The combination was pioneered in the early 1990s in an academic predecessor of Forte called Voss [19]; this was followed by a series of systems that linked STE and theorem proving, culminating in today's mature integration within Forte of the comparatively full-featured theorem prover Goaled—an integrated combination that is seeing increasing use in production verification projects.

HOL-Voss was a mathematically-principled hybrid of symbolic trajectory evaluation and reasoning in the HOL theorem prover [20], [21]. Formal definitions were made in HOL of the mathematical entities of trajectory evaluation and some of the functional programming constructs used in Voss to specify data operations and circuit properties assertions. STE proofs done by Voss could then generate HOL theorems framed in this theory, and one could use HOL to construct higher-level arguments from these. The circuit model remained external to HOL and was represented by an uninterpreted logical constant.

At around the same time, Hazelhurst and Seger developed a simple theorem prover within Voss itself for reasoning in a sound and complete system of inference rules for the temporal logic of STE [22]. The prover was augmented with some ad-hoc 'domain knowledge', such as algebraic rules of multiplication, and had some automatic proof heuristics. The idea was to integrate—in a single framework this time—model checking by symbolic simulation and deductive reasoning about 'deeply embedded' assertions of the STE logic.

A step-change in integration came with the realisation that one might unify the functional *programming* language for scripting STE and the *logical* language within which reasoning is done.[1] Lifted-FL [15] was a deep embedding of the underlying term structure of Voss's functional programming language, called 'FL', within itself. This term structure is essentially the typed λ-calculus, the same as that of the higher-order logic in a theorem prover, such as HOL, that uses Church's formulation of the logic [24]. The syntactic *theorems* of a theorem prover implemented within Voss could now just be quoted fragments of FL—i.e. functional program expressions, of Boolean type, that are manipulated as *data* within the system. Moreover, since logical formulas were now just program expressions, FL's native evaluator could (sometimes) be used to prove them. This gave very fast *proof by evaluation*, as a complement to more laborious deductive inference.

Two generations of a theorem prover called ThmTac [1], [15] were built around this idea and used with STE to verify several challenging industrial circuit designs for which decomposition was essential [1], [25], [26]. This strongly validated the general approach, but the system still had some shortcomings. The logical and programming languages didn't completely align—the most notable gap being pattern matching, which was compiled away in the process of parsing quoted FL expressions. The correct logical treatment of type definitions and recursive function definitions was largely passed over. The core of the theorem prover consisted of a collection of 'trusted tactics', so the logical basis was ad-hoc. More seriously, this meant that theorem proving was biased towards interactive, goal-directed proof [27], limiting its appeal to non-experts. Finally, reasoning was still focussed on circuit properties expressed in the primitive temporal logical of STE.

The next step was an extensive 'rational reconstruction' of the programming language, FL. This was aimed, among other things, at making the connection between the programming and logical language much more principled. It also enabled a host of engineering improvements to the system. The result was *reFLect* [16], the language at the heart of the version of Voss—by now renamed Forte—used today in production formal verification projects at Intel.

ReFLect was designed from the start with theorem proving in mind. The aim was to have precisely the *same* λ-calculus at the core of both the logic and the programming language, and for the theorem prover and the *reFLect* interpreter to use identical internal data structures. This allows flexible (but, for logical soundness, still carefully regulated) intermixture of deduction in the theorem prover and evaluation in the interpreter, including proof by evaluation. To gain confidence in the soundness of this integration, a rather intricate reduction semantics for the language was designed, the rules of which could then be made primitive inference rules of the theorem prover. One innovation was the introduction of function definitions by pattern matching over quoted code, and a major challenge was formulating the right reduction rules for this.

Goaled is a full-featured, but still lightweight, higher order logic theorem prover built in *reFLect* for reasoning about *reFLect* programs. The system is heavily influenced by HOL and HOL Light [28]; it can be seen as a reimplementation of these, but with a radically extended λ-calculus. Following the LCF paradigm, it is built on a trusted core of primitive inference rules, expressed in terms of new formulations of the usual *pre-logic* primitives of α-equivalence, term matching, substitution, and so on. The core includes a full basis for ordinary higher-order logic, full rules for term reduction—including pattern matching over quoted terms—and certain reflection rules for moving between logic and native evaluation.

Function definitions and type definitions in the logic, including quotient types, originate as programming language definitions made at the *reFLect* interpreter level. But, although arbitrary definitions are permitted in the interpreter, definitions are made visible in the logic only after surviving scrutiny. This design choice is motivated by the reality of how *reFLect* is used in practice. *ReFLect* is used for hardware specification and proof scripting, where formal proof is valuable or even essential—but also for tool implementation and general purpose programming, where proof is optional, if it is possible

[1] This idea had long been predated, of course, by the pioneering ACL2 theorem prover [23], which uses Applicative Common Lisp for both the implementation and the logical languages.

at all. Quotient types are defined in *reFLect* by proposing an equality-testing function, and are made visible in the logic upon proof that the equality-testing function is an equivalence relation. Functions that operate on quotient types are made visible in the logic upon proof that they respect equality for that type. Recursive function definitions that pass a syntactic test for primitive recursion are admitted immediately, but in general termination must be proved by exhibiting a well founded relation R and supplying a proof that the arguments to each recursive call decrease with respect to R.

Above this fundamental level, Goaled includes formalized theories of Booleans, the option type, natural numbers, integers, rationals, functions, pairs, and lists. There are also more hardware-oriented theories of fixed-width and variable-length bitvector arithmetic. Proof automation, largely ported from HOL, includes full-featured rewriting and simplification, a meson first order solver, and a Fourier-Motzkin solver for linear arithmetic over $\mathbb{N}$, $\mathbb{Z}$, and $\mathbb{Q}$. Following common practice in this domain, Goaled has sequent 'tagging' to enable integration with external decision procedures.

All these capabilities were added to Goaled in response to practical reasoning needs. In particular, they support the features of *reFLect* commonly used in verification practice, including overloading, records and quotient types. For the time being, we have found this to be adequate for our domain and a good characterization of 'lightweight' theorem proving in this setting. We needed much more than originally envisaged when, say, ThmTac was introduced. But it is still much less than what mainstream theorem provers such as HOL, Coq [29], or Isabelle/HOL [30] have. This is natural, because the activity we support is more about reasoning about functional programs than, say, doing proofs in algebraic mathematics or reasoning about inductively-defined discrete structures.

III. Relational STE: from Simulation to Logic

Symbolic trajectory evaluation (STE) is a model-checking algorithm that proves properties of circuit behaviour using ternary symbolic circuit simulation. The STE algorithm takes as inputs a circuit and a pair of *trajectory formulas*, called the *antecedent* and *consequent*, that together constitute the property to be checked. Roughly speaking, the intuition is that the antecedent determines certain bits in the initial state and provides stimuli to selected circuit inputs at certain points within a bounded period of time. The consequent specifies the values expected to appear on selected circuit nodes as a response, while the circuit model is simulated.

A successful run of STE establishes a *trajectory assertion* saying that any execution of the circuit that conforms to the antecedent also satisfies the consequent. In essence, STE checks that circuit behaviour has simple stimulus-response properties, framed within a finite window of time. In addition, there is a mechanism for abstraction of circuit behaviour in which circuit nodes can carry 'unknown' values. This is overlaid by a symbolic representation for groups of properties that allows relationships between values on different circuit nodes to be expressed. Together, these provide a means by which families of abstractions, each covering only part of the circuit's behaviour, can be checked simultaneously. This mechanism for partitioned abstraction is called *symbolic indexing* [12] and can sometimes achieve dramatic efficiency gains [13], [31].

In the classical formulation of STE, the antecedent and consequent are written in a very simple linear-time temporal logic. In Forte, these formulas are represented concretely by lists of 5-tuples of the form

$$(guard, node, value, start, end)$$

where $guard$ and $value$ are formulas of propositional logic (usually BDDs, but a non-canonical representation aimed at SAT is supported too), $node$ is a node name (a string), and $start$ and end are non-negative integers. The meaning is that if $guard$ holds, then $node$ has $value$ from simulation cycle $start$ up to but excluding simulation cycle end.

ThmTac and earlier theorem provers for STE provided a specialised deductive system for compositional reasoning about the trajectory assertions in the form just introduced. In essence, lists of 5-tuples constituted a 'deep embedding' of a syntax of trajectory assertions; and rules were added to higher order logic that constituted an axiomatic theory of the embedded STE simulation logic. Consequently, the deductive system for circuit properties was not well integrated with the ordinary higher order logic of these theorem provers.[2]

A. Circuit Execution Semantics and STE Formulas

Let us introduce some basic definitions. A *circuit* is a well-formed interconnection of combinational gates and sequential elements, such as flip-flops and latches. An *execution* is a function of type $(\mathit{string} \times \mathit{num}) \rightarrow \mathit{bool}$ that assigns to each circuit node, named by the string, a Boolean value at each point in time, represented by a natural number. The *behaviour* of a circuit is a predicate on executions—or, equivalently, a set of executions.

We shall assume the existence of a function

$$[\![ckt]\!] :: ((\mathit{string} \times \mathit{num}) \rightarrow \mathit{bool}) \rightarrow \mathit{bool}$$

that gives us the behaviour of a circuit, i.e. a predicate determining whether a given execution e is consistent with the circuit. Analogously, we also write $[\![ant]\!]$ for a predicate specifying whether an execution e is consistent with the five-tuple list ant. We are deliberately vague about how a circuit is represented concretely, but note that if a representation is chosen it is a relatively simple matter to define the function $[\![_]\!]$ mathematically. In essence, one would use the classical 'relational' approach that is well-known from hardware modelling in higher-order logic [32].

B. Relational STE

STE has proven to be an extremely useful verification engine in practice, and has been the key enabler for most of Intel's formal verification success stories. Nevertheless, the

[2]Some bridges between the two levels were, however, provided by certain quantifier rules and axioms about 'parametric' encoding of assumptions. See Section VII of [1] for details.

language of trajectory assertions severely limits the classes of properties that can be expressed natively. In effect, trajectory assertions require a specification to be functional: given inputs, the specification dictates the values the outputs shall have, potentially under some do-care conditions.

Many informal specifications occurring in practice do not fall into this category, the simplest one being 'nodes a and b are mutually exclusive'. Such relational specifications become especially important as the abstraction level of specifications rises. For example, a natural specification of a scheduler might say 'an operation with its sources ready will be scheduled for execution', while intentionally leaving open the selection between different ready operations. When verifying relational specifications with STE, the established practice has been to use STE as a symbolic simulation engine only, and to write ad hoc FL code to compute the satisfaction of the specification on the basis of simulation values queried from the STE trace.

Relational STE, or rSTE in short, has been crafted as a systematic solution to the problem of expressing and verifying general relational specifications, while retaining the use of STE as the underlying symbolic simulation engine.

In what follows, we give an overview of the technical underpinnings of rSTE. The notation we use is an idealization of the Goaled higher order logic. As discussed in Section II, phrases of this logic are simultaneously *reFLect* programs, so we shall employ a mixture of functional programming and logical notation. The reader is spared the concrete syntax of the actual Forte implementation, which for historical reasons is similar to that of the original 'Edinburgh' ML [33].

The basic building blocks for rSTE specifications are called *constraints*. Conceptually, a constraint is simply a predicate on circuit executions. Technically, a constraint c consists of three parts: name, predicate and signature, denoted by $\mathtt{name}\,c$, $\mathtt{pred}\,c$ and $\mathtt{sig}\,c$, respectively. The name is simply a string that is used to identify the constraint for user convenience, e.g for informational messages. The predicate is a function

$$\mathtt{pred}\,c :: ((\mathit{string} \times \mathit{num}) \to \mathit{bool}) \to \mathit{bool}$$

and the signature is a list of $\mathit{string} \times \mathit{num}$ pairs. The predicate refers to a finite collection of individually specified circuit nodes and points of time, conceptually querying their values in a circuit execution given to the predicate as an argument, and computes a Boolean function of these values. The signature lists all the nodes referred to by the predicate, and the times at which their value is accessed.

For example, the constraint shown below could be used to express the informal property that 'circuit nodes a and b are mutually exclusive at time point 2'.

$$\begin{array}{l} \mathtt{CONSTR}\ \text{“ab_mutex”} \\ \qquad (\lambda e. \neg((e(\mathsf{a},2)) \wedge (e(\mathsf{b},2)))) \\ \qquad [(\mathsf{a},2),(\mathsf{b},2)] \end{array}$$

We extend the predicate function pred to a constraint list $cl = [c_1, \ldots, c_n]$, implicitly considered conjuncted, by:

$$\mathtt{predl}\ cl\ e \ \stackrel{\text{def}}{=}\ (\mathtt{pred}\,c_1\ e) \wedge \ldots \wedge (\mathtt{pred}\,c_n\ e)$$

The user interface to rSTE is through a *reFLect* function

$$\mathtt{rSTE}\ \mathit{ckt}\ \mathit{antc}\ \mathit{antv}\ \mathit{cin}\ \mathit{cout}\ \mathit{opts}$$

The first two arguments to rSTE, the circuit ckt and the constant antecedent $antc$, describe the 'static' aspects of the verification task. Here $antc$ is a five-tuple list, exactly as used in classical STE; but it is used in rSTE only to set constant value assignments, e.g. clock patterns, testability signals, etc., that are shared by all verification tasks on the circuit. In classical STE, the antecedent is also used to assign symbolic variables to circuit nodes; in rSTE this aspect is separated out as a distinct variable binding antecedent $antv$, which allows the user only to bind positive instances of distinct symbolic variables to circuit nodes. Every circuit node is mentioned in $antc$ or $antv$ or is assigned an unknown 'X' value at the start of the simulation, so this covers the full state-space. For more discussion on symbolic variable bindings, see [34].

The main logical content of an rSTE verification task is described with the input and output constraint lists cin and $cout$. The meaning is that if the input constraints cin hold, then circuit behaviour under simulation will satisfy the output constraints $cout$. In theory, the constraints could be combined into an implication $cin \Rightarrow cout$. In practice, however, verification of each element in $cout$ may be carried out separately to alleviate complexity. Separating out the constraints cin can also allow some them to be injected into the symbolic simulation using parametric substitution [26], futher improving efficiency.

Finally the options list $opts$ gives the user fine-grained control over how rSTE carries out the verification task. For example, the computation may require use of a parametric representation of boolean functions [26], a specific circuit abstraction method, or certain simulation or constraint satisfaction engines: e.g. BDD-based analysis vs SAT-based analysis. The user specifies all such choices through the rSTE options.

Internally, the *reFLect* function rSTE uses the constant and variable binding antecedents $antv$ and $antc$ as well the user-given options $opts$ to construct a classical STE antecedent. It then carries out symbolic simulation with STE using this antecedent as stimulus. Following the symbolic simulation, an execution $\hat{e}$ is available that 'reads off' symbolic values on circuit nodes at any specified times. To establish truth or falsehood of the rSTE assertion, the implication

$$(\mathtt{predl}\ \mathit{cin}\ \hat{e}) \implies (\mathtt{predl}\ \mathit{cout}\ \hat{e})$$

is evaluated over $\hat{e}$, either using BDDs or a SAT solver, depending on user options.

The relational formulation of symbolic trajectory evaluation preserves the power of the underlying STE algorithm while enabling much richer specifications—in particular ones describing relations between nodes values rather than just functions. Since its introduction, rSTE has been the workhorse of datapath verification at Intel; many thousands of individual operations, from the very simple to the very complex, have been verified in several microprocessor families and over the course of several generations.

In the classical theory of STE, the 'fundamental theorem' relates a successful run of the symbolic simulator to the logical property it establishes. For rSTE, the fundamental theorem has a particularly elegant formulation:

$$\begin{array}{l} \forall ckt\ antc\ antv\ cin\ cout\ opts\,. \\ \quad \texttt{rSTE}\ ckt\ antc\ antv\ cin\ cout\ opts \Longrightarrow \\ \quad\quad \forall e. [\![ckt]\!]\ e \wedge [\![antc]\!]\ e \Longrightarrow \\ \quad\quad\quad (\texttt{predl}\ cin\ e) \Longrightarrow \\ \quad\quad\quad (\texttt{predl}\ cout\ e) \end{array}$$

Most important for the purpose of this paper, the relational formulation eliminates the need to use specialized STE inference rules and apparatus for temporal reasoning. The relational formulation makes it 'just' higher order logic. Reasoning about functional and temporal aspects of circuit behavior takes place in a uniform framework: higher order logic. Indeed, rSTE itself is a function that can be reasoned about in higher order logic. In Section IV, we illustrate a practical application of this: reasoning about automatically generated specifications and automatically generated calls to rSTE.

Notice that the symbolic variables bound by *antv* do not appear in the correctness property inferrable from a successful run of rSTE. This is intentional: we use symbolic simulation and computation only as a means to the end goal of verifying universally quantified claims. The identity of the symbolic variables and the precise bindings do not matter, as long as distinct variables are bound to distinct circuit nodes and times, which rSTE checks automatically.

IV. A Framework for Integer Multiplication

Figure 1 shows a Booth multiplier. Its principles of operation are simple. First, one of the operands, S1 say, is Booth encoded: N Booth coefficients $\texttt{BE}_i(\texttt{S1})$ are computed such that

$$-2^k < \texttt{BE}_i(\texttt{S1}) < 2^k.$$

for $0 \leq i < N$ and $k > 0$. A given multiplicand S1 has many valid Booth encodings, but the Booth coefficients are always required to satisfy

$$\texttt{S1} = \sum_{i=0}^{N-1} \texttt{BE}_i(\texttt{S1}) \times 2^{ki}. \tag{1}$$

The quantity 2^k is called the *radix*.

Second, a set of N *partial products* is computed, one for each Booth coefficient:

$$\texttt{PP}_i = \texttt{BE}_i(\texttt{S1}) \times \texttt{S2} \tag{2}$$

for $0 \leq i < N$.

Finally, the N partial products are shifted and summed to yield the product PROD:

$$\texttt{PROD} = \sum_{i=0}^{N-1} \texttt{PP}_i \times 2^{ki} \tag{3}$$

Automatic input-to-output verification of even a moderately-sized multiplier is beyond the capacity of the BDD- and SAT-based verification approaches commonly deployed in industry.

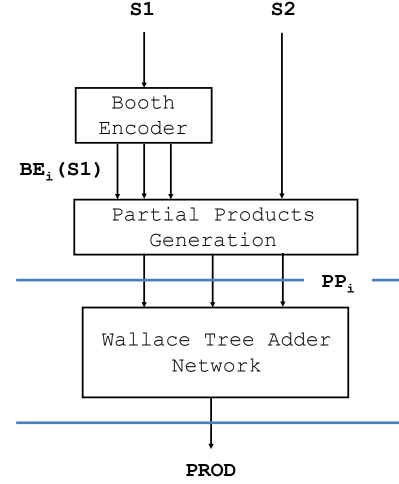

Fig. 1. A Booth Multiplier.

But verification of Equations (1), (2), and (3) is tractable and can be done automatically using rSTE. Once these are established it will require only straightforward algebraic reasoning to prove, on paper or in Goaled, that

$$\texttt{PROD} = \texttt{S1} \times \texttt{S2} \tag{4}$$

Consider now the task of verifying an actual circuit implementation of a Booth multiplier. Typically, a circuit expects to receive a valid clock pattern and the assertion of some interface control signals requesting the execution of a multiplication operation. In rSTE verification, we would fix a constant reference time for the start of the operation, and code the expected clock and control signal patterns by a constant antecedent antc. The circuit will read source data values on designated signals at some fixed delay after the start of the operation. Then, Booth encodings and partial products will be computed, and partial products summed together to produce a final product on designated result signals at some later time.

To map the conceptual proof stages above to an actual circuit implementation, we need to first identify the circuit signals for both sources S1 and S2, the partial products and the final product, with the appropriate timing relative to the fixed start of the operation, and code these as *string* × *num* lists s1, s2, $\texttt{pp}_i$ and prod, respectively. The function s2i interprets the values on such lists, relative to a given circuit execution, as integers using two's complement encoding.

We construct an rSTE constraint to check correctness of the Booth coefficients using Equation (1) as specification:

$$\begin{array}{lcl} \texttt{eqn1}(x) & \stackrel{\text{def}}{=} & x = \sum_{i=0}^{N-1} \texttt{BE}_i(x) \times 2^{ki} \\ \texttt{boothc} & \stackrel{\text{def}}{=} & \texttt{CONSTR}\ \text{“boothOK”} \\ & & \quad (\lambda e.\, \texttt{eqn1}\ (\texttt{s2i}\ e\ \texttt{s1})) \\ & & \quad \texttt{s1} \end{array}$$

As before, we use ordinary mathematical notation in definitions; *reFLect* notation differs in style but not in substance.

We now use rSTE to verify that the constraint boothc holds for all executions of the circuit. We will use the variable binding antecedent antv to assign distinct symbolic variables to all the source data signals at the time the circuit is expected to read their values, and execute the rSTE call

$$\texttt{rSTE ckt antc antv } [\,]\ [\texttt{boothc}]\ \texttt{opts}.$$

Success of this call allows us to conclude, via the fundamental theorem, that

$$\forall e.\ [\![\texttt{ckt}]\!]\ e \wedge [\![\texttt{antc}]\!]\ e \Longrightarrow (\texttt{predl}\ [\,]\ e) \Longrightarrow (\texttt{predl}\ [\texttt{boothc}]\ e)$$

By expanding the definition of predl and employing a few of Goaled's standard theorems about lists, this simplifies to

$$\forall e.\ [\![\texttt{ckt}]\!]\ e \wedge [\![\texttt{antc}]\!]\ e \Longrightarrow \texttt{pred boothc}\ e$$

Expanding the definitions of boothc and pred plus a little more rewriting yields

$$\forall e.\ [\![\texttt{ckt}]\!]\ e \wedge [\![\texttt{antc}]\!]\ e \Longrightarrow \texttt{eqn1}(\texttt{s2i}\ e\ \texttt{s1}).$$

Finally, expanding the definition of the auxiliary function eqn1 yields a theorem asserting that the Booth coefficients bear the correct relationship to the circuit nodes s1.

$$\forall e.\ [\![\texttt{ckt}]\!]\ e \wedge [\![\texttt{antc}]\!]\ e \Longrightarrow \texttt{s2i}\ e\ \texttt{s1} = \textstyle\sum_{i=0}^{N-1} \texttt{BE}_i(\texttt{s2i}\ e\ \texttt{s1}) \times 2^{ki}.$$

In similar fashion, we define constraints corresponding to Equations (2) and (3):

$$\begin{aligned}
\texttt{eqn2}_i(w_i, x, y) &\stackrel{\text{def}}{=} w_i = \texttt{BE}_i(x) \times y \\
\texttt{ppc}_i &\stackrel{\text{def}}{=} \texttt{CONSTR}\ \text{“ppiOK”} \\
&\qquad (\lambda e.\ \texttt{eqn2}(\ \texttt{s2i}\ e\ \texttt{pp}_i,\ \texttt{s2i}\ e\ \texttt{s1},\ \texttt{s2i}\ e\ \texttt{s2}\ )) \\
&\qquad (\texttt{s1 @ s2 @ pp}_i) \\
\texttt{eqn3}(z, w) &\stackrel{\text{def}}{=} z = \sum_{i=0}^{N-1} w_i \times 2^{ki} \\
\texttt{prodc} &\stackrel{\text{def}}{=} \texttt{CONSTR}\ \text{“prodOK”} \\
&\qquad (\lambda e.\ \texttt{eqn3}(\ \texttt{s2i}\ e\ \texttt{prod},\ \texttt{map}\ (\texttt{s2i}\ e)\ \texttt{pp}\ )) \\
&\qquad (\texttt{prod @ flat pp})
\end{aligned}$$

With these definitions in hand we execute the remaining rSTE runs. There are $N + 2$ in all, one for boothc, one instance of $\texttt{ppc}_i$ for each of the N partial products, and one final run to check prodc. For all except the final run we use the variable binding antecedent to assign variables to the source data signals; in the final run we use it to assign distinct symbolic variables to the partial product signals.

Using a BDD variable ordering that aligns the bits in partial products according to their position in the summation, we can handle verification of most Wallace tree adders occurring in Intel designs, up to extended precision floating point multipliers. For a minority of designs, a further cut-point in the middle of the Wallace tree is needed to manage BDD complexity, and Equation (3) is split into two obligations.

If all of these verification runs are successful, we can use the fundamental theorem of rSTE and simple rewriting and logical reasoning to conclude that

$$\begin{aligned}
&\forall e.\ [\![\texttt{ckt}]\!]\ e \wedge [\![\texttt{antc}]\!]\ e \Longrightarrow \\
&\quad (\texttt{s2i}\ e\ \texttt{s1} = \textstyle\sum_{i=0}^{N-1} \texttt{BE}_i(\texttt{s2i}\ e\ \texttt{s1}) \times 2^{ki}) \wedge \\
&\quad (\textstyle\bigwedge_{i=0}^{N-1} \texttt{s2i}\ e\ \texttt{pp}_i = \texttt{BE}_i(\texttt{s2i}\ e\ \texttt{s1}) \times \texttt{s2i}\ e\ \texttt{s2}) \wedge \\
&\quad (\texttt{s2i}\ e\ \texttt{prod} = \textstyle\sum_{i=0}^{N-1} (\texttt{s2i}\ e\ \texttt{pp}_i) \times 2^{ki})
\end{aligned}$$

From here, routine arithmetic reasoning yields a theorem asserting the correctness of the multiplier implementation:

$$\forall e.\ [\![\texttt{ckt}]\!]\ e \wedge [\![\texttt{antc}]\!]\ e \Longrightarrow (\texttt{s2i}\ e\ \texttt{prod}) = (\texttt{s2i}\ e\ \texttt{s1}) \times (\texttt{s2i}\ e\ \texttt{s2})$$

Note that these results were obtained using only standard logical reasoning, without the use of purpose-built inference rules for STE. This is due to the direct representation of rSTE constraints in the logic of Goaled, and our formulation of the fundamental theorem of rSTE that directly exposes the logical import of each successful rSTE run.

A. A General Framework for Multipliers

The method outlined above suffices to verify the correctness of one multiplier. Wide deployment across a large corporation presents additional challenges. Surface details like signal names and their timing vary from design to design, as do radix and operand widths. Designs differ more fundamentally in how they choose to encode Booth coefficients and partial products and how they represent flags and exceptional conditions. Many design-specific quirks are handled by customizing functions like s2i that extract values from the circuit trace, allowing us to view the circuit as if it was a vanilla design. If an implementation feature cannot be 'explained away' like this, the reference model is generalized to handle it. In addition, each *design organization* has its own validation and regression practices and software that supports them.

To address this diversity of designs and design environments, we have developed over the last decade a general framework for multiplier verification. The notion of constraints is generalized to allow predicates over arbitrary domains, with the constraints over circuit executions described in Section III being a special case. Abstraction mappings between constraints are used to separate the essence of the specifications and proofs shown above from accidental details of particular designs. The framework is designed to be configured and used by non-experts, who are responsible for supplying design details (signal names, representation of partial products, and so on) and setting configuration parameters (for example, radix and operand width). Behind the scenes, a sophisticated set of *reFLect* scripts arranges for design-specific specifications to be generated and orchestrates the necessary runs of rSTE.

Although the scripts are written with care, they are under continual refinement—each new design and design environment introduces some wrinkle—making it impossible to prove correctness of the scripts once and for all. There is a very real

risk that script errors will result in generation of incorrect specifications or an incomplete set of rSTE runs. To verify correct operation of the scripts, we have integrated Goaled with our multiplier verification framework. During nightly RTL regression, Goaled analyzes the scripts at source level to determine what specifications are generated and what rSTE runs are executed. A proof is programatically constructed, along the lines shown above, that these specs and rSTE runs are sufficient to ensure correctness of the circuit. This capability is currently deployed in a 'live' CPU design project.

Goaled also plays a more traditional role in our multiplier verification framework. Several side conditions arise in our compositional proofs that would be difficult or impossible to prove using BDDs or SAT. For example, this assumption is required in the proof of the Wallace tree adder:

$$\min\{\overline{\mathtt{S1}}\overline{\mathtt{S2}}, \overline{\mathtt{S1}}\underline{\mathtt{S2}}, \underline{\mathtt{S1}}\overline{\mathtt{S2}}, \underline{\mathtt{S1}}\underline{\mathtt{S2}}\} \leq \sum_{i=0}^{N-1} \mathtt{PP}_i \times 2^{ki}$$

where $\overline{x}$ ($\underline{x}$) denotes the largest (smallest) value of the bitvector x. A Goaled proof is immediate from Equations (1) and (2).

V. Related Work

Integration of model-checking and theorem proving was proposed as early as the mid 1990s [35], and many experiments in combining the techologies have been reported. One of these, ACL2SIX [36], integrates the ACL2 theorem prover and IBM's SixthSense verification tool. The combination was used to verify a pipelined 53×43 bit multiplier, by a decomposition strategy similar to the example presented in Section IV. Some more general verification frameworks that effectively combine two technologies have also been designed, two prominent examples being Prosper [37] and SAL [38].

The published research on applied formal verification most closely related to Forte is work on a verification toolflow used at Centaur Technology to help ensure correctness of their X86-compatible microprocessors [17], [18]. Developed and deployed by a team of engineers and scientists at Centaur and UT Austin, the framework integrates several reasoning tools and is based around the well-established ACL2 theorem prover [23]. As with Forte at Intel, a prominent application is the verification of floating-point and integer arithmetic hardware. Of course ACL2 itself, and its predecessor the Boyer-Moore theorem prover, have a long history of successful application to hardware verification [39], [40].

A notable feature of the Centaur framework is that it is built on top of publicly-available software tools: ACL2 itself and special-purpose tools such as the ZZ framework [41] and ABC [42]. The impressive verification results cited in [17], [18] and [43] show that a robust and practical toolflow of considerable capacity can be built in this way. By contrast, Forte is an in-house tool, highly engineered and optimised through years of use on challenging problems at Intel.

The two frameworks have many features in common, at least at a general level: symbolic circuit simulation is a central technology for generating circuit properties; individual properties of a proof are composed together in a theorem prover, to build up more complete verifications; and both systems are embedded in a general purpose programming language, so they can be extended and customised. Both tools can read and give semantics to large circuit models at either gate or transistor levels. Proof regression is a common verification activity carried out with both frameworks.

A significant difference between Forte and the Centaur framework is the depth of integration of circuit simulation. In Forte, the symbolic simulation algorithm is built in to *reFLect* and not exposed to reasoning at the level of the Goaled theorem prover. There are *reFLect* functions that can be used to explore and manipulate the structure of the circuit model, and access its state-transition semantics. But for efficiency the simulator itself is hard-coded into the internals of *reFLect*. Although the STE algorithm has been independently verified [44], there are no plans to verify the highly engineered Forte internal simulator. In the Centaur system based on ACL2, the symbolic circuit simulator is written in ACL2 itself. It is hence available as a formal object about which proofs can be done—and, indeed, has been verified correct [45].

The successful industrial deployment of two major verification frameworks, Forte at Intel and the ACL2-based tools at Centaur Technology, show that this idea has come of age industrially. Moreover the parallels between the two systems, each quite different from the other in numerous matters of detail, strengthens the conclusion that this kind of architecture represents a general solution in this important domain.

VI. Summary and Prospects

This paper has described what we hope to be the basis for a step-change in the exploitation of theorem proving as a complement to symbolic simulation for compositional verification of circuit designs. Relational STE raises the level of the properties obtained from symbolic trajectory evaluation to pure logic. Compositional reasoning can then be done straightforwardly in higher-order logic, rather than with specialised STE inference rules. A 'lightweight' theorem prover, Goaled, has been designed for this purpose and tightly integrated into Forte, the STE programming environment used at Intel. The utility of this approach is exemplified by a general tool for validation of integer multipliers that soundly automates complex proof decompositions for non-expert users, entirely 'hiding' the theorem proving support for this.

Future work on Goaled includes development of a theory of floating point operations at bit level. This is intended for certification of conformance to IEEE Standard 754 of 'reference models' of floating point algorithms expressed as *reFLect* programs. Valuable results of this kind been obtained in the past using ThmTac [1]; it is hoped that a capability for this in Goaled will aid in maintaining the certification of reference algorithms as they become more complex over time.

Future work on Relational STE includes fully integrating SAT-based symbolic trajectory evaluation into the framework, alongside BDDs. This is largely a matter of engineering. More challenging from a research perspective will be the incorporation of *symbolic indexing*, a flexible and somewhat

subtle mechanism for abstraction in STE, into the Relational STE flow. This may leverage past work on abstraction transformations [46] and automatic symbolic indexing [13].

REFERENCES

[1] C.-J. H. Seger, R. B. Jones, J. W. O'Leary, T. Melham, M. D. Aagaard, C. Barrett, and D. Syme, "An industrially effective environment for formal hardware verification," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 24, no. 9, pp. 1381–1405, Sept. 2005.

[2] J. O'Leary, X. Zhao, R. Gerth, and C.-J. H. Seger, "Formally verifying IEEE compliance of floating-point hardware," *Intel Technical Journal*, First quarter, 1999.

[3] R. Kaivola and K. R. Kohatsu, "Proof engineering in the large: formal verification of Pentium 4 floating-point divider," *Int. J. on Software Tools for Technology Transfer*, vol. 4, no. 3, pp. 323–334, 2003.

[4] R. Kaivola and N. Narasimhan, "Formal verification of the Pentium-4 multiplier," in *High-Level Design Validation and Test*. IEEE, 2001, pp. 115–122.

[5] A. Slobodová and K. Nagalla, "Formal verification of floating point multiply add on itanium processor," in *Fifth International Workshop on Designing Correct Circuits: Barcelona*. ETAPS 2004, Mar. 2004.

[6] A. Slobodová, "Formal verification of hardware support for advanced encryption standard," in *2008 Formal Methods in Computer Aided Design*. IEEE, 2008, pp. 1–4.

[7] A. Flaisher, A. Gluska, and E. Singerman, "Case study: Integrating FV and DV in the verification of the Intel Core 2 Duo microprocessor," in *Formal Methods in Computer Aided Design*. IEEE, 2007, pp. 192–195.

[8] R. Kaivola, R. Ghughal, N. Narasimhan, A. Telfer, J. Whittemore, S. Pandav, A. Slobodova, C. Taylor, V. Frolov, E. Reeber, and A. Naik, "Replacing testing with formal verification in Intel core i7 processor execution engine validation," in *Computer Aided Verification*, ser. LNCS. Springer-Verlag, 2009, pp. 414–429.

[9] R. Kaivola, "Formal verification of pentium® 4 components with symbolic simulation and inductive invariants," in *Computer Aided Verification*, ser. LNCS, K. Etessami and S. K. Rajamani, Eds., vol. 3576. Springer-Verlag, 2005, pp. 170–184.

[10] C.-J. H. Seger and R. E. Bryant, "Formal verification by symbolic evaluation of partially-ordered trajectories," *Formal Methods in System Design*, vol. 6, no. 2, pp. 147–189, Mar. 1995.

[11] R. B. Jones, J. W. O'Leary, C.-J. H. Seger, M. D. Aagaard, and T. F. Melham, "Practical formal verification in microprocessor design," *IEEE Design & Test of Computers*, vol. 18, no. 4, pp. 16–25, 2001.

[12] R. E. Bryant, D. L. Beatty, and C.-J. H. Seger, "Formal hardware verification by symbolic ternary trajectory evaluation," in *ACM/IEEE Design Automation Conference*. ACM Press, June 1991, pp. 397–402.

[13] S. Adams, M. Björk, T. Melham, and C.-J. Seger, "Automatic abstraction in symbolic trajectory evaluation," in *FMCAD*, 2007, pp. 127–135.

[14] M. J. C. Gordon and T. F. Melham, Eds., *Introduction to HOL: A theorem proving environment for higher order logic*. Cambridge University Press, 1993.

[15] M. D. Aagaard, R. B. Jones, and C.-J. H. Seger, "Lifted-FL: A pragmatic implementation of combined model checking and theorem proving," in *Theorem Proving in Higher Order Logics*, ser. LNCS, vol. 1690. Springer-Verlag, 1999, pp. 323–340.

[16] J. Grundy, T. Melham, and J. O'Leary, "A reflective functional language for hardware design and theorem proving," *Journal of Functional Programming*, vol. 16, no. 2, pp. 157–196, Mar. 2006.

[17] W. A. Hunt, Jr., S. Swords, J. Davis, and A. Slobodova, *Use of Formal Verification at Centaur Technology*. Springer-Verlag, 2010, pp. 65–88.

[18] A. Slobodová, J. Davis, S. Swords, and W. Hunt, "A flexible formal verification framework for industrial scale validation," in *9th IEEE/ACM International Conference on Formal Methods and Models for Codesign*. IEEE, 2011, pp. 89–97.

[19] C.-J. H. Seger, "Voss — a formal hardware verification system: User's guide," University of Brtish Columbia Department of Computer Science, Tech. Rep. TR-93-45, Dec. 1993.

[20] J. Joyce and C.-J. Seger, "Linking BDD-based symbolic evaluation to interactive theorem-proving," in *Design Automation Conference*, 1993, pp. 469–474.

[21] C.-J. Seger and J. Joyce, "A mathematically precise two-level formal hardware verification methodology," Department of Computer Science, University of British Columbia, Report 92-34, Dec. 1992.

[22] S. Hazelhurst and C.-J. Seger, "A simple theorem prover based on symbolic trajectory evaluation and BDD's," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 14, no. 4, pp. 413–422, 1995.

[23] M. Kaufmann and J. Moore, "An industrial strength theorem prover for a logic based on common lisp," *IEEE Transactions on Software Engineering*, vol. 23, no. 4, pp. 203–213, 1997.

[24] A. Church, "A formulation of the simple theory of types," *The Journal of Symbolic Logic*, vol. 5, pp. 56–68, 1940.

[25] M. D. Aagaard, R. B. Jones, and C.-J. H. Seger, "Combining theorem proving and trajectory evaluation in an industrial environment," in *ACM/IEEE Design Automation Conference*, 1998, pp. 538–541.

[26] ——, "Formal verification using parametric representations of boolean constraints," in *ACM/IEEE Design Automation Conference*, 1999, pp. 402–407.

[27] R. Milner, "The use of machines to assist in rigorous proof," in *Proc. of a discussion meeting of the Royal Society of London on Mathematical logic and programming languages*. Prentice-Hall, 1985, pp. 77–88.

[28] J. Harrison, "HOL light: A tutorial introduction," in *Proc. Formal Methods in Computer-Aided Design (FMCAD'96)*, ser. LNCS, M. Srivas and A. Camilleri, Eds., vol. 1166. Springer-Verlag, 1996, pp. 265–269.

[29] Coq Development Team, *The Coq Proof Assistant: Reference Manual, V8.4*. Inria, Aug. 2012.

[30] T. Nipkow, L. C. Paulson, and M. Wenzel, *Isabelle/HOL: A Proof Assistant for Higher-Order Logic*, ser. LNCS. Springer-Verlag, 2002, vol. 2283.

[31] M. Pandey, R. Raimi, R. E. Bryant, and M. S. Abadir, "Formal Verification of Content Addressable Memories using Symbolic Trajectory Evaluation," in *Design Automation Conference*. ACM Press, Jun. 1997, pp. 167–172.

[32] T. Melham, *Higher Order Logic and Hardware Verification*, ser. Cambridge Tracts in Theoretical Computer Science. Cambridge University Press, 1993, vol. 31.

[33] *Edinburgh LCF: A Mechanised Logic of Computation*, ser. LNCS. Springer-Verlag, 1979, vol. 78.

[34] Z. Khasidashvili, G. Gavrielov, and T. Melham, "Assume-guarantee validation for STE properties within an SVA environment," in *Formal Methods in Computer-Aided Design: FMCAD 2009*. IEEE, 2009, pp. 108–115.

[35] S. Rajan, N. Shankar, and M. Srivas, "An integration of model-checking with automated proof checking," in *Computer-Aided Verification*, ser. LNCS, vol. 939. Springer-Verlag, Jun. 1995, pp. 84–97.

[36] J. Sawada and E. Reeber, "ACL2SIX : A hint used to integrate a theorem prover and an automated verification tool," in *Proceedings of Formal Methods in Computer Aided Design: FMCAD 2006*. IEEE Computer Society, 2006, pp. 161–170.

[37] L. A. Dennis, G. Collins, M. Norrish, R. J. Boulton, K. Slind, and T. F. Melham, "The PROSPER toolkit," *Int. J. on Software Tools for Technology Transfer*, vol. 4, no. 2, pp. 189–210, Feb. 2003.

[38] N. Shankar, "Symbolic analysis of transition systems," in *Abstract State Machines: Theory and Applications*, ser. LNCS, no. 1912. Springer-Verlag, 2000, pp. 287–302.

[39] M. Kaufmann and J. S. Moore, *ACL2 and Its Applications to Digital System Verification*. Springer-Verlag, 2010, pp. 1–21.

[40] W. A. Hunt, Jr., "Microprocessor design verification," *Journal of Automated Reasoning*, vol. 5, no. 4, pp. 429–460, Dec. 1989.

[41] N. Een. ABC/ZZ. [Online]. Available: https://bitbucket.org/niklaseen/abc-zz

[42] Berkeley Logic Synthesis and Verification Group. ABC: A system for sequential synthesis and verification. [Online]. Available: http://www.eecs.berkeley.edu/ alanmi/abc/

[43] W. A. Hunt, Jr. and S. Swords, "Centaur technology media unit verification," in *Computer Aided Verification*, ser. LNCS, A. Bouajjani and O. Maler, Eds., vol. 5643. Springer, 2009, pp. 353–367.

[44] D. A. Jamsek, *Symbolic Trajectory Evaluation*, ch. 12, pp. 185–200.

[45] S. O. Swords, "A verified framework for symbolic execution in the ACL2 theorem prover," Ph.D. dissertation, University of Texas as Austin, 2010. [Online]. Available: http://hdl.handle.net/2152/ETD-UT-2010-12-2210

[46] T. F. Melham and R. B. Jones, "Abstraction by symbolic indexing transformations," in *Formal Methods in Computer-Aided Design*, ser. LNCS, vol. 2517. Springer-Verlag, 2002, pp. 1–18.