An Industrially Effective Environment for Formal Hardware Verification

Carl-Johan H. Seger, Robert B. Jones, John W. O'Leary,
Tom Melham, Mark D. Aagaard, Clark Barrett, and Don Syme

***Abstract*— We describe the Forte formal verification environment for datapath-dominated hardware, which has proved effective in large-scale industrial trials. Forte combines an efficient linear-time logic model checking algorithm, symbolic trajectory evaluation, with lightweight theorem proving in higher-order logic. These are tightly integrated in a general-purpose functional programming language, which both allows the system to be easily customized and also serves as a specification language. We also describe the design philosophy behind Forte and elements of the verification methodology that make it effective in practice.**

***Index Terms*— Formal verification, theorem proving, model checking, Symbolic Trajectory Evaluation, BDDs.**

I. Introduction

FUNCTIONAL validation is one of the major challenges in chip design today, with conventional approaches to design validation a serious bottleneck in the design flow. Over the past ten years, formal verification [1] has emerged as a complement to simulation and has delivered promising results in trials on industrial-scale designs [2], [3], [4], [5], [6].

Formal equivalence checking is widely deployed to compare the behavior of two models of hardware, each represented as a finite state machine or simply a Boolean expression (often using Binary Decision Diagrams, BDDs [7]). It is typically used in industry to validate the output of a synthesis CAD tool against a 'golden model' expressed in register-transfer level HDL, and in general to check consistency between other adjacent levels in the design flow.

Property checking with a model checker [8], [9], [10], [11] also involves representing a design as a finite state machine, but it has wider capabilities than equivalence checking. Not only can one check that a design behaves the same as another model, one can also check that the hardware possesses certain desirable properties expressed more abstractly in a temporal logic. An example is checking that all requests are eventually acknowledged in a protocol. Model checking is currently much less widely used in practice than equivalence checking.

C.-J. H. Seger, R. B. Jones, and J. W. O'Leary are at Strategic CAD Labs, Intel Corporation, JF4-211, 2111 NE 25th Avenue, Hillsboro, OR 97124, USA. E-mail: {cseger,rjones,joleary}@ichips.intel.com.

T. Melham is at Oxford University Computing Laboratory, Wolfson Building, Parks Road, Oxford, OX1 3QD, England. E-mail: Tom.Melham@comlab.ox.ac.uk.

M. D. Aagaard is at Department of Electrical and Computer Engineering, University of Waterloo, Wateroo, Ontario, N2L 3G1, Canada. E-mail: maagaard@uwaterloo.ca.

C. Barrett is at Department of Computer Science, Courant Institute of Mathematical Sciences, New York University, 251 Mercer Street, New York, NY 10012, USA. E-mail: barrett@cs.nyu.edu.

D. Syme is at Microsoft Research, 7 J J Thomson Ave., Cambridge, UK, CB3 0FB. E-mail: dsyme@microsoft.com.

Theorem proving [12], [13] allows higher-level and more abstract properties to be checked. It provides a much more expressive language for stating properties—for example, higher-order logic [14]—and it can deal with infinite-state systems. In particular it allows one to reason with unknowns and parameters, so a general class of designs can be checked—for example, parameterized IP blocks [15]. Industrially, theorem proving is still viewed as very advanced technology and its use is not yet widespread.

BDDs, equivalence checkers, and model checkers all suffer from severe capacity limits. In practice only small fragments of systems can be handled directly with these technologies, and much current research is aimed at extending capacity. Of course it is unrealistic to expect a completely automatic model checking solution. Instead, one needs to find good ways of using human intelligence to extract the maximum potential from model checking algorithms and to decompose problems into appropriate pieces for automated analysis. One approach is to combine model checking and BDD-based methods with theorem proving [16], [17], [18]. The hope is that theorem proving's power and flexibility will enable large problems to be broken down or transformed into tasks a model checker finds tractable. Another approach is to extend the top-level of a model checker with ad-hoc theorem proving rules and procedures [19].

In this paper, we describe a formal verification environment called Forte that combines an efficient linear-time logic model checking algorithm, symbolic trajectory evaluation [20], with lightweight theorem proving in higher-order logic. These are interfaced to and tightly integrated with FL [21], a strongly-typed, higher-order functional programming language. As a general-purpose programming language, FL allows the Forte environment to be customised and large proof efforts to be organized and scripted effectively. FL also serves as an expressive specification language at a level far above the temporal logic primitives.

The Forte environment has proved highly effective in large-scale industrial trials on datapath-dominated hardware [3], [22], [23]. The restricted temporal logic of symbolic trajectory evaluation does not, however, limit Forte to pure datapath circuits. Many large 'control' circuits are 'datapath-as-control', and these can also be handled effectively. In addition, the tight connection to higher-order logic and theorem proving provides great flexibility in decomposing verifications into sub-problems that fall into the scope of symbolic trajectory evaluation.

In the next section, we give an account of the design

philosophy behind the Forte environment and describe key aspects of the verification methodology that make it effective in practice. The rest of the paper is then structured as follows. Section III introduces some notation used in the remainder of the paper. Sections IV and V describe symbolic trajectory evaluation ('STE') and some techniques used to scale it up to large circuits. Section VI then explains how the Forte environment embeds STE in the context of the FL programming language. Sections VII and VIII describe how Forte's higher order logic theorem prover is built on FL and how Forte combines STE model checking and deductive theorem proving. Finally, Section IX describes some industrial case studies that illustrate some of the main verification strategies supported by Forte.

The Forte system [24] has recently been made publicly available for non-commercial use.[1]

II. Verification and Tool Design Philosophy

There is a large literature on formal proof methods and tools for hardware design verification and debugging. In our experience, this technology is practical only when embedded in a rather sophisticated and finely-tuned environment. All the components deployed—specification languages, model checking algorithms, theorem proving techniques, debugging aids, and so on—must work smoothly together. Substantial engineering effort is needed to move implementations beyond the academic prototype stage typical of most research tools. Additionally, it is critical that the technology be supported by a realistic usage methodology [25], [26].

A. Formal Specification

The backbone of any verification or design debugging effort is a formal *specification* of required behaviour—or, more loosely, some group of *properties* the design is expected to satisfy, expressed in a formal specification language. We advocate a 'foundational' approach; specifications are expressed in a formalism with only a few very simple temporal logic primitives, but which is also embedded in a full-featured functional programming language. Expressions of the language have a mathematical (logical) interpretation, and so they provide a powerful and extensible layer of specification language on top of the logic primitives.

This approach gives a generic, open framework in which to engineer tailored solutions to individual verification problems. For each verification effort or project, the user can create in the FL functional language just the right specification constructs for the problem domain. In practice, one can reuse much of this over a whole class of verifications, e.g. floating point algorithms. For many verifications suitable FL libraries may already exist, so the tailoring effort can be cost-effective.

By contrast, specialized formal languages are ready for use 'out of the box' but are limited in scope. They can also lead to biases in specifications; just because a branch predictor *can* be written in a notation specialised for some other domain doesn't mean it's the most natural way to describe branch prediction. The use of FL with temporal logic primitives also gives a specification language that is, in principle, verification-algorithm neutral and can support abstraction extensions.

Specifications should ideally be concise, implementation-independent, and at a high level of abstraction. Otherwise, there is the danger that specifications are too 'brittle' to track a rapidly-changing design or to be reused on a similar project later on. Small-block verification runs the further danger of reverse engineering meaningless 'specifications' from the circuit itself.

A corollary is that we aim to specify and verify the implementation of *functions*, not to describe and prove correct specific blocks of circuitry. For example, we speak of verifying floating point *instructions* [3], not execution units. A rule of thumb would be to first decompose by function computed and only then structurally (e.g. into pipe stages).

Model checking capacity limits can, of course, compromise the quality of specifications by preventing the verification at the scale needed to implement coherent functionality. Also, optimizing a specification for model checking efficiency can tangle the specification, making the intention less clear. Hence Forte's model checking algorithms and BDD data structures are engineered to scale up to fairly large blocks of hardware.

B. Debugging in Verification

The bulk of any verification effort is debugging, so it is crucial to optimize the whole verification environment for proof *failure*, not success. We not only want the system to inform us (quickly) when a verification fails; we also want it to provide focused feedback to help us pinpoint the cause of failure. This means a tight, rapid debug loop: simulate (or, later on, verify) the circuit, analyze and debug any counter-example, modify the specification or circuit, and re-simulate.

In practice, most of the bugs are in the verification process itself rather than the device being verified. Early on there will be many bugs in the specification; later, these will become more subtle and harder to distinguish from genuine circuit bugs. An effective environment must supply good machinery for exploring both kinds of bugs, and it must give good feedback in the user's terms from the tools.

Experience has shown that automation and visualization play a large role in providing effective debugging support. It is, of course, essential to be able to both execute specifications and simulate circuits for specific input values to investigate disagreements. Executable specifications are naturally expressible in FL, and simulation is fundamental to symbolic trajectory evaluation. The Forte environment also provides automatic counter-example generation for failed model checking runs, with special care taken about translating internally-generated counter-examples into the user's terms. Additionally, counter-example analysis in Forte is tightly integrated with tools for visualizing circuits and waveforms.

Using a concrete counter-example to isolate the source of a problem is very helpful, but requires the user to understand the behavior of internal signals and do a mental comparison between the circuit and the correct behavior to identify where

[1] Available for download at www.intel.com/software/products/opensource/tools1/verification/download.htm.

the circuit fails to meet the specification. It can be much more effective if the user can explore the entire failure domain, or at least intellectually-recognizable subsets of it. For this purpose, Forte provides 'what-if analysis' to help the user understand the failed proofs. The STE model-checking algorithm computes a data structure that gives complete characterization of the difference between the circuit and specification; the user can then invoke FL programs, either drawn from a library or tailor-made for the problem at hand, that probe this data-structure in informative ways.

This facility allows the user to focus on interesting and easily understood counter-examples, instead of being limited to ones chosen arbitrarily by the system. For example, it is often helpful to see a counter-example with as few signals asserted as possible. This can easily be achieved by calling an FL library function that generates a concrete counter-example with this property from the failure domain computed by STE. With a little bespoke FL programming, more domain-specific analyses are also possible. For example, knowing that an arithmetic circuit processes odd numbers correctly but not even ones could focus attention on the least-significant bit, rather than (say) straying into areas of the circuit that compute the sign. One can obtain this information by writing appropriate FL functions to probe the failure domain—in this case discovering that it contains only even numbers.

C. Reuse of Verification Effort

Verification is an expensive, human-intensive activity. We therefore want to support the reuse of proof efforts to amortize verification cost over the lifetime of a changing design, or even over multiple design projects. Two particularly good targets for reuse are specifications and high-level problem decomposition strategies. These often do not vary greatly from implementation to implementation, and with the right technical machinery they can be insulated from the messy details of individual circuits.

Reuse of specifications depends on having the capacity to treat fairly large-grained functionality, so that specifications can be made as circuit-independent as possible. It must also be possible to structure specifications in a way that separates the circuit-dependent parts and functional parts. In Forte, program structuring in FL is the technology that makes this possible. In practice, of course, achieving a well-structured specification and interface to the circuit needs thoughtful and skilled design. Forte's usage methodology provides some guidance here.

Reuse of problem decomposition strategies depends on making the strategies circuit-independent. A proof based on structural decomposition, or some other implementation feature, is not likely to be reusable for future designs. In particular, model checking routinely handles circuits so large that it is unlikely that one will ever see two identical components whose proofs can be the same.

A better strategy is to base verification methodologies on patterns or common structures in specifications, rather than on patterns in implementations. Specifications are cleaner and generally suffer from fewer idiosyncrasies than circuits. For example, case-splitting strategies derived from analysis of an algorithmically-formulated specification can often be made to transfer across many different circuit implementations.

D. Usage Methodology

Although continued advances in algorithm and data structure design have increased the reach of formal verification, a large gap remains between the capability offered by verification point-tools and modern design complexity. Much current research targets the well-known problem of model checking capacity limits, but often overlooks the equally important problem of managing the complexity of the verification activity itself. We have therefore coupled our research on technology with work on practical verification *methodology* [25], [26].

Our aim is to make formal verification work at an industrial scale, where any serious verification effort faces many complexity problems in addition to model checking capacity. For example, large verifications are almost always decomposed into many model checking runs—frequently many hundreds. Organizing all the cases to be considered into a coherent whole or even specifying them clearly (let alone discovering them) is complex, intellectually demanding, and error-prone. Our methodology addresses this particular complexity problem by generating and organizing model checking runs systematically.

More generally, the Forte usage methodology gives guiding structure and sequence to the many interdependent and complex activities of a large verification effort. It also helps structure the associated code and proof-script artifacts. The methodology aims, on the one hand, to face the messy realities of design practice (e.g. rapid changes and incomplete specifications) and, on the other hand, to produce high-quality results that are understandable, maintainable—and possibly even reusable.

In related work, Martin *et al.* have developed Versys2, a tool and methodology that uses STE to verify switch-level models of embedded memories against register-transfer-level specifications [27], [28]. By specializing their application to embedded memories, they are able to automate up to 90% of the task of generating specifications and verification scripts. The resulting reduction in verification effort and prerequisite verification expertise enables design engineers to perform most of the verification of embedded memories. In other work, Abadir *et al.* have used the Cycle Based Verilog (CBV) language as the basis for a verification methodology that evolves from formal verification of small blocks to conventional simulation for larger blocks and full-chip verification [29]. In contrast to this work, our aim is to enable industrial-scale formal verification on a wide variety of circuits.

Experience from Forte shows that an effective, broad-spectrum verification methodology must meet several key requirements:

Realism: An effective verification methodology cannot depend on resources that are not available in the design environment. For example, complete specifications are usually not available, and access to design engineers may be limited.

Transparency and confidence: The verification engineer (and design managers) should clearly know what has been proved and what has not. Of course the methodology and tools should also be sound; false positives should not be possible.

Structure: An effective methodology imposes structure on the overall verification effort. This not only helps new users learn, but also increases the productivity of experienced users.

The Forte methodology moves through a series of specific phases, each of which has a specific aim and produces well-defined FL code artifacts [25], [26]. Briefly, the phases are:

1) *Wiggling (understanding circuit I/O)*
2) *Targeted scalar verification*
3) *Symbolic model checking*
4) *Theorem proving*

Having a clear understanding of this sequence gives guiding structure to the work of Forte users. Each phase also produces well-defined code artifacts, which helps to structure verification code. For example, each phase produces an FL function (called a 'test rig') that drives circuit simulation for the purposes of that phase. In each phase, this code evolves from the code of the previous phase by adding some specific new elements. At the completion of a phase, the code is archived for regression.

Early results: Preliminary results are needed early in a verification effort. There must be a smooth transition between simulation of special cases and a full proof, so that the effort spent can deliver 'debugging value' very early on.

The simulation-based technology in Forte is especially helpful here. Conventional reachability analysis encounters capacity problems from the very beginning; its goal is to carve out a chunk of circuitry small enough for the capacity of the model checker, but not so small that false counter-examples are generated. In contrast, with symbolic trajectory evaluation and the Forte methodology, one encounters capacity problems gradually. Well before full model checking is attempted, symbolic trajectory evaluation allows debugging with straight simulation or even mixed scalar and symbolic simulation.

Incrementality and regression: If changing a specification, circuit, or library causes a previously-passing proof to fail, it should be possible to use proof artifacts (e.g. test rigs or simulations) from earlier in the effort to help isolate the problem. The methodology's verification artifacts should be easy to maintain and adapt to changing specifications and designs. Test cases from initial proof development should continue to be usable in exploring these changes.

This notion of *incrementality*—always being able to retreat onto solid ground—is especially important when moving into a more advanced phase of work. For example, when a model checking run fails inside a Forte theorem proving proof, it is helpful to be able to 'back off' and execute the model checking run by itself without performing any theorem proving. This makes the debug loop faster, since it is easier to use the debugging facilities tightly integrated with the symbolic trajectory evaluation model checker than operate them through a layer of theorem proving. Likewise, when a symbolic model checking run fails, we can generate a counter-example and then retreat to the simulation domain to analyse it.

Bottom-up and top-down: An effective methodology must support a mix of bottom-up and top-down techniques. The subtle features of designs and the capacity limits of model checking are discovered through bottom-up exploration. Overall problem reduction is achieved by top-down decomposition, using case splitting, induction, or some other algorithm-specific technique.

The beginning phases of the Forte methodology are primarily bottom-up. This gives early delivery of results and grounds verification in the concrete, transparent world of simulation. One also discovers the (rather unpredictable) limits of model checking by bottom-up exploration, which is much easier than trying somehow to arrive at suitable sub-problems by a top-down decomposition.

The top-down activity of developing case-splits, induction strategies, and other problem reduction strategies—i.e. the high-level proof strategy—starts a bit later, but then proceeds in parallel. This aspect requires an understanding of the algorithm, but the earlier bottom-up explorations help to provide this and to set a definite target for top-down problem reductions to be discovered.

E. Approach to Technology

Verification by model checking using symbolic trajectory evaluation ('STE') lies at the core of the Forte environment.[2] STE can be viewed as a hybrid between a symbolic simulator and a symbolic model checker [20]. As a traditional simulator, it can compute the result of executing a circuit with concrete Boolean test vectors as inputs; as a *symbolic* simulator, it can also compute symbolic expressions giving outputs as a function of arbitrary inputs. As a model checker, STE can automatically check the validity of a simple temporal logic formula for arbitrary inputs—computing an exact characterization of the region of disagreement in case the formula is not unconditionally satisfied. STE's seamless connection between simulation and verification is crucial to satisfying our requirement for early results.

STE is a particularly efficient model checking algorithm, in part because it has a very restricted temporal logic. But it is well-known that the capacity of any model checker is very limited. To be practical on even small examples, significant engineering effort must be combined with special algorithmic techniques, like partial order reduction [30] or on-the-fly simplification of transition relations [31]. Forte employs a full range of such techniques, but also tackles capacity limits by complementing STE model checking with higher-order logic theorem proving [12], [13].

Theorem proving bridges the gap between big, practically-important verification tasks and tractable model checking problems. From the users' point of view, the Forte philosophy is to have as thin a layer of theorem proving as possible, since using theorem proving technology is still quite difficult. Our experience from case studies is that a surprising amount of added value can be gained from even very simple (mathematically 'shallow') theorem proving.

Architecturally, the Forte approach is to tightly integrate model checking and theorem proving by implementing them within a single framework—the FL programming language

[2]Forte also includes a CTL model checker and others, but these will not be discussed in this paper.

and runtime system. A highly engineered and efficient (C-coded) implementation of STE is built into the core of FL, with numerous entry points into STE provided as user-visible FL functions. The Forte theorem prover (called 'ThmTac') is implemented in FL, with an architecture loosely based on the well-tested model of LCF-style proof systems [12].

Two key aspects of this architecture are that it is a 'white-box' integration of model checking and theorem proving and that the FL programming language plays a central role in scripting verification efforts.

1) White-Box Integration: Early efforts in integrating model checkers into theorem provers treated the model checker as a black-box decision procedure that could be invoked in the course of a proof [16], [17]. But experience has shown that a much tighter, white-box integration is far more practical. White box integration means having explicit access to the inner workings of the integrated model checker, for example to analyse or manipulate its internal data structures.

When the model checker is just a decision procedure within a theorem prover, the user can only invoke it to discharge proof obligations. This isolates the user from the powerful debugging and analysis capabilities typically built into model checkers. In addition, practical model checking is rarely, if ever, a simple function that comes back with 'true' or 'false'.

Moreover, if the model checker is a black box with command-line switches, then users learn to convolute their specifications to suit the model checking implementation and become experts at selecting particular flags for particular classes of problems. Verification scripts are then fragile with respect to changes in the specification, circuit, and model checking algorithm. This often leads to precipitous, rather than gradual, degradation when something goes wrong.

Practical model checking involves significant manual interaction, dealing with a variety of issues such as computation of intermediate state sets, installing and analyzing BDD variable orders, and modeling environments. A much more flexible and robust solution is a white-box model checker with a general-purpose programming language as its interface.

2) Proof-Script Programming: A large industrial verification will involve decomposition into many model checking runs—typically several hundred, or even thousands. Clearly some kind of 'verification script' is needed to manage this complex activity, and to form an editable and permanent record of the verification project.

At the very least, a verification script must generate the cases to be checked and control the invocation of individual model checking runs. Assumptions about the operating environment must be formally described (preferably in a user-comprehensible notation) and fed as contraints into the model checking algorithms. It is also essential to be able to check the completeness of coverage. Certainly it should be possible to do this by visual inspection, for example in formal code reviews. But completeness might also be assured by systematically generating the cases to be checked, or even by verifying coverage with theorem proving.

Few model checkers have native scripting capabilities that satisfy these requirements. Existing scripting languages such as Perl or Python are also less than ideal. Perl is best suited for what we might call 'file scripting', controlling essentially stateful actions that must happen in a roughly linear order. But verification scripting is more like the partially-ordered composition of functional operations. Each computation takes some previous results and combines them to produce the next result, and the final outcome is a single piece of data saying that the verification was successful. Moreover, Perl scripts are unsuitable subjects for reasoning about in a theorem prover.

Forte uses the functional language FL to script proof efforts. FL is a full-featured, lazy functional programming language, and so has the advantages of extensibility, semantic cleanness, and perspicuity. It also provides an interface to many of the functions and data structures of STE model checking, so model checking can be highly controlled and easily observed.

But the role of FL in Forte also extends much beyond these basic control and housekeeping functions. As will be seen in later sections, FL provides a specification language for hardware, and it is both the implementation language of the Forte theorem prover as well as the term language for its higher-order logic. FL therefore has a central, unifying role in the Forte architecture.

III. Mathematical Preliminaries

We write $\triangleq$ to mean *equals by definition*. We assume familiarity with elementary propositional logic and predicate calculus notation and use the symbol $\supset$ for logical implication, reserving the possibly more familiar symbols $\rightarrow$ and $\Rightarrow$ for other uses.

We use lower-case letters (e.g. a, p_1, v, x, y) for Boolean variables, and upper-case letters (e.g. P, Q) to stand for formulas of propositional logic (i.e. 'Boolean functions'). We write xs to mean a vector of distinct variables $x_0, x_1, \ldots, x_n$, for indeterminate n, and Ps to stand for a vector of formulas.

The notation $P[Qs/xs]$ stands for the result of simultaneously substituting the formulas Qs for all occurrences of the Boolean variables xs in P. When the notation $P[Qs]$ is used, it should be understood to represent a term obtainable as the result of such a substitution. Hence $P[xs]$ stands for a logic formula that may contain the variables xs. Normally, $P[xs]$ should simply be taken to mean a formula containing exactly the distinct Boolean variables in xs. In a context in which a formula has been written $P[xs]$, subsequent use of the notation $P[Qs]$ should be understood to mean $P[Qs/xs]$.

We also assume familiarity with the basic notation of naive set theory. (See, for example, [32].) If A and B are sets, we write $A \rightarrow B$ for the set of all total functions from A to B. We assume that $\rightarrow$ associates to the right, so $A \rightarrow (B \rightarrow C)$ may be written $A \rightarrow B \rightarrow C$. Function application associates to the left, so if $f \in A \rightarrow B \rightarrow C$, $a \in A$, and $b \in B$, we can write $f\ a\ b$ for $(f\ a)\ b$.

The semantics of STE depend on some elementary concepts of lattice theory [33]. If $(S, \sqsubseteq)$ is a partial order and $A \subseteq S$, then $x \in S$ is an *upper bound* for A iff $a \sqsubseteq x$ for all $a \in A$. A *lower bound* is defined dually. An upper bound x of A is the *least upper bound* of A, written $lub(A)$, if $x \sqsubseteq y$ for every upper bound y of A. The *greatest lower bound*, written $glb(A)$, is defined dually. We write $a \sqcup b$ (read 'a join b') for $lub\{a, b\}$

when it exists and $a \sqcap b$ (read 'a meet b') for $glb\{a,b\}$ when it exists.

A partial order $(S, \sqsubseteq)$ is a *complete lattice* iff $lub(A)$ and $glb(A)$ exist for all $A \subseteq S$. If S is finite and $a \sqcup b$ and $a \sqcap b$ exist for all $a, b \in S$, then $(S, \sqsubseteq)$ is a complete lattice.

IV. STE Model Checking

Symbolic trajectory evaluation [20] is an efficient model checking algorithm especially suited to verifying properties of large datapath designs. The most basic form of STE works on a very simple linear-time temporal logic, limited to implications between formulas built from only conjunction and the next-time operator.[3] In addition, STE is based on *ternary simulation* [35], in which the Boolean data domain $\{0,1\}$ is extended with a third value 'X' that stands for 'either 0 or 1, but we don't know which'. As will be seen later, this gives STE very powerful automatic state-space abstraction.

These characteristics allow STE to perform property checking much more efficiently than conventional model checking algorithms, which operate over more expressive logics like CTL [10]. While the logic of basic STE seems very weak, its expressive power is greatly extended by implementing a *symbolic* ternary simulation algorithm and by being embedded within the FL programming language.

Symbolic ternary simulation [36] uses symbolic Boolean variables and expressions over them (i.e. BDDs [7]) to represent whole classes of data values on circuit nodes. The ternary value associated with each node is represented by a BDD data structure whose variables act as parameters to that value. With this representation, STE can combine many (ternary) simulation runs—one for each assignment of values to the BDD variables—into a single *symbolic* simulation run covering them all.

The BDDs representing values at different circuit nodes can have variables in common, so this representation can also record complex interdependencies among node values. Symbolic values therefore greatly increase the expressive power of the limited temporal logic of STE. For example, input/output relations can be extracted from a circuit by using symbolic simulation to derive BDDs for the values on output nodes as functions of variables standing for arbitrary values on input nodes. These can then be checked against a specification in the form of some reference BDDs.

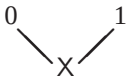
The expressive power of STE for specifications is also much extended in Forte by embedding the STE logic, including BDD variables, within the FL programming environment. For example, the 'reference BDDs' just mentioned for specifications are typically computed by FL programs, whose source text is the user's view of 'the specification'. Specifications can therefore be expressed clearly and in the user's own terms by using the full expressive power of FL, together with FL library functions especially tuned to describing the problem domain.

The rest of this section explains the background theory of STE model checking in a bit more detail. A full account of the theory can be found in [20] and a useful alternative perspective is given in [37].

[3] Extensions of STE to more expressive logics exist [20], [34] and some have implementations in Forte. But this paper will focus on the simplest form, which is also the most widely tested on industrial applications.

A. Circuit Models

Symbolic trajectory evaluation employs a ternary data model with values drawn from the set $D = \{0,1,\mathsf{X}\}$. A partial order relation $\leq$ is introduced, with $X \leq 0$ and $X \leq 1$:



This orders values by information content: X stands for an unknown value and so is ordered below 0 and 1.

We suppose there is a set of *nodes*, N, naming observable points in circuits. A *state* is an instantaneous snapshot of circuit behavior given by assigning a value in D to every circuit node in N. The ordering $\leq$ on D is extended pointwise to get an ordering $\sqsubseteq$ on states. We wish this to form a complete lattice, and so introduce a special 'top' state, $\top$, and define the set of states S to be $(N \to D) \cup \{\top\}$. The required ordering is then defined for states $s_1, s_2 \in S$ by

$$s_1 \sqsubseteq s_2 \triangleq \begin{cases} s_2 = \top, \text{or} \\ s_1, s_2 \in N \to D \text{ and } s_1(n) \leq s_2(n) \text{ for all } n \in N \end{cases}$$

The intuition is that if $s_1 \sqsubseteq s_2$, then s_1 may have 'less information' about node values than s_2, i.e. it may have Xs in place of some 0s and 1s. If one considers the three-valued 'states' s_1 and s_2 as *constraints* or *predicates* on the actual, i.e. Boolean, state of the hardware, then $s_1 \sqsubseteq s_2$ means that every Boolean state that satisfies s_2 also satisfies s_1. We say that s_1 is 'weaker than' s_2. (Strictly speaking, $\sqsubseteq$ is reflexive and we really mean 'no stronger than', but it is common to be somewhat inexact and just say 'weaker than'.) The top value $\top$ represents the unsatisfiable constraint. The *join* operator on pairs of states in the lattice is denoted by '$\sqcup$'.

The theory of symbolic trajectory evaluation can in fact be developed for any complete lattice of states [20]. But this generality is not exploited in mainstream implementations of STE, and so we restrict the presentation in this paper to the simple state lattice introduced above.

To model dynamic behavior, a sequence of the values that occur on circuit nodes over time is represented by a function $\sigma \in \mathbb{N} \to S$ from time (the natural numbers $\mathbb{N}$) to states. Such a function, called a *sequence*, assigns a value in D to each node at each point in time. For example, $\sigma\, 3\, reset$ is the value present on the *reset* node at time 3. We lift the ordering on states pointwise to sequences in the obvious way:

$$\sigma_1 \sqsubseteq \sigma_2 \triangleq \sigma_1(t) \sqsubseteq \sigma_2(t) \text{ for all } t \in \mathbb{N}$$

One convenient operation, used later in stating the semantics of STE, is taking the ith suffix of a sequence. The ith suffix of a sequence σ is written σ^i and defined by

$$\sigma^i\, t \triangleq \sigma\,(t+i) \text{ for all } t \in \mathbb{N}.$$

The suffix operation σ^i simply shifts the sequence σ forward i points in time, ignoring the states at the first i time units.

In symbolic trajectory evaluation, the formal model of a circuit c is given by a next-state function $\mathsf{Y}_c \in S \to S$ that maps states to states. Intuitively, the next-state function expresses a constraint on the real, Boolean states into which the circuit may go, given a constraint on the current Boolean state it is in.

A trivial example is this the unit-delay AND-gate, shown together with a partial tabulation of its Y function in Fig. 1. The circuit has three nodes, a, b, and o, and we write a state s as a vector $s(\mathsf{a})\,s(\mathsf{b})\,s(\mathsf{o})$. For example $s = 1\mathsf{X}0$ means $s(\mathsf{a}) = 1$, $s(\mathsf{b}) = \mathsf{X}$, and $s(\mathsf{o}) = 0$. Reading from the left, we first see that if the inputs a and b are both 1, then the next state is $\mathsf{XX}1$, regardless of whether o is initially 0 or 1. Hence the output o is 1 in the next state and the inputs a and b are both X (i.e. they can be either Boolean value). In fact, the value of o in the next state doesn't depend on the value of o in the current state, so a little further along in the table we also find $\mathsf{Y}(11\mathsf{X}) = \mathsf{XX}1$.

We also see that if b is 0 in the current state, then the output o is going to be 0 in the next state—regardless of the value of a in the current state. Hence we have $\mathsf{Y}(\mathsf{X}0\mathsf{X}) = \mathsf{XX}0$; we know what the output value will be, even when we have no information about the value on a. Finally, we sometimes have insufficient information to determine the value of the output. If the current state is $\mathsf{X}1\mathsf{X}$, for example, then we don't know whether a is going to be 0 or 1—it may be either, and hence $\mathsf{Y}(10\mathsf{X}) = \mathsf{XXX}$.

In STE, the next-state function for any circuit must be monotonic and a requirement for implementations of STE is that they extract a next-state function that has this property from the circuit under analysis. This condition can be met for a wide variety of common circuit design styles, including synchronous systems with latches as well as flip-flops and systems with gated clocks.

A sequence σ is said to be a *trajectory* of a circuit if it represents a set of behaviors that the circuit could actually exhibit. That is, the set of behaviors that σ represents (i.e. possibly using unknowns) is a subset of the Boolean behaviors that the real circuit can exhibit (where there are no unknowns). For a circuit c, we define the set of all its trajectories, $T(c)$, as follows:

$$T(c) \triangleq \{\sigma \mid \mathsf{Y}_c(\sigma\, t) \sqsubseteq \sigma\,(t{+}1) \text{ for all } t \in \mathbb{N}\}$$

For a sequence σ to be a trajectory, the result of applying Y_c to any state must be no more specified (with respect to the $\sqsubseteq$ ordering) than the state at the next moment of time. This ensures that σ is consistent with the circuit model Y_c.

B. Trajectory Evaluation Logic

One of the keys to the efficiency of STE and its success with datapath circuits is its restricted temporal logic. A *trajectory formula* is a simple linear-time temporal logic formula with the following syntax:

$$\begin{array}{lcll} f,g & ::= & n \text{ is } 0 & \text{- node } n \text{ has value 0} \\ & \mid & n \text{ is } 1 & \text{- node } n \text{ has value 1} \\ & \mid & f \text{ and } g & \text{- conjunction of formulas} \\ & \mid & P \triangleright f & \text{- } f \text{ is asserted only when } P \text{ is true} \\ & \mid & \mathsf{N} f & \text{- } f \text{ holds in the next time step} \end{array}$$

where f and g range over formulas, $n \in N$ ranges over the nodes of the circuit, and P is a propositional formula over Boolean variables (i.e. a 'Boolean function') called a *guard*.

The basic trajectory formulas 'n is 0' and 'n is 1' say that the circuit node n has value 0 or value 1, respectively. The operator and forms the conjunction of trajectory formulas. The trajectory formula $P \triangleright f$ weakens the subformula f by requiring it to be satisfied only when the guard P is true. Finally, $\mathsf{N} f$ says that the trajectory formula f holds in the next point of time.

In essence, a trajectory formula represents a whole *set* of assertions about the presence of the Boolean values 0 and 1 on particular circuit nodes. A guard is a propositional formula that may contain Boolean variables, and a trajectory formula $P \triangleright f$ with a guard P asserts f only for satisfying assignments of values to the Boolean variables in P. So for any trajectory formula, each assignment of values to the variables in its guards gives a (possibly different) assertion about 0s and 1s on certain circuit nodes at particular points in time.

The various guards that occur in a trajectory formula can have variables in common, so this mechanism gives STE the expressive power needed to represent interdependencies among node values. For example, we can associate an arbitrary propositional formula with a node using the construct 'n is P' defined by

$$n \text{ is } P \triangleq P \triangleright (n \text{ is } 1) \text{ and } \neg P \triangleright (n \text{ is } 0)$$

We can then specify input-output functions using this construct. For example, we might require that if 'in is x' then 'out is $F[x]$' for some input node in and output node out.

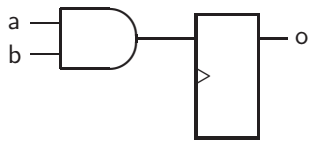
The definition of when a sequence σ satisfies a trajectory formula f is now given. Satisfaction is defined with respect to an assignment ϕ of Boolean truth-values to the variables that appear in the guards of the formula. Following conventional terminology from logic semantics, we call ϕ a *valuation*. We also write $\phi \models P$ to mean that the propositional formula P is true under the valuation ϕ.

For a given valuation ϕ, we define when a sequence σ satisfies a trajectory formula recursively as follows:

$$\begin{array}{lcl} \phi,\sigma \models n \text{ is } 0 & \triangleq & \begin{cases} \sigma(0) = \top, \text{ or} \\ \sigma(0) \in N \to D \text{ and } \sigma\,0\,n = 0 \end{cases} \\ \phi,\sigma \models n \text{ is } 1 & \triangleq & \begin{cases} \sigma(0) = \top, \text{ or} \\ \sigma(0) \in N \to D \text{ and } \sigma\,0\,n = 1 \end{cases} \\ \phi,\sigma \models f \text{ and } g & \triangleq & \phi,\sigma \models f \text{ and } \phi,\sigma \models g \\ \phi,\sigma \models P \triangleright f & \triangleq & \phi \models P \text{ implies } \phi,\sigma \models f \\ \phi,\sigma \models \mathsf{N} f & \triangleq & \phi,\sigma^1 \models f \end{array}$$

Note that the same valuation ϕ applies to all the guards that appear in a trajectory formula—so the scope of any Boolean variable is the entire formula. The valuation also does not depend on time.

The key feature of this logic is that for any trajectory formula f and assignment ϕ, there exists a unique weakest sequence that satisfies f. This sequence is called the *defining sequence* for f and is written $[f]^\phi$. It is defined recursively as follows:



$$\begin{array}{rcl} s & = & 111 \quad 110 \quad 101\cdots11\mathsf{X} \quad 10\mathsf{X}\cdots\mathsf{X}1\mathsf{X} \quad \mathsf{X}0\mathsf{X}\cdots\mathsf{XXX} \\ \mathsf{Y}(s) & = & \mathsf{XX}1 \quad \mathsf{XX}1 \quad \mathsf{XX}0\cdots\mathsf{XX}1 \quad \mathsf{XX}0\cdots\mathsf{XXX} \quad \mathsf{XX}0\cdots\mathsf{XXX} \end{array}$$

Fig. 1. Simple example of the next-state function

$$\begin{array}{lcl} [m \text{ is } 0]^\phi\, t & \triangleq & \lambda n.\, 0 \text{ if } m{=}n \text{ and } t{=}0, \text{ otherwise } \mathsf{X} \\ [m \text{ is } 1]^\phi\, t & \triangleq & \lambda n.\, 1 \text{ if } m{=}n \text{ and } t{=}0, \text{ otherwise } \mathsf{X} \\ [f \text{ and } g]^\phi\, t & \triangleq & ([f]^\phi\, t) \sqcup ([g]^\phi\, t) \\ [P \triangleright f]^\phi\, t & \triangleq & [f]^\phi\, t \text{ if } \phi \models P, \text{ otherwise } \lambda n.\mathsf{X} \\ [\mathsf{N} f]^\phi\, t & \triangleq & [f]^\phi\, (t{-}1) \text{ if } t{\neq}0, \text{ otherwise } \lambda n.\mathsf{X} \end{array}$$

The crucial property enjoyed by this definition is that $[f]^\phi$ is the unique weakest sequence that satisfies f for the given ϕ. That is, for any ϕ and σ, $\phi, \sigma \models f$ if and only if $[f]^\phi \sqsubseteq \sigma$.

The algorithm for STE is also concerned with the weakest *trajectory* that satisfies a particular formula. This is the *defining trajectory* for a formula, written $[\![f]\!]^\phi$. It is defined by the following recursive calculation:

$$\begin{array}{lcl} [\![f]\!]^\phi\, 0 & \triangleq & [f]^\phi\, 0 \\ [\![f]\!]^\phi\, (t{+}1) & \triangleq & [f]^\phi\, (t{+}1) \sqcup \mathsf{Y}_c([\![f]\!]^\phi\, t) \end{array}$$

The defining trajectory of a formula f is its defining sequence with the added constraints on state transitions imposed by the circuit, as modeled by the next-state function Y_c. It can be shown that $[\![f]\!]^\phi$ is the unique weakest trajectory that satisfies f. That is, for any ϕ and σ, we have that $\sigma \in T(c)$ and $\phi, \sigma \models f$ if and only if $[\![f]\!]^\phi \sqsubseteq \sigma$.

As will be seen in the next section, these properties justify using the calculation of defining sequences and defining trajectories as the basis of the STE model checking algorithm.

C. Model Checking Trajectory Assertions

Circuit correctness in symbolic trajectory evaluation is stated with *trajectory assertions* of the form $A \Rightarrow C$, where A and C are trajectory formulas. The intuition is that the *antecedent* A provides stimuli to circuit nodes and the *consequent* C specifies the values expected on circuit nodes as a response.

For example, the AND-gate shown in Fig. 1 would be verified with the following trajectory assertion:

$$\models (\mathsf{a} \text{ is } a) \text{ and } (\mathsf{b} \text{ is } b) \Rightarrow (\mathsf{o} \text{ is } a \wedge b)$$

The Boolean variables a and b are used to represent the values on the input nodes and to relate these to the expected value, '$a \wedge b$', on the output node.

A trajectory assertion is true for a given assignment ϕ of Boolean values to the variables in its guards exactly when every trajectory of the circuit that satisfies the antecedent also satisfies the consequent. For a given circuit c, we define $\phi \models A \Rightarrow C$ to mean that for all $\sigma \in T(c)$, if $\phi, \sigma \models A$ then $\phi, \sigma \models C$. The notation $\models A \Rightarrow C$ means that $\phi \models A \Rightarrow C$ holds for all ϕ.

The fundamental theorem of trajectory evaluation [20] follows immediately from the previously-stated properties of $[f]^\phi$ and $[\![f]\!]^\phi$. It states that for any ϕ, the trajectory assertion $\phi \models A \Rightarrow C$ holds exactly when $[C]^\phi \sqsubseteq [\![A]\!]^\phi$. The intuition is that the sequence characterizing the consequent must be 'included in' the weakest sequence satisfying the antecedent that is also consistent with the circuit.

This theorem gives a model-checking algorithm for trajectory assertions: to see if $\phi \models A \Rightarrow C$ holds for a given ϕ, just compute $[C]^\phi$ and $[\![A]\!]^\phi$ and compare them point-wise for every circuit node and point in time. This works because both A and C will have only a finite number of nested next-time operators N, and so only finite initial segments of the defining trajectory and defining sequence need to be calculated and compared.

In practice, the defining trajectory of A and the defining sequence of C are computed iteratively, and each state is checked against the ordering requirement as it is generated. Each state of the defining trajectory is computed from the previous state by simulation of a netlist description of the circuit over the value domain $\{0, 1, \mathsf{X}\}$.

Symbolic Trajectory Evaluation: The model checking algorithm just sketched requires ϕ to be supplied; given a specific assignment ϕ of values to Boolean variables in the guards of a formula, we can calculate and point-wise compare $[C]^\phi$ and $[\![A]\!]^\phi$. But much of the debugging power of STE comes from the key observation that it is not necessary to supply ϕ in advance; instead, the comparison can be computed 'symbolically' to give a *constraint* on ϕ. Such a constraint is called a *residual* and represents precisely the conditions under which the property $A \Rightarrow C$ is true of the circuit.

This symbolic version of the model checking algorithm works as follows. At the level of basic data values in $\{0, 1, \mathsf{X}\}$, the required computation is to show that

$$[C]^\phi\, t\, n \leq [\![A]\!]^\phi\, t\, n \tag{1}$$

for all $t \geq 0$ and $n \in N$. For each circuit node at each relevant point in time, we compare the data values expected by the consequent to those given by the circuit and antecedent. To make this comparison 'symbolic', we use a pair of BDDs to encode functions from ϕ to data values in D. This is the so-called 'dual-rail' encoding employed STE implementations [38]. We also extend the simulation algorithm to a symbolic version, in which data values are these pairs of BDDs. The model checking algorithm then compares symbolic states, resulting in the residual.

D. The STE Deductive System

STE has a sound and complete deductive system for proving trajectory formulas [39], [40], which have been implemented as a set of inference rules in the Forte theorem prover. The complete set of rules is as follows.

- *Reflexivity.* $\models A \Rightarrow A$ holds for any trajectory formula A.
- *Time Shift.* For any trajectory formulas A and C, if $\models A \Rightarrow C$ then $\models \mathsf{N}A \Rightarrow \mathsf{N}C$.
- *Antecedent Strengthening.* For any trajectory formulas A and C, if $\models A \Rightarrow C$ then for any trajectory formula A' for which $[A]^\phi \sqsubseteq [A']^\phi$ for all ϕ, we have $\models A' \Rightarrow C$.
- *Consequent Weakening.* For any trajectory formulas A and C, if $\models A \Rightarrow C$ then for any trajectory formula C' for which $[C']^\phi \sqsubseteq [C]^\phi$ for all ϕ, we have $\models A \Rightarrow C'$.
- *Conjunction.* For any trajectory formulas A_1, A_2, C_1, and C_2, if $\models A_1 \Rightarrow C_1$ and $\models A_2 \Rightarrow C_2$, then $\models A_1 \mathsf{\ and\ } A_2 \Rightarrow C_1 \mathsf{\ and\ } C_2$.
- *Transitivity.* For any trajectory formulas A, B, and C, if $\models A \Rightarrow B$ and $\models B \Rightarrow C$, then $\models A \Rightarrow C$.
- *Substitution.* For any trajectory formulas A and C, if $\models A \Rightarrow C$, then $\models A[Ps/xs] \Rightarrow C[Ps/xs]$ for any substitution of formulas Ps for Boolean variables xs.

The main purpose of these rules is to combine individual STE model-checking results together [41] to derive correctness results that are infeasible to model-check directly. The inference rules can also be used to transform trajectory formulas to increase model checking efficiency [42]. The use of STE inference rules to support these strategies is illustrated by the examples in Section IX.

V. STE in Practice

For a given trajectory assertion $A \Rightarrow C$ and circuit c, STE implementations construct the defining trajectory $[\![A]\!]^\phi$ incrementally by ternary symbolic simulation of an HDL or a netlist source for the circuit c under the antecedent A. The circuit model Y_c exists only implicitly in the sequence of simulation states constructed. The number of circuit nodes that can be handled by the symbolic simulator used in this process is essentially unlimited; the limit on the capacity of STE comes from the memory requirements for representing the symbolic values on each node.

Two important optimizations enable STE to be applied to a much larger class of circuits and properties than would otherwise be feasible. *Weakening*, the first of these optimizations, exploits the partially-ordered lattice of STE. The second optimization, the *parametric representation*, takes advantage of the fact that STE is implemented with a symbolic simulator.

Both optimizations can make a big difference to the time and space needed for verification, and can enable verification of circuits that are infeasible to verify directly with STE.

A. Weakening

Weakening is a data abstraction technique that exploits the partially-ordered state space of STE. It is an implementation optimization, in that it reduces the complexity of the BDDs needed to verify a circuit property.

Recall from the previous section that the definition of a trajectory assertion $\models A \Rightarrow C$ is $[C] \sqsubseteq [\![A]\!]$, i.e. the defining sequence of the consequent C must be weaker than the defining trajectory of the antecedent A. We say that a node is *weakened* when its value is moved down in the lattice (towards X).

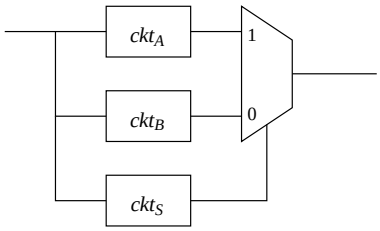


Fig. 2. Dynamic Weakening

Consider the extreme case, when a node's value is replaced with X by modifying the next-state function Y_c. This means that the node could be either 1 or 0, resulting in a new defining trajectory $[\![A]\!]_w$ that is 'weaker' than the original defining trajectory $[\![A]\!]$. If $[C] \sqsubseteq [\![A]\!]_w$ holds, then from definition of $\sqsubseteq$ and monotonicity we know that $[C] \sqsubseteq [\![A]\!]$ also holds. Note that if the verification fails with the weakened defining trajectory, then we can draw no conclusions about the original trajectory assertion.

Forte provides fine-grained access to weakening by user-level directives that list selected nodes and simulation times at which to weaken them. Users can manually weaken individual nodes at arbitrary points of time during simulation, with a view to reducing the BDD complexity of their values. This is safe, because the theory just sketched tells us that however a node's value is weakened during a verification, if the verification succeeds then the assertion being checked still holds.

One useful application of weakening is when different parts of a circuit require different BDD variable orderings. Consider the circuit shown in Fig. 2, which selects between the two values computed by ckt_A and ckt_B on the basis of a decision made by ckt_S. Such circuits are common in high-performance pipelines, where multiple speculative results are computed in parallel before knowing which result will be selected.

Suppose that for a particular case of a verification proof, we know that the value produced by the select logic in ckt_S will be 1. This means that the result of ckt_B will be 'blocked' at the mux while the result of ckt_A will be passed to the output. In this case of the proof, the computation done by ckt_B is irrelevant to the result, but because the nodes of ckt_B are in the fanin of the output they will not be removed from simulation by automatic cone-of-influence reduction.

If the variable ordering needed for computing the output of ckt_A is different from that for computing the output of ckt_B, the BDDs in ckt_B can explode in size. Because we know from the case split that the value computed by ckt_B is irrelevant, we would like to prevent its calculation in the first place. This can be accomplished by weakening every input and state node of ckt_B. If we do this, the output of ckt_B will be driven X by the circuit, and the BDD explosion will be avoided.

Of course the difficulty with this approach is knowing which nodes to weaken. It is often far from obvious which nodes are *not* involved in a particular computation, and discovering this can involve a very tedious and time-consuming manual effort. Moreover, our wish to support reuse means proof scripts should not be cluttered with implementation-specific information such as node names. This makes the proof scripts

brittle in the face of potential design changes and unsuable on future designs.

Forte therefore supports two other approaches that use weakening in a more automatic fashion. The first, *dynamic weakening*, applies weakening in a rather coarse way and requires little or no user intervention. The second, *symbolic indexing*, uses weakening systematically for verification of regular structures such as memory arrays.

1) Dynamic Weakening: In *dynamic weakening*, nodes in the circuit are simply weakened when their associated BDDs exceed some size threshold. This occurs dynamically during symbolic simulation and without user intervention. As a result, the weakening takes place without any mention of circuit nodes in proof scripts; the user provides only the size threshold. This works surprisingly well in practice, typically because the nodes with exploding BDDs (e.g. those in ckt_B above) are exactly the ones that do not have a substantive part to play in the property of interest.

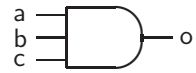
A subtle difficulty remains. Consider the case where the outputs of two sub-circuits are both needed but where the two subcircuits have different variable-ordering requirements. We cannot weaken one of the sub-circuits with Xs because its outputs are necessary for the computation. We address this difficulty by running STE more than once, each time with a different variable order.

Consider again the circuit in Fig. 2. Suppose the the BDD orderings needed for sub-circuits ckt_A and ckt_S are different. We first run STE with an ordering for ckt_S while weakening ckt_A and ckt_B. During this STE run, the output of ckt_S is traced and saved for later use; this process can be repeated as many times as necessary for each sub-circuit that requires a different ordering. For the final STE run, the variable ordering for ckt_A is used. The results from the earlier run for ckt_S (and potentially other sub-circuits) are composed with this STE run by strengthening the antecedent with the values traced from the output of ckt_S that were saved earlier.

An interesting example where we have encountered this situation is in the verification of floating-point adders. Modern adders use a performance-enhancing 'leading-zero anticipator' (LZA) circuit in subtract mode. The BDD variable order required to reason about LZA circuits is different from the variable orderings required for a variable-shift operation that is internal to the adder. We verified the adder by first running STE with the LZA ordering while tracing the outputs of the LZA sub-circuit. The antecedents for the 'main' STE runs were strengthened with the traced LZA values and performed with the variable orderings required by the core adder circuits. This was significantly easier than explicitly reasoning about the LZA circuitry, as required by other approaches [43].

2) Symbolic indexing: Symbolic indexing is a systematic way of using weakening to perform data abstraction for regular circuit structures. Like dynamic weakening, it is an implementation optimization. However, instead of managing BDD size by driving ternary values towards X, it reduces the number of BDD variables needed to verify certain circuit properties. Intuitively, symbolic indexing is a way to use BDD variables only 'when needed'.

The idea behind symbolic indexing can be illustrated using the following trivial example. Consider the three-input AND-gate shown below:



With direct use of STE, the assertion we would formulate to verify this device is the following:

$$\models (\mathsf{a}\ \mathsf{is}\ a)\ \mathsf{and}\ (\mathsf{b}\ \mathsf{is}\ b)\ \mathsf{and}\ (\mathsf{c}\ \mathsf{is}\ c) \Rightarrow (\mathsf{o}\ \mathsf{is}\ a \wedge b \wedge c) \qquad (2)$$

In primitive form, this would be expressed as follows:

$$\begin{array}{l} \models \neg a \triangleright (\mathsf{a}\ \mathsf{is}\ 0)\ \mathsf{and}\ a \triangleright (\mathsf{a}\ \mathsf{is}\ 1)\ \mathsf{and} \\ \quad \neg b \triangleright (\mathsf{b}\ \mathsf{is}\ 0)\ \mathsf{and}\ b \triangleright (\mathsf{b}\ \mathsf{is}\ 1)\ \mathsf{and} \\ \quad \neg c \triangleright (\mathsf{c}\ \mathsf{is}\ 0)\ \mathsf{and}\ c \triangleright (\mathsf{c}\ \mathsf{is}\ 1) \\ \qquad \Rightarrow \\ \quad \neg a \vee \neg b \vee \neg c \triangleright (\mathsf{o}\ \mathsf{is}\ 0)\ \mathsf{and}\ a \wedge b \wedge c \triangleright (\mathsf{o}\ \mathsf{is}\ 1) \end{array} \qquad (3)$$

The strategy here is to place unique and unconstrained BDD variables onto each input node in the device, and symbolically simulate the circuit to check that the desired function of these variables will appear on the output node. The total number of variables needed is the same as the number of input (plus state) nodes, in this case three.

Symbolic indexing exploits STE's partially-ordered state spaces to reduce the number of variables needed to verify a property. In the case of the AND gate, it turns out that we need to verify only the four cases enumerated in the table below:

case	a	b	c	o
0	0	X	X	0
1	X	0	X	0
2	X	X	0	0
3	1	1	1	1

If all three inputs are 1, then the output is 1 as well. But if at least one of the inputs is 0, the output will be 0 regardless of the values on the other two inputs. In these cases, therefore, we may use the lattice value X to represent the unknown truth-value present on the other two input nodes. As any weakened property implies a stronger property with any substitution of 0 or 1 for the unknown nodes, the four cases we have enumerated cover all possible input patterns of 0s and 1s and are sufficient for a complete verification of the AND-gate.

Symbolic indexing is the technique of introducing Boolean variables to enumerate or 'index' groups of cases like that just described. In the STE assertions (2) and (3) above, the cases we would like to enumerate are represented in terms of the three Boolean variables a, b, and c. Since there are just four cases to check, we can index the cases to be considered with two Boolean variables p and q, as shown below:

p	q	a	b	c
0	0	0	X	X
0	1	X	0	X
1	0	X	X	0
1	1	1	1	1

To verify these cases with STE, we would check the following trajectory assertion:

$$\begin{aligned}\models\ & \neg p \wedge \neg q \triangleright (\mathsf{a\ is\ 0}) \text{ and } p \wedge q \triangleright (\mathsf{a\ is\ 1}) \text{ and} \\ & \neg p \wedge q \triangleright (\mathsf{b\ is\ 0}) \text{ and } p \wedge q \triangleright (\mathsf{b\ is\ 1}) \text{ and} \\ & p \wedge \neg q \triangleright (\mathsf{c\ is\ 0}) \text{ and } p \wedge q \triangleright (\mathsf{c\ is\ 1}) \text{ and} \\ & \Rightarrow \\ & \neg p \vee \neg q \triangleright (\mathsf{o\ is\ 0}) \text{ and } p \wedge q \triangleright (\mathsf{o\ is\ 1})\end{aligned} \tag{4}$$

If this property is true, the device satisfies the specification of intended behavior for an AND-gate.

Symbolic indexing finds its greatest utility in verification of regular memory structures, as it significantly reduces the number of BDD variables required to encode data values [44], [45], [46]. Consider an $n \times m$-bit memory M with n rows and m bits per row, i.e. the memory is accessed with a $log_2 n$-bit address and returns m bits of data. Suppose we want to verify that the memory correctly stores and returns arbitrary data at every address. We could first perform a write operation to the symbolic address a_1 with the symbolic data vector d_1. Next, we would perform a read operation of a symbolic address a_2 and check that the result data d_2 matched what we had written previously.

To distinguish between each memory location in the direct verification approach would require $n \times m$ unique bits: m bits for each of n rows. For even a small memory, the number of variables required is too large for symbolic verification. But suppose we replace the i-th bit in the j-th row with the expression $P_j[a] \triangleright m_i$, where $P_j[a]$ is the appropriate address function for the j-th row. If the address bits select row j, then the value of bit i will be m_i. If the address bits select a different row, the value of bit i will be X. If we apply the same expression in each row of the memory array, only $m + \log_2 n$ variables are required: a significant reduction from the $m \times n$ otherwise needed.

See [47] for further details on symbolic indexing, including an algorithm for transforming directly-stated trajectory assertions, of the kind suitable for higher-level reasoning, into symbolically indexed form for efficient model-checking.

B. The Parametric Representation

In this section, we begin with a brief explanation of the parametric representation and then describe its use in STE verification. An extended treatment is found in [48].

The goal of the parametric representation is to encode a Boolean predicate P as a vector of Boolean functions whose range is exactly the set of truth assignments satisfying P. The technique is independent of the symbolic simulation algorithm, does not require any modifications to the circuit, can be used to constrain both input and internal signals, and is applicable to a wide variety of circuits.

In a parametric representation, a vector of functions over fresh parametric variables encodes a set of Boolean vectors whose elements are defined by a characteristic function. The range of the functional vector is exactly the original set. To illustrate, consider the following set:

$$S \triangleq \{(1,0,0,1),(1,0,0,0),(0,1,0,1)\}$$

If the input variables for the circuit are expressed as a vector $as = a_0, a_1, a_2, a_3$ then a non-minimized characteristic function (predicate) for S is:

$$\begin{aligned}P = & (a_0 \wedge \neg a_1 \wedge \neg a_2 \wedge a_3) \quad \vee \\ & (a_0 \wedge \neg a_1 \wedge \neg a_2 \wedge \neg a_3) \vee \\ & (\neg a_0 \wedge a_1 \wedge \neg a_2 \wedge a_3)\end{aligned}$$

The same set can be represented using a parametric functional vector with new *parametric* variables p_0 and p_1 as:

$$Qs = (p_0, \neg p_0, 0, \neg p_0 \vee p_1)$$

Any assignment to the parametric variables yields a truth assignment that satisfies P, as shown in the table below. Note that although $(0,1,0,1)$ appears twice in the table, the range of the parametric function vector is exactly the set S.

p_0	p_1	value of Qs
0	0	$(0,1,0,1)$
0	1	$(0,1,0,1)$
1	0	$(1,0,0,0)$
1	1	$(1,0,0,1)$

The parametric representation is used with a symbolic simulator by applying the vector Qs to the inputs in place of the original vector as. Each circuit input is replaced with the corresponding function from the vector of parametric functions:

$$\begin{aligned}a_0 &\mapsto p_0 \\ a_1 &\mapsto \neg p_0 \\ a_2 &\mapsto 0 \\ a_3 &\mapsto \neg p_0 \vee p_3\end{aligned}$$

A symbolic simulator can use these functions as inputs without any modification. An algorithm for computing a parametric representation, correctness requirements for the algorithm, and correctness proofs are found in [48]. Similar approaches to the use of parametric encoding are described in [49], where parametrically encoded functional dependencies are used to reduce the complexity of symbolic simulation; and in [50], where parametric representations are used in bounded model checking.

1) Application: Many hardware circuits are designed to function on a defined set of legal inputs and circuit behavior on inputs not in this set does not matter. Additionally, reduction of formal verification complexity is often accomplished by *case splitting*, dividing the verification into multiple cases that when considered together imply the full verification.

Consider a Boolean predicate $P[xs]$ defined over a vector of variables xs, each of which represents a circuit input or an internal signal at a fixed (but possibly different) point of time. If $P[xs]$ describes an environment constraint or one case of a case split, we need only consider the behavior of the circuit under valuations that satisfy $P[xs]$. The desired behavior of the circuit will be expressed as an assertion $\models A[xs] \Rightarrow C[xs]$ over the same variables xs. Because this assertion needs to hold only when $P[xs]$ is true, we wish to establish that $P[xs]$ implies $\models A[xs] \Rightarrow C[xs]$. We will express this implication by writing $P[xs] \models A[xs] \Rightarrow C[xs]$.

A naive approach would do this verification in three steps:

1) Represent the desired restriction as a predicate $P[xs]$.
2) Express the specification as a consequent C and compute $\models A[xs] \Rightarrow C[xs]$ by symbolic circuit simulation.
3) Evaluate $P[xs] \models A[xs] \Rightarrow C[xs]$ by checking that $\phi \models P[xs]$ implies $\phi \models A[xs] \Rightarrow C[xs]$ for all ϕ.

This approach has the disadvantage that it evaluates (symbolically simulates) $\models A[xs] \Rightarrow C[xs]$ for *all* valuations of the variables xs, not just the ones that satisfy $P[xs]$. But in many cases $\models A[xs] \Rightarrow C[xs]$ cannot be computed directly with a symbolic simulator because the complexity is too great.

A better approach is to evaluate $\models A[xs] \Rightarrow C[xs]$ only for valuations that satisfy $P[xs]$. We do this with a parametric representation that encodes $P[xs]$ as a vector of functions over fresh parametric variables. Suppose the function param computes a parameterized functional vector representation

$$Qs = \mathsf{param}(xs, P[xs])$$

Then the implication $P \models A \Rightarrow C$ we wish to prove becomes a simple trajectory assertion $\models A[Qs/xs] \Rightarrow C[Qs/xs]$, in which the original input variables are replaced by the parametric functions.

It is often feasible to compute this encoded trajectory assertion with a symbolic simulator when a direct computation of the trajectory assertion is not possible. The parametric representation is also used in case splitting: each case is characterized by a Boolean predicate that is similarly encoded into the trajectory assertion.

Computing the assertion $\models A[Qs/xs] \Rightarrow C[Qs/xs]$ is equivalent to checking that $P[xs]$ implies $\models A[xs] \Rightarrow C[xs]$ for every assignment of values to the variables xs. A proof is provided in [48]. A side condition requires that $P[xs]$ be satisfiable because no functional vector can encode *false*.

It is often the case that symbolic simulation is infeasible even within the restricted domain of $P[xs]$. Verification complexity can be further reduced by decomposing $P[xs]$ into multiple partitions to represent case splits. In fact, decomposing $P[xs]$ is our primary use of the parametric representation. We illustrate this with the examples in Section IX. Often, different variable orderings are used for different case splits. These orderings can be supplied by the user or, in some cases, determined automatically by the BDD engine.

The strategy of input case-splitting is also the basis of the 'quasi-symbolic simulation' method of Wilson and Dill [51], in which simulation complexity is controlled by representing input cases in an approximate way using constants (called 'symbolic variables') as input values. More generally, simple case-splitting on the two binary values of an input is of course a widespread verification technique.

VI. The FL Programming Language and STE

FL is a strongly-typed, lazy, functional programming language. Syntactically, it borrows heavily from Edinburgh-ML [12]. Semantically, its core is similar to lazy-ML [52]. A distinguishing feature of FL is that a BDD package is integrated with the language's runtime system, with every object of the Boolean type `bool` being represented as a BDD.[4]

The FL language lies at the heart of Forte. Through its embedded BDD package and primitive or defined functional entry-points, it provides a flexible interface for invoking and orchestrating model checking runs. It is also used as an extensible 'macro language' for expressing specifications, which are therefore human-readable but when executed compute efficiently checkable properties in a low-level temporal logic. Finally, it provides the control language for Forte's theorem prover and—through the concept of *lifted FL* [21]—the primitive syntax of its higher order logic.

Trajectory formulas are implemented in FL as lists of *five-tuples*. Each five-tuple specifies an assertion about a single signal and contains the following elements:

- a Boolean guard specifying when the assertion is active
- the name of the signal (a string)
- a Boolean value to assert on the signal
- a start time (integer)
- an end time (integer)

A list of five-tuples represents the conjunction of the individual assertions. The representation of temporal information by intervals (start and end times) is more convenient in practice than the use of a "next time" operator.

The five-tuple representation adds no logical expressiveness. Its major advantage is that it represents formulas using standard data types in FL and similar languages. It is particularly important that the guard and value fields in each five-tuple use FL's BDD representation of Boolean propositions. Users therefore have the complete freedom of a general-purpose functional programming language in which to write both temporal and value properties, facilitating concise and readable specifications.

The orthogonality of the temporal aspect (the two time fields) of the five-tuples from the data computation (the guard and value fields) has a number of positive ramifications. Although trajectory formulas are not generally executable, the individual fields are executable. For example, users can evaluate the data aspect of their specifications simply by evaluating the FL function that computes the intended result. Standard rewrite rules and decision procedures over Booleans can be applied to the guard and data fields. Rewrite rules and decision procedures for integers can be applied to the temporal fields.

Circuits are represented in FL as objects of a special built-in type `fsm`. An `fsm` is a directed graph where nodes correspond to gates and edges to connections between gates. Nodes in the graph are annotated with their stimulus functions, delays, and other attributes. A number of built-in functions support structural queries, breadth-first or depth-first traversal, and finding out dynamic information about circuit nodes (e.g. waveforms). This functionality is available through a graphical user interface and as primitive functions in FL, enabling users to program their own custom queries. This has proved extremely valuable for circuit analysis and debugging. Objects of type

[4]Strictly speaking, the type of these objects should be something like `env` → `bool`, where `env` is an interpretation of the variables used in the BDD. However, for convenience the global environment is kept implicit and the type abbreviated to `bool`.

fsm can arise from a number of sources. Translators have been written from Intel's gate-level and schematic-level netlist formats to a special data representation which can then be loaded by FL. Furthermore, primitives available in FL allow the interactive, programmable construction of fsms.

Like the circuit manipulation functions, symbolic trajectory evaluation is available as a primitive function in FL:

$$\texttt{STE} :: \texttt{fsm} \rightarrow \texttt{tf} \rightarrow \texttt{tf} \rightarrow \texttt{bool}$$

STE takes three arguments. The first is an fsm. The second and third are an antecedent A and a consequent C, both temporal formulas represented as lists of five-tuples. The result returned is a Boolean formula that is the weakest condition under which $A \Rightarrow C$ (Section IV).

In our methodology STE is usually invoked in a stylized way.

$$\texttt{STE ckt (A vs) (C S vs)}$$

Here, ckt is the circuit under consideration. The antecedent A captures the protocols and timing required at the circuit inputs, and is an FL function parameterized by vs, a list of symbolic values to use as input data to the circuit. The consequent C formalizes the protocols and timing required at the circuit outputs, and is parameterized by a *functional specification* S as well as the list of symbolic values vs. We call the functions A and C, taken together, a *circuit API*.[5] S serves as a specification of the intended function of the circuit, independent of timing: for a list of symbolic input variables vs, S vs computes the symbolic values expected at the circuit's outputs.

We have developed a wide range of FL functions to ease the task of writing circuit APIs and functional specifications. For example, in capturing input/output protocols it is often convenient to specify event timing relative to a given clock signal, rather than in absolute time ticks. Such a specification style is supported by a library of temporal abstraction and clocking functions. As another example, functional specifications must often model arithmetic and logical computations on bit vectors. Common bit-vector operations are supported by another library of FL functions.

In addition to FL's role as a specification language, FL also serves as a scripting language for invoking the STE algorithm and orchestrating large verifications. FL is used to control BDD variable orderings, manage case splits, and conduct multi-run verifications. Thus, the outcome of a verification project includes, in addition to a set of specifications, a suite of FL scripts to effect the verification. The existence and maintainability of the script is crucially important for maintaining productivity and repeatability in a live design environment, where RTL code changes from day to day.

VII. Theorem Proving

Our higher-order logic theorem prover, ThmTac, provides a simple but principled way to justify the composition of model checking results and manage the proof of higher-level properties. Composing model checking results allows us to prove properties that are beyond the verification capacity of model checkers. ThmTac's logic and inference rules allow us to state (and prove) specifications that are beyond the expressive power of the model checker's specification language.

ThmTac is implemented in the LCF style [53], with the representation of theorems protected by abstract datatypes so that only prescribed operations can yield theorems. In LCF and its successors (including the widely-used HOL system [12]), theorems can be constructed only through application of a core set of axioms and primitive rules of inference.

However, a major goal in designing ThmTac was to provide a seamless transition between model checking, in which we *execute* FL functions, and theorem proving, in which we *reason about* the syntax of FL functions. A mechanism that we refer to as *lifted-FL* enables us to use FL as both the object and meta languages of our proof tool; consequently, certain goals in the theorem prover can be proved simply by evaluating them. Such 'proof by evaluation' extends the notion of proof beyond that of LCF.

A second design goal for ThmTac was a level of automation sufficient to allow routine use of ThmTac by users who are not expert logicians. To that end, we follow the lead of PVS [54] and Nuprl [55] and allow theorems to be generated by trusted decision procedures (for example, a decision procedure for linear arithmetic) as well as the core set of axioms and rules.

In the following subsections, we provide an introduction to lifted-FL, an overview of theorem-proving in higher-order logic, describe the architecture of our theorem prover, and comment on the soundness of our implementation.

A. Lifted-FL

Ordinary programming languages operate on the *values* of expressions and not their structure. For example, the value of $3+4$ is 7, and we cannot distinguish $3+4$ from $8-1$ by examining their values. An equality test in a programming language is a therefore test of *semantic equality*. In logic, the equality symbol also forms an assertion about semantic equality (e.g., $\texttt{a}+\texttt{b} = \texttt{b}+\texttt{a}$), but the primitive inference rule for equality of two terms is a test of *syntactic* equality (e.g., $\texttt{a} = \texttt{a}$). The power of theorem proving comes from the ability to use syntactic manipulation to prove the semantic equality of expressions that cannot easily be evaluated.

FL expressions have the following syntax:

e	::=	v	- variable
	|	c	- constant
	|	$e_1\ e_2$	- function application
	|	$\lambda v.\ e$	- abstraction
	|	` $e$ `	- lifting

The first four syntactic forms—variables, constants, application, and abstraction—are those of the applied lambda calculus, the term language for higher-order logic theorem provers like HOL [12]. In addition, we provide the capability to 'lift' an FL expression by enclosing it in backquotes (e.g., `` `3+4` ``). Lifting an FL expression makes its abstract syntax tree available for other functions to examine, manipulate, and evaluate. The expression will have been typechecked, and its type is also made available for manipulation.

[5]The term API, *Application Programmer Interface*, is borrowed from software engineering.

The objective of lifted-FL is to enable syntactic reasoning about FL programs to be conducted within FL itself. Lifted-FL is similar in spirit to Lisp's quotation mechanism, with the important difference that FL is statically typed, while Lisp is dynamically typed. Parsing a lifted-FL expression gives two representations of the expression: a combinator graph[6] for evaluation purposes, and an abstract syntax tree representing the text of the expression. This link between the abstract syntax and combinator graphs allows lifted-FL expressions to be evaluated as efficiently as normal FL code. An evaluation function `eval` takes a lifted-FL expression, evaluates it, and returns the result as an FL value.

To support theorem proving, lifted-FL includes some features that are not a part of regular FL. Lifted-FL expressions can contain free variables, but evaluating a lifted-FL term raises an exception if the associated expression contains any free variables. Existential and universal quantifiers are implemented as functions that raise exceptions when evaluated, with special axioms provided for reasoning about them in ThmTac.

B. Theorem proving in higher-order logic

The version of higher order logic supported by ThmTac is based on Church's formulation of simple type theory [57]. The formulas of the logic are terms lifted from FL expressions of type `bool`. For the purposes of this paper, the logic can be viewed as a typed extension of the conventional syntax of predicate calculus in which functions may be 'curried' and one may quantify over functions.

The notation is illustrated by the theorem shown below.

$$\vdash \exists f.\forall x.\, f(\mathsf{g}\ x) = x$$

This says that there exists a left inverse of the function g. The quantified variable f in this formula ranges over functions. We adopt the convention that italic identifiers (e.g. x, x_1, F) are variables and sans serif identifiers (e.g. a, F, Tau) are constants. The constants T and F denote Boolean *true* and *false*, respectively. The turnstile symbol '$\vdash$' indicates that the formula that follows is a formal theorem of the logic.

The most primitive notion of formal proof is one in which rules of inference are simply applied in sequence to axioms and previously-proved theorems until the desired theorem is obtained. This is often not a feasible way of finding a proof, since the exact sequence of inferences required—or even the first inference required—is rarely known in advance.

A more promising and natural approach is to set about discovering a proof by working backward from the statement to be proved (called a *goal*) to previously proved theorems that imply it. This is the *backward proof* style, in which the search for a proof is the activity of exploring possible strategies for achieving a goal. For example, one possible approach to proving a conjunctive formula $P \wedge Q$ is to break this goal down into the two separate subgoals of proving P and proving Q. Likewise, one may seek to prove an implication $\forall x.P[x] \supset Q[x]$ by reducing this to the subgoal of proving $Q[x]$ under the assumption $P[x]$ for arbitrary x.

ThmTac, like LCF and HOL, supports the backward style of proof by means of FL functions called *tactics*. In theorem provers adhering to the LCF philosophy, tactics are used to break goals down into increasingly simple subgoals, until the subgoals obtained are axioms or theorems already proved. Conventional proofs in higher-order logic proof systems are operational, in that tactics describe how to move from one proof step to the next, but do not describe what the next proof step is. ThmTac also provides declarative proof tactics, which describe what the next proof step should be and give only minimal information on how to prove that the next step follows from the current step. Until recently, declarative proofs were rare in mechanized higher-order logic [58], [59], [60].

In addition to tactics, FL allows one to implement functions (called *tacticals*) that combine elementary tactics together into more complex ones. This allows the user to build composite tactics that fully decompose a conjecture into immediately-provable subgoals, and hence can be executed to generate a complete proof. In practice, these monolithic, composite tactics are the main products of the theorem-proving activity.

C. Theorem proving tools

One of the main tools for higher order logic proof in Forte is proof by evaluation. This is integrated into the tactic mechanism through a primitive tactic, `Eval_tac`, defined using the FL `eval` function. This tactic evaluates the conclusion of a sequent and solves the goal if the result of evaluation is true. For example, a goal stating that an STE model-checking run succeeds can be solved by running the STE algorithm.

This mechanism makes use of the fact that ThmTac's term language is lifted-FL, hence certain proof goals (in the logic) can be solved simply by evaluating them (in FL). Since Booleans are built into FL as BDDs, `Eval_tac` also provides an efficient decision procedure for quantified Boolean formulas.

Proof goals in ThmTac can also be solved using a number of built-in decision procedures and heuristics:

- `Trivial`, a heuristic that checks for a number of obvious conditions, such as an assumption that is also the goal, or two assumptions that contradict one other,
- `IntSimplex`, a decision procedure that combines linear programming and BDDs to solve goals that combine Boolean and integer reasoning,
- `OneTab`, a first-order, Prolog-style decision procedure that uses backward chaining to find instantiations for monomorphic first-order quantified variables,
- `Rewrite`, a higher-order conditional rewriter, and
- `Qed`, which combines a large set of standard rewrite rules with the other tactics for solving goals.

Some of these procedures also integrate FL evaluation into proof, notably the rewriter. Although our primary intention with lifted-FL was to provide for proof by evaluation of STE runs, FL evaluation can also be applied to general expressions. This is easier and more efficient than applying libraries of rewrite rules for each of the different functions to be evaluated.

[6]Combinator graphs are a particular representation of compiled functional programs in which variable names have been 'compiled away'. Running a compiled program in this form consists in applying certain graph-transforming reductions to the combinator graph. See [56] for an introduction to this topic.

`Eval_rw` is a rewrite that evaluates a term or sub-term of a formula under consideration, and substitutes the result in for the original term.

D. The implementation of ThmTac

The implementation of ThmTac is based loosely on that of LCF [53] and HOL [12], with a trusted core set of axioms and inference rules protected by abstract datatypes. Proof goals are recorded as *sequents*; each sequent is in turn a pair (Γ, t) where Γ is a finite set of formulas called the *assumptions* and t is a formula called the *conclusion*.

Inside the trusted core, a sequent is represented by a list of 'clauses' labeled by strings. The FL type definition is

$$\texttt{Sequent} \overset{type}{\equiv} \texttt{SEQUENT}\ (\texttt{string\#term})\ \texttt{list};$$

where # is FL's pair (cross-product) type constructor and `term` is the type of lifted-FL expressions. Each clause is just a lifted-FL term (expected to be of type `bool`) and there is one clause for each assumption of the sequent and one for the negation of its conclusion. Labeling each clause with a string gives users a robust method for identifying particular assumptions or conclusions in a proof script. Tactics are implemented as functions that map a sequent to a list of sequents, with each element of the result list representing a subgoal. When a tactic returns an empty sequent list, it means that the tactic solved the goal.

Finally, as in LCF, an abstract datatype is used to distinguish theorems from arbitrary terms.

$$\texttt{Theorem} \overset{type}{\equiv} \texttt{THEOREM term};$$

$$\texttt{Prove} :: \texttt{term} \rightarrow \texttt{Tactic} \rightarrow \texttt{Theorem}$$

The principal interface to ThmTac is the function `Prove`, which takes a Boolean term `` `b` `` stating a proposition to be proved and a tactic, attempts to use the tactic to construct a proof that `b = T` and, if the construction succeeds, returns `THEOREM` `` `b` `` (also written $\vdash$ `` `b` ``).

E. Soundness

Proof systems often put a high priority on mathematical purity, in the sense that every theorem is derived solely from the primitive inference rules and axioms of the logic. The motivation is to ensure the system is *sound*—i.e. it can not be used to prove a false statement. In reality, no formal verification system can provide an absolute guarantee of soundness. And in the world of industrial microprocessor design formal verification must compete for resources against other validation techniques amidst many other demanding design goals, such as timeliness to market, and performance, area, and power requirements.

In designing ThmTac, our goal was to strike a balance between soundness and productivity—both in system building and verification. With respect to soundness, we focused our efforts on preventing users from inadvertently proving false statements, but did not exert unjustifiable effort in protecting against adversarial users (users who intend to prove false statements).

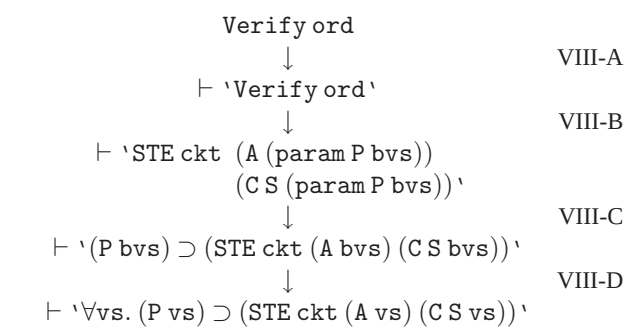


Fig. 3. Road map for combining STE and ThmTac

The FL feature most likely to lead to unsoundness is recursion. For example, the definition `letrec x = NOT x;` can be used to introduce a contradiction, from which anything at all can be proved. We rely on the user to ensure that recursive definitions used in hardware specifications are terminating. In practice, our experince is that this is sufficient. But if higher assurance is needed, then there do exist automated tools, such as Slind's TFL [61], that could also be used to prove termination of FL functions with reasonable effort.

VIII. Combining Theorem-Proving and STE

Fig. 3 illustrates the process in which STE and ThmTac reasoning are combined to verify a typical property. Our philosophy is not the top-down idea of model checking as a decision procedure, but of knitting together model checking runs bottom-up or transforming model checking goals sideways into forms that are easier to solve.

The process begins with a verification script that invokes various functions written in FL. Let us suppose this script is just '`Verify ord`', where `Verify` is some FL function that we have written. Invocation of this script will install the BDD variable ordering given by the `ord` argument and then invoke STE on the circuit of interest with some trajectory assertion we have chosen. The details of the script are not important here, and the script need not have been written with theorem proving in mind.

The first step, discussed in Section VIII-A, uses ThmTac's capacity for proof by evaluation to assert the truth of the proof script. The second step, discussed in Section VIII-B, uses ThmTac's ability to reason about FL programs to unfold the proof script, revealing the underlying call to STE. In this case, the call to STE makes use of the parametric representation (Section V) to make model-checking tractable. Section VIII-C describes how special-purpose axioms are used to distill the logical content from such optimized model-checking calls.

In the last step (Section VIII-D) BDD variables and quantifiers in the property are replaced with their counterparts in higher-order logic. This brings the entire property within the scope of ThmTac's deductive apparatus, which in turn enables the large-scale verification strategies used in the case studies in Section IX. Sections VIII-E and VIII-F discuss the support for logical and algorithmic reasoning provided by ThmTac.

A. Proof by Evaluation

In Forte, the entry point to theorem proving is through evaluation of model checking runs. As discussed above, suppose that the FL function

$$\texttt{Verify} :: (\texttt{string list}) \rightarrow \texttt{bool}$$

has been designed to execute some STE model checking run of interest. Its argument is a list of strings, specifying a BDD variable ordering to be used in the verification. If running `Verify ord` evaluates to `T`, then proof by evaluation justifies introduction of the corresponding ThmTac theorem

$$\vdash \text{`}\texttt{Verify ord}\text{`}$$

In this way, proof by evaluation provides a smooth transition from STE model checking to theorem proving. To get our theorem, we just lift the *source text* of the model-checking code already developed. This imposes no extra tax on the model-checking user.

The ease of importing model checking results into the theorem prover has a dramatic impact on the user's view of the system. For example, when proof by evaluation is invoked as a tactic and fails, it generates a counter-example in the form of a residual. Since theorem proving, model checking, and debugging are conducted in the same environment, the user can debug the counter-example using the debugging aids for STE available in Forte.

B. Reasoning about scripting code

ThmTac supports reasoning about FL programs, through axioms about FL constants, tactics for step-by-step evaluation, unfolding of user definitions, and facilities for rewriting and partial evaluation. This support is essential for manipulating proof scripts in order to expose the underlying calls to the model checker.

Suppose the definition of `Verify` were

```
Verify n ≜
  var_order n
    fseq
  STE ckt (A (param P bvs))
          (C S (param P bvs))
```

When executed, the FL function `var_order` installs the specified order in Forte's underlying BDD manager. The infix sequencing operator `fseq` fully evaluates its left argument, ignores the result, and returns its right argument. The function `param` was explained in Section V-B; its use in ThmTac will be explained in the next section.

Unfolding the definition of `Verify` and performing one step of evaluation (in technical terms, β reduction) transforms the theorem

$$\vdash \text{`}\texttt{Verify ord}\text{`}$$

to

```
⊢ `var_order ord
    fseq
  STE ckt (A (param P bvs))
          (C S (param P bvs))`
```

Both these transformations are supported by ThmTac's rewriting engine. Next, employing axioms about the FL function `fseq` we can prove the theorem

```
⊢ `STE ckt (A (param P bvs))
           (C S (param P bvs))`
```

C. Model checking optimizations and transformations

Section V introduced techniques (the parametric representation and various kinds of weakening) for extending the reach of STE model checking. These are transformations that allow us to transform the proof goal without altering its logical content. In particular, they are targeted to transform infeasible model-checking computations to feasible ones. The fact that these transformations are used is recorded in the formal proof.

The axiom that justifies the use of the parametric representation is `Param_ax`:

```
⊢ `∀ckt A C F P xs.
   (∃ys. P ys) ⊃
   ((STE ckt (A (param P xs))
             (C S (param P xs)))
      ≡
    (P xs) ⊃ (STE ckt (A xs) (C S xs)))`
```

The function call `param P xs` computes a parametric substitution of the BDD vector `xs` with respect to `P` (in the sense of Section V). In other words, using parameterization, the predicate `P` can be encoded inside the STE assertion.

Assuming we can prove the side condition $\exists$`ys. P ys` holds, application of `Param_ax` allows us to go from

```
⊢ `STE ckt (A (param P bvs))
           (C S (param P bvs))`
```

to

$$\vdash \text{`}(\texttt{P bvs}) \supset (\texttt{STE ckt (A bvs) (C S bvs)})\text{`}$$

D. Transforming BDD variables to logical variables

The final step in bridging the semantic gap between theorem proving and model checking is the transformation from formulas containing BDD variables and quantifiers (the language of model checking) to formulas containing logical variables and quantifiers (the ordinary domain of discourse in higher-order logic). Replacing BDD variables and quantifiers with logical ones makes available the full range of rules for quantifier reasoning in higher-order logic. This replacement allows us to regard the theorem

$$\vdash \text{`}(\texttt{P bvs}) \supset (\texttt{STE ckt (A bvs) (C S bvs)})\text{`}$$

as equivalent to

$$\vdash \text{`}\forall\texttt{vs}.\ (\texttt{P vs}) \supset (\texttt{STE ckt (A vs) (C S vs)})\text{`}$$

The former property is stated in terms of BDD variables that are implicitly universally quantified. The latter property is stated purely in the higher-order logic supported by ThmTac (with the BDD variables `bvs` replaced by the logical variables `vs`) and universal quantification is explicit.

It is also useful to cross this gap in the opposite direction; we could, for example, transform ⊢ ‘∀x. P x‘ to

$$\vdash \text{`}\forall \texttt{x}.\ \texttt{P}\ \texttt{x}\text{`}$$

to

$$\vdash \text{`}\texttt{QuantForall "x" (P(variable "x"))}\text{`}$$

where `variable` is a built-in FL function that creates a BDD variable (in this case, named by the string `"x"`) and `QuantForall` is a BDD operation that universally quantifies the BDD variable `"x"` in the BDD resulting from evaluating `P(variable "x")`. If `P` contains only Boolean operations, the latter goal can be solved by evaluation.

Replacing higher-order logic variables and quantifiers with their BDD counterparts is not as simple as it seems at first glance, as there are performance and soundness issues. If ThmTac were to provide a new and unique name for the BDD variable, it would not be placed in an optimal location in the all-important BDD variable order defined by the user. Additionally, increasing the number of BDD variables globally active in the system slows down some BDD operations. Thus, for performance reasons, the user needs to provide the variable name. However, if the user inadvertently provides a variable name that is already used elsewhere in the term, a free BDD variable can become bound unintentionally and the proof rendered unsound. Thus, ThmTac has the burden of making sure that the name provided by the user is truly a fresh variable.

The process of replacing term quantifiers and variables with BDD quantifiers and variables and then evaluating the goal is implemented by the tactic `BddInstEval_tac`. The user provides the name of fresh BDD variable (say `"y"`). The tactic first checks that this variable has not yet been used in the proof. If the variable is fresh, `BddInstEval_tac` replaces the term quantifier $\forall x$ with the BDD quantifier `QuantForall "y"`, instantiates x in $P(x)$ with `variable "y"`, and then applies `Eval_tac`.

E. Reasoning about trajectory assertions in ThmTac

Section IX presents some case studies that illustrate some strategies for large-scale verifications. The various elements of these strategies are supported by reasoning in ThmTac; indeed, it is at this level that the overall verifications are logically orchestrated.

Complexity reduction strategies like input case-splitting, induction, and others can be justified using the ordinary rules of higher-order logic. For example, we could manage a case-splitting strategy that takes us from a family of assertions indexed by `i`:

$$\vdash \text{`}\forall \texttt{vs}.\ (\texttt{Q}_\texttt{i}\ \texttt{vs}) \supset (\texttt{STE ckt}\ (\texttt{A vs})\ (\texttt{C S vs}))\text{`}$$

to the single assertion

$$\vdash \text{`}\forall \texttt{vs}.\ (\texttt{Q vs}) \supset (\texttt{STE ckt}\ (\texttt{A vs})\ (\texttt{C S vs}))\text{`}$$

The reasoning required, including proof of the side condition

$$\forall \texttt{vs}.\ (\texttt{Q vs}) \supset \bigvee_{\texttt{i}=1}^{\texttt{N}} (\texttt{Q}_\texttt{i}\ \texttt{vs})$$

is carried out using ThmTac's deduction system.

$$\vdash \text{`}\forall \texttt{ckt A C1 C2}.\ (\texttt{STE ckt A C1}) \wedge (\texttt{STE ckt A C2}) \supset \texttt{STE ckt A (C1 and C2)}\text{`}$$

Fig. 4. STE conjunction inference rule

A number of inference rules that support compositional reasoning were presented in Section IV. From the logical viewpoint these provide an axiomatic characterization of STE's behavior as a functional program. Use of the STE inference rules can substantially reduce the complexity of underlying STE runs.

Fig. 4 shows the inference rule for conjunction. Additional trajectory evaluation rules include pre-condition strengthening, post-condition weakening, transitivity, and case-splitting [39], [42]. After using the inference rules to decompose a proof obligation into a set of smaller STE goals, we use `Eval_tac` to carry out the individual STE runs. For example, we could use the inference rule for conjunction, along with standard deduction rules of higher-order logic, to combine the two trajectory assertions

$$\vdash \text{`}\forall \texttt{vs}.\ (\texttt{P vs}) \supset (\texttt{STE ckt}\ (\texttt{A vs})\ (\texttt{C}_1\ \texttt{S vs}))\text{`}$$
$$\vdash \text{`}\forall \texttt{vs}.\ (\texttt{P vs}) \supset (\texttt{STE ckt}\ (\texttt{A vs})\ (\texttt{C}_2\ \texttt{S vs}))\text{`}$$

to yield the composite assertion

$$\vdash \text{`}\forall \texttt{vs}.\ (\texttt{P vs}) \supset (\texttt{STE ckt}\ (\texttt{A vs})\ (\texttt{C}_1\ \texttt{S vs and C}_2\ \texttt{S vs}))\text{`}$$

F. Reasoning About Functional Specifications

So far we have paid little attention to the functional specification `S` in the consequent of the formula:

$$\vdash \text{`}\forall \texttt{vs}.\ (\texttt{P vs}) \supset (\texttt{STE ckt}\ (\texttt{A vs})\ (\texttt{C S vs}))\text{`}$$

Since `S` is itself merely an FL program, we can use ThmTac to reason about `S` in isolation, proving a theorem of the form

$$\vdash \text{`}\forall \texttt{vs}.\ \texttt{Spec vs}\ (\texttt{S vs})\text{`}$$

Here, `Spec` is some higher-level property relating the arguments and results of `S`. For example, if `S` describes the bit-level computation performed by a floating-point adder, `Spec` might state that the result of the computation conforms to the IEEE standard for floating-point arithmetic. We can now prove a top-level correctness property in higher-order logic that combines temporal and high-level correctness:

$$\vdash \text{`}\forall \texttt{vs}.\ (\texttt{STE ckt}\ (\texttt{A vs})\ (\texttt{C S vs})) \wedge (\texttt{Spec vs}\ (\texttt{S vs}))\text{`}$$

We have used this combination of trajectory evaluation and theorem proving in several large verification efforts. Published accounts include a variety of floating-point circuits and an IA-32 instruction-length decoder [3], [23], [42].

IX. Verification Case Studies

In Section I, we detailed aspects of contemporary formal verification problems that pose significant problems for conventional verification approaches. We then described symbolic trajectory evaluation, STE optimizations, the FL programming language, lifted-FL, and the combination of STE and theorem proving. The presentation has aimed to illustrate our system-building philosophy and its embodiment in Forte.

The greatest challenge in formal verification is computational complexity. This is addressed primarily by decomposing a given problem into multiple smaller problems. But for a decomposition to be effective, the decomposition itself must be manageable, as well as result in sub-problems that have manageable complexity. One of the requirements of an effective verification system is to provide good support for this kind of activity.

In this section, we describe three of the verification studies we have completed with the Forte system. Our emphasis is on the synergy between the typical problems encountered in industrial verification and the capabilities in Forte. We will discuss a branch-target buffer, a floating-point adder, and an instruction-length decoder. Space does not permit a complete exposition of any example, but we aim to provide a good sense of why the verification of these circuits is difficult and how the different aspects of Forte combine to make verification feasible in these instances.

The adder and length-decoder represent two of the most-complex hardware verification results to date. Both have complex functional specifications and require extensive tool and system support to manage verification complexity.

A. Branch-target buffer

Our first verification example is a branch-target buffer, a functional block that contains an embedded memory array. This example illustrates the use of symbolic indexing in the verification of a memory array and the application of STE inference rules to support a structural decomposition strategy. Structural decomposition, in which individual pieces of the circuit are verified independently, is perhaps the most common approach to making verification tractable. Structural decomposition is a powerful technique, because it can be applied repeatedly until each sub-component of the circuit is small enough to handle automatically with STE. The results of the sub-problems are then combined using STE inference rules, as enumerated in Section IV, which are implemented as core tactics in ThmTac.

The branch-target buffer (BTB), as shown in Fig. 5, interfaces primarily to an instruction-fetch unit (IFU) and a branch-address calculator (BAC). The IFU sends the BTB a series of instruction addresses (that is, program counters) that index the BTB's internal memory, which maintains information on the history of branches. Based on the branch history found in its memory, the BTB makes a prediction of whether the cache line containing each instruction also contains a branch instruction and, if so, whether the branch is taken or not taken. In the case of a taken branch, the predicted target address is sent back to the IFU, which begins fetching instructions from the new address. The BTB's interface with the BAC is used to update the BTB's stored branch history with information on the true sense of the branches. This is used to adaptively predict the direction of future branches.

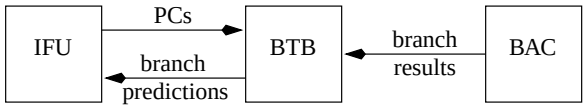


Fig. 5. BTB Interfaces

We wish to verify that the branch predictions are made and the branch history updated correctly according to some given prediction algorithm. The BTB we verified uses Yeh's two-level adaptive branch prediction algorithm [62]. Yeh's algorithm is a heuristic technique that has proven statistically effective for branch prediction, and there is no more abstract specification for the BTB. For this example, we focus on verifying the behavior of the branch translation pipeline rather than the branch update pipeline.

We verified five properties of the prediction pipeline:

1) For a given incoming program counter, the correct line from the BTB memory array is written into a temporary register.
2) The taken branches in the selected line are correctly marked.
3) The target address of the taken branch with the least offset relative to the incoming program counter (that is, the 'next' taken branch) is correctly extracted.
4) The updated (speculative) branch history is computed correctly.
5) The updated branch history is written back into the BTB memory array.

We will briefly describe two of these properties. Property 1 checks that for a given instruction address, the correct line is retrieved from the BTB. The number of bits ($128 \times 200 + > 25000$) in the BTB memory array is far too large to verify directly. Instead, we encode the property using symbolic indexing. This requires only $\log_2 128 + 200 = 208$ BDD variables.

Property 4 states that the BTB logic correctly updates the (speculative) branch prediction. To do this, the BTB must read the appropriate branch information out of the array (Property 1) and update it in a very straightforward way. In fact, the antecedent of Property 4 is largely implied by Property 1.

We use the transitivity inference rule in ThmTac to compose Properties 1 and 4. The other properties were also composed using STE inference rules. Our final, high-level property states that the BTB always produces the correct prediction (according to the algorithm) for an arbitrary BTB initial state and arbitrary instruction address.

While the BTB memory we verified is small by memory standards (128 lines by approximately 200 bits per line) it is very large by formal verification standards. In addition to its large embedded memory, the BTB also poses a verification challenge because it implements a large, sequential computation over several pipe stages. We overcame these obstacles by exploiting two key aspects of Forte.

- First, we used ThmTac to mechanically verify the decomposition of the overall correctness statement for the BTB into properties that were within the capacity of STE. Our proof required both general purpose reasoning at the algorithmic level and STE-specific reasoning to combine the low-level results. Using a model checker alone, one faces the choice of abstracting the algorithm (difficult, and it is possible to inadvertently abstract away behavior that is significant) or chipping away at the corners of the hardware, not verifying the really important properties at all.
- Second, we used symbolic indexing to reduce the complexity of verifying the embedded memory array. Pandey and Bryant [45] have also applied symbolic indexing in the verification of a variety of memory arrays, including content-addressable memories (CAMs). If the array were considered in isolation, the complexity of verifying it could be mitigated by model reduction via a symmetry argument [63], [64]. However, the efficacy of the branch prediction algorithm relied upon the array being of a certain size. Also, as pointed out by Bhadra *et al.*, high-speed memories in isolation often do not exhibit the clean symmetries that might be expected [28]. For this reason Bhadra *et al.*, like us, relied on STE and symbolic indexing.

It took approximately two engineer-months to understand the BTB algorithm and implementation, develop a high-level specification, and verify Properties 1 through 5 of the prediction pipeline. Mechanical verification that Properties 1 through 5 together imply the high-level specification took a further month, using an early prototype of the ThmTac tool. A subtle mismatch between the properties—manifested by the failure of the STE reasoning tactics—was detected by ThmTac, underscoring the importance of verifying the decomposition.

B. Floating-point adder

Our next example is the verification of a floating-point adder. There were two primary challenges in verifying the adder. First, there is a large gap in abstraction between the high-level specification (that specifies the relationship between the real-valued operands and result of their addition) and the circuit design (that describes a computation on bit vectors). To address this challenge we developed specifications at two levels of abstraction, ranging from a top-level specification that captures the essence of IEEE Standard 754-1985 [65] to a bit-level reference model that is close to the algorithm performed by the hardware. It is essential that these specifications are written in FL enabling them to be linked by formal proof in the ThmTac theorem prover. Fig. 6 shows the structure of the verification.

The second challenge—dealing with the sheer complexity of the algorithm and its hardware implementation—arises in the proof of equivalence between the FL reference model and the RTL implementation. This complexity is made evident by an explosion in the size of the BDDs used in the verification. To address the complexity challenge, we used a decomposition approach based on *data-space partitioning*, where the input

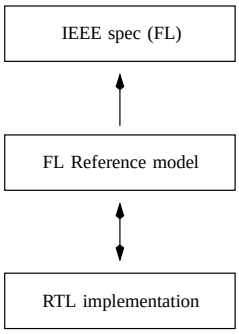


Fig. 6. FADD Verification

data space is broken into a number of sets. Each set in the data space is treated as a separate case for verification. Each case is represented as a Boolean predicate and then encoded on circuit nodes with the parametric representation, as discussed in Section V. A side condition requires that the case splits completely cover the input state space. This can often be verified directly with BDDs; more complex cases require justification within ThmTac.

Data-space partitioning makes it easy to compose verification results (only simple propositional reasoning is required) and is fairly robust to changes in the circuit's internal implementation. Finding a set of cases that significantly reduces the size of the BDDs requires some understanding of the *algorithm* that the circuit implements, but usually requires only minimal knowledge of the internal structure, e.g. signal names and timing. In contrast, structural decomposition as used in the BTB verification can run into significant problems in practice. Structural decomposition requires detailed knowledge of the internal signals, timing, and functionality of the circuit. On large, complex circuits, it is often difficult to identify clean partitions, and the functionality of sub-circuits can become hard to specify cleanly.

1) Functional specification: The floating-point adder (FADD) we verified is compliant with the IEEE Standard 754-1985 [65]. A floating-point number is represented with three fields: a *sign*, a *significand*, and an *exponent*. The significand is usually represented in hardware as a *fractional part* field with an implicit 1 added, i.e.

$$\text{sig} = 1.\text{fpfrac} \tag{5}$$

In the IEEE standard, the exponent is *biased*. A fixed bias is selected such that the sum of the bias and the number being represented will always be non-negative. An exponent is represented by first adding it to the bias and then encoding the sum as an ordinary unsigned number. The number represented in the floating-point format is derived from the machine representation as shown:

$$\texttt{number} = (-1)^{\text{sign}} * \text{sig} * 2^{\text{fpexp}-\text{bias}} \tag{6}$$

More detail on floating-point arithmetic is found in Appendix A of [66].

In floating-point addition and subtraction, there is a difference between true and general operators. *True addition* occurs

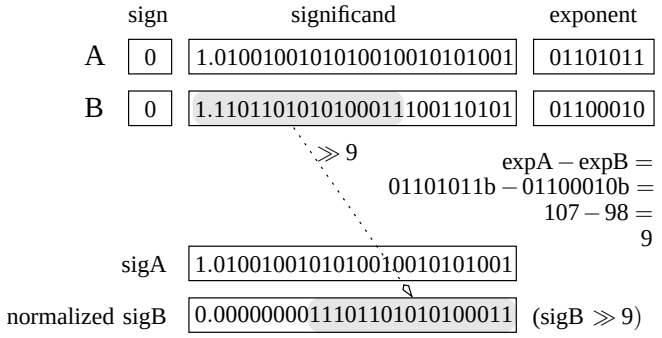


Fig. 7. Normalization of smaller significand for floating-point add

when adding operands with the same sign and when subtracting operands with opposite signs. Similarly, *true subtraction* occurs when adding operands with opposite signs and when subtracting operands with the same sign.

The arithmetic portion of our specification was based upon a textbook algorithm by Feldman and Retter [67]. Specifications for flags and special cases were based on IEEE Standard 754 and internal Intel microarchitecture-specific documentation.

Our functional specification was coded in FL, and served as a bridge for the abstraction gap between the IEEE specification and the FADD circuit. A large ThmTac proof established that the FL specification is faithful to the IEEE specification. Proving that the FADD circuit is faithful to the FL specification required decomposition.

2) Data-space decomposition: The hardware that implements the floating-point add algorithm is quite complex. We limit this discussion to only certain aspects. The first step in the algorithm is to compare the exponent values of the two operands. If the exponents differ by more than the width of the significand, then the significands do not overlap. The result is (modulo rounding) the larger of the two operands. However, if the exponent difference is less than the width of the significand, the smaller operand must be normalized (shifted) for adding with the larger operand. The amount of this shift is variable, and depends on the difference between the two exponents. This is illustrated in Fig. 7, where exponent A is larger than exponent B by 9. As a result, the smaller significand (sigB) is shifted right by 9 bits to be normalized with respect to the significand of A.

The variable-shift of the smaller significand causes BDD explosion during the addition operation. Even the functionality of a single-precision floating-point adder is not representable with BDDs. The explosion is not a result of a non-optimal logic design, but of the *algorithm* for floating-point add. So, in addition to the circuit result not being representable, the specification of correct behavior is also not representable.

The BDD complexity issue for the FADD algorithm is addressed by case splitting on the *difference* between the exponents of the two operands. This scheme results in a constant input to the shifter that, in turn, results in manageable BDD sizes in the adder itself. Many different combinations of exponents will result in the same difference, e.g. $13-1=12$, $14-2=12$, $60016-60004=12$. We represent each constant shift amount by a symbolic predicate encoded parametrically on the adder inputs. Fig. 8 shows the outline of the case splits used. The first case split is on which of the two exponents is greater. At the extremes, the exponents differ by more than the width of the significand. All of these possibilities can be handled by a single case split. The regions in the middle are where the aligned significands overlap. One case is used for each amount that the smaller operand must be shifted to the right.

The power of this approach is that each case split uses a different BDD variable ordering that is optimized for the specific case. The cases in Fig. 8 are illustrated with a tree diagram. Of course, a data-space decomposition approach based on the parametric representation allows arbitrary collections of cases. This is advantageous because it allows great flexibility in structuring the verification, which in turn allows more readable and robust verification scripts.

We estimate that it took 3 engineer-months for the STE part of the FADD verification, with the caveat that some of this time was spent developing the methodology and inventing new techniques (e.g. dynamic weakening). This does not include the time taken to do the theorem proving part that links the reference model to IEEE specification. This part took an additional 3 engineer-months, including time to develop the necessary results about bit-vector arithmetic, but of course needs to be done only once. Having been 'certified' by formal proof as conforming to the IEEE standard, the FL reference model could be reused across many implementations. Moreover, the case-splitting proof strategy can also be reused. It therefore now takes a matter of days to formally verify the IEEE-compliance of a new adder implementation, with most of the effort confined to updating the circuit API.

Chen and Bryant [31] have applied a similar decomposition strategy to the verification of floating-point adders. Their case splits are also based on the difference between the operand exponents and they use a variation on the parametric representation when symbolically simulating their circuit. Their approach requires that they remove pipe-stage latches from the circuit before verification. In an era of high-speed circuits and complex clocking schemes, this 'unlatching' becomes increasingly difficult and risky. To mitigate the risk, Chen and Bryant rely on structural patterns in the circuit to preserve some aspects of the clocking behaviour in their unlatched circuit. With STE, unlatching is unnecessary. Also, in contrast to our work, Chen and Bryant verified only one data format (double precision), one rounding mode (round to nearest), and considered only normalized input data.

C. Instruction-length decoder

Our final example is an instruction-length decoder for the Intel IA-32 instruction set. This verification illustrates the use of induction to reason about infinite streams in ThmTac, the parametric representation for data-space decomposition, and using FL as a specification language.

Instructions in the IA-32 instruction set can vary in length from 1 to 11 bytes. Modern Intel processor implementations have divided the decoding process into three parts: a pre-decode pipeline that marks instruction boundaries, a

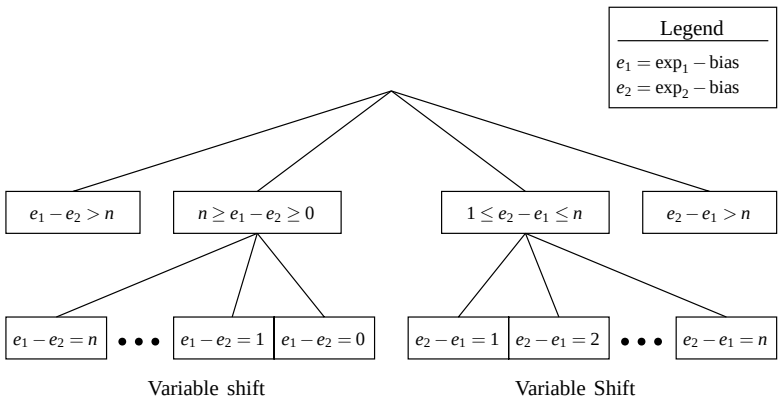


Fig. 8. Case splits for true addition

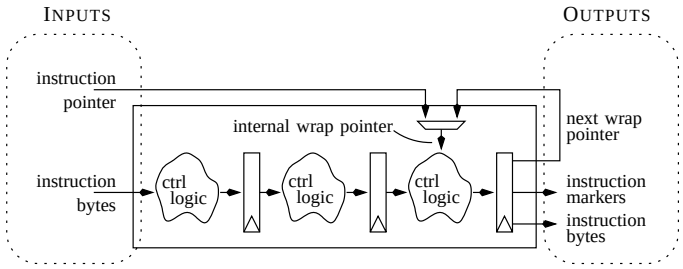


Fig. 9. IA-32 length-decoder high-level inputs and outputs

pipeline for aligning marked instructions, and a final pipeline that decodes aligned instructions using standard table-lookup techniques. This example concerns the pre-decode pipeline, called the ILD (instruction-length decoder). In this pipeline, instruction lengths are marked by annotating a stream of instruction bytes with markers that delineate the beginning and end of instructions

The instruction-length decoder has two primary inputs: a fixed-length *parcel* of instruction bytes, and a *wrap-around pointer* (wrap pointer for short) that indicates where to start decoding. The wrap pointer is necessary because variable-length IA-32 instructions are not word-aligned in memory—a preceeding instruction can end at any byte. Thus, some number of bytes in the current input parcel may be part of an instruction from a previous parcel. The two primary outputs to verify are the associated length marks for the inputs and the new value of the wrap pointer for the next parcel. The ILD is implemented as a pipelined datapath with internal state. For the last instruction in each parcel, the ILD computes the number of bytes that overflow into the next parcel. This number, along with some additional information, is stored as internal state. This is illustrated in Fig. 9.

Two attributes of the ILD make it difficult to verify. First, its functional specification reflects the significant complexity of the IA-32 instruction set. The instruction set has difference semantics depending on machine mode. Single ‘prefix’ bytes can change the semantics and even the length of the ensuing instruction. The combination of 2500 different opcodes, multiple addressing modes, and multiple machine modes conspire to create a very large space of possible behavior.

As with the FADD circuit, we address the specification difficulty by crafting the specification directly in FL. The functional specification includes textual tables similar in spirit to those found in the programmer-reference manual (PRM) [68]. The textual tables are translated into FL lists, which are in turn translated to Boolean relations represented as BDDs. The IA-32 relations are used to characterize legal instruction sequences and the corresponding marking information. The functionality of the specification is too complex to specify directly. The advantage of using FL to specify the functionality is that it can be organized using the same techniques as a large piece of software.

The second verification difficulty arises because the ILD must correctly mark instruction streams of arbitrary length. Obviously, STE cannot reason about arbitrary-length streams directly. Instead, we must frame the correctness statement in a richer logic (ThmTac, in this instance) and induct on the *length* of a given sequence. The induction step is accomplished with STE, and the STE inference rules in ThmTac are used to establish the overall correctness statement.

For this example, the base case consists of the beginning of instruction sequences. While intuition might suggest that this is a rare occurence, in fact every taken branch instruction marks the begining of an instruction sequence. The base case is decomposed into many cases, one for each possible alignment of branch targets within an instruction parcel.

The inductive step consists of the situation where the ILD continues to process instructions linearly. The inductive hypothesis is that the ILD has marked all instructions correctly up to the current cycle. The proof obligation establishes that the ILD will correctly mark the instructions in the current cycle. As with the base case, case-splitting is required to reason about every possible instruction alignment in the parcel.

A top-level case split is used to choose which architectural mode (16- or 32-bit) the ILD is operating in. In total, there were 56 cases. Both the induction argument and case-splitting were managed with ThmTac. We performed the induction reasoning directly in ThmTac’s logic; the case splitting was

managed with STE inference rules encoded as ThmTac tactics. The same BDD variable order was used for all cases. An initial order was created manually, after studying the specification, and then refined by automatic reordering.

Two aspects of this verification would make it very difficult in a platform without Forte's features:

- First, the specification is encoded by an FL function that generates arbitrary-length streams. By breaking these streams into fixed-length parcels that are the same length processed by the pipeline, we can compute outputs and next-state values without ever explicitly coding the next-state function. The only way to explicitly code the next-state function of the instruction pre-decoder would be to create a separate hardware implementation—a significantly more difficult process than writing it as a recursive functional program.
- Second, verifying such a specification in a traditional model checker would require creating the product machine between specification and implementation and verifying equivalence in the next cycle. This would require that the specification and implementation machines have equivalent don't-care sets—an infeasible requirement. Alternatively, a specification of the don't care sets would have to be created, also a difficult proposition.

In summary, verification of the instruction pre-decoder would have been difficult, if not impossible, without features that are unique to the Forte system: a general-purpose programming language for creating the specification and an interface in the same system to STE.

We estimate that the initial ThmTac proof took 2–3 weeks. Creating the initial specification of the IA-32 specification was a tedious task and required 5–6 weeks, including the initial debugging. Applying the proof to the ILD took another 3–4 weeks.

To preserve independence from internal design documents for the hardware implementation, the specification was written from a publicly-available architecture reference manual. Once created and debugged, the specification has been surprisingly robust. We have applied it, and found bugs, on several IA-32 processor designs.

As new features have been added to the IA-32 instruction set, it has been straightforward to add them to the formal specification. When the first additions occurred, they were added to the specification by the original author. Later additions and ports to new microprocessor designs have been performed by verification engineers that were not involved in the original proof. These efforts have required 2-3 months. Once the proof is ported to a new design, maintaining it is relatively straightforward because of the input/output nature of the specification, e.g. signal mappings are usually the only changes required when the underlying RTL changes.

The verification times of the proof have continually decreased. When first performed about six years ago, the model-checking part of the verification used 347 MB of RAM and 273 minutes of CPU time on a proprietary workstation. On a modern microprocessor running Linux, the complete verification uses 198 MB of RAM and only 9 minutes of CPU time—a 30x improvement. The improvement in memory utilization and about 3x of the CPU time improvement are due to ongoing optimizations in Forte.

X. Conclusions

We have described the Forte formal verification environment, which combines symbolic trajectory evaluation and lightweight theorem proving in higher-order logic. A guiding principle in the design of Forte is our view of formal verification as an interactive activity, with the major result being a set of proof scripts that can be used for debugging and regression, and re-used in future verification efforts targeting similar functionality. We see proof script development as program development, and therefore have interfaced and tightly integrated symbolic trajectory evaluation and lightweight theorem proving with FL, a general-purpose functional programming language. FL allows the environment to be customized and large proof efforts organized and scripted effectively, and also serves as an expressive specification language at a level much above the temporal logic primitives.

While developing Forte, we have been conscious of the competing goals of capability and usability for the tools. We are also keenly aware that a routine verification for the technology developer may be virtually impossible for others to duplicate. We have developed a methodology for using Forte that aims to address these issues by making the Forte environment usable in practice on industrial-scale problems by verification engineers. The Forte environment coupled with our methodology has proved highly effective in large-scale industrial trials on datapath-dominated hardware [3], [22], [23].

Acknowledgments

Feedback from exercising our methodology and tools on realistic designs was essential to making Forte effective. We are particularly grateful to the users of Forte at Intel and to the Intel design teams who supplied case studies for our own example verifications. We also thank the anonymous referees of this article, whose comments prompted many improvements to our presentation.

References

[1] T. Kropf, *Introduction to Formal Hardware Verification*. Springer-Verlag, 1999.

[2] Á. T. Eirksson, "The formal design of 1M-gate ASICs," in *Formal Methods in Computer-Aided Design*, ser. Lecture Notes in Computer Science, G. Gopalakrishnan and P. Windley, Eds., vol. 1522. Springer-Verlag, 1998, pp. 49–63.

[3] 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, available at developer.intel.com/technology/itj/.

[4] T. Schubert, "High level formal verification of next-generation microprocessors," in *ACM/IEEE Design Automation Conference*. ACM Press, June 2003, pp. 1–6.

[5] Y. Xu, E. Cerny, A. Silburt, A. Coady, Y. Liu, and P. Pownall, "Practical application of formal verification techniques on a frame mux/demux chip from Nortel Semiconductors," in *Correct Hardware Design and Verification Methods*, ser. Lecture Notes in Computer Science, L. Pierre and T. Kropf, Eds., vol. 1703. Springer-Verlag, 1999, pp. 110–124.

[6] Y. Lu and M. Jorda, "Verifying a gigabit ethernet switch using SMV," in *ACM/IEEE Design Automation Conference*. ACM Press, June 2004, pp. 230–233.

[7] R. E. Bryant, "Graph-based algorithms for boolean function manipulation," *IEEE Transactions on Computers*, vol. C-35, no. 8, pp. 677–691, 1986.

[8] C. Berthet, O. Coudert, and J. C. Madre, "New ideas on symbolic manipulations of finite state machines," in *IEEE/ACM International Conference on Computer Aided Design (ICCAD)*, 1990, pp. 224–227.

[9] O. Coudert, J. C. Madre, and C. Berthet, "Verifying temporal properties of sequential machines without building their state diagrams," in *Computer Aided Verification: 2nd International Workshop, June 18–21, 1990: Proceedings*, ser. Lecture Notes in Computer Science, E. M. Clarke and R. P. Kurshan, Eds., vol. 531. Springer-Verlag, 1991, pp. 23–32.

[10] K. L. McMillan, *Symbolic Model Checking*. Kluwer Academic Publishers, 1993.

[11] E. M. Clarke, O. Grumberg, and D. Peled, *Model Checking*. MIT Press, 1999.

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

[13] S. Owre, J. M. Rushby, and N. Shankar, "PVS: A prototype verification system," in *Proceedings of CADE-11*, ser. Lecture Notes in Artificial Intelligence, D. Kapur, Ed., vol. 607. Springer-Verlag, 1992, pp. 748–752.

[14] T. Melham, *Higher Order Logic and Hardware Verification*. Cambridge University Press, 1993.

[15] A. Ferrari and A. Sangiovanni-Vincentelli, "System design: traditional concepts and new paradigms," in *International Conference on Computer Design (ICCD)*, 1999, pp. 2–12.

[16] J. Joyce and C.-J. Seger, "Linking BDD based symbolic evaluation to interactive theorem proving," in *ACM/IEEE Design Automation Conference*, 1993.

[17] S. Rajan, N. Shankar, and M. Srivas, "An integration of model checking automated proof checking," in *Computer Aided Verification: 8th International Conference, New Brunswick, USA, July 31 - August 3, 1996: Proceedings*, ser. Lecture Notes in Computer Science, R. Alur and T. Henzinger, Eds., vol. 1102. Springer-Verlag, 1996, pp. 411–414.

[18] L. A. Dennis, G. Collins, M. Norrish, R. Boulton, K. Slind, G. Robinson, M. Gordon, and T. Melham, "The PROSPER Toolkit," in *Tools and Algorithms for the Construction and Analysis of Systems*, ser. Lecture Notes in Computer Science, S. Graf and M. Schwartzbach, Eds., vol. 1785. Springer-Verlag, 2000, pp. 78–92.

[19] K. L. McMillan, "Verification of infinite state systems by compositional model checking," in *Correct Hardware Design and Verification Methods*, ser. Lecture Notes in Computer Science, L. Pierre and T. Kropf, Eds., vol. 1703. Springer-Verlag, 1999, pp. 219–233.

[20] 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, March 1995.

[21] 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. Lecture Notes in Computer Science, Y. Bertot, G. Dowek, A. Hirschowitz, C. Paulin, and L. Théry, Eds., vol. 1690. Springer-Verlag, 1999, pp. 323–340.

[22] R. Kaivola and M. D. Aagaard, "Divider circuit verification with model checking and theorem proving," in *Theorem Proving in Higher Order Logics: 13th International Conference, TPHOLs 2000, Portland, August 2000: Proceedings*, ser. Lecture Notes in Computer Science, M. Aagaard and J. Harrison, Eds., vol. 1869. Springer-Verlag, 2000, pp. 338–355.

[23] 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.

[24] Technical Publications and Training, Intel Corporation, *Forte/FL User Guide: Version 1.0*, Design Technology, Intel Corporation, 2003, release for academic, non-commercial use.

[25] M. D. Aagaard, R. B. Jones, T. F. Melham, J. W. O'Leary, and C.-J. H. Seger, "A methodology for large-scale hardware verification," in *Formal Methods in Computer-Aided Design: Third International Conference, FMCAD 2000: Austin, November 2000: Proceedings*, ser. Lecture Notes in Computer Science, J. W. A. Hunt and S. D. Johnson, Eds., vol. 1954. Springer-Verlag, 2000, pp. 263–282.

[26] 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, July/August 2001.

[27] N. Krishnamurthy, M. S. Abadir, A. K. Martin, and A. J. A., "Design and development paradigm for industrial formal verification CAD tools," *IEEE Design and Test of Computers*, vol. 18, no. 4, pp. 26–35, july–august 2001.

[28] J. Bhadra, A. K. Martin, M. S. Abadir, and A. J. A., "Using abstract specifications to verify powerpc custom memories by symbolic trajectory evaluation," in *Correct Hardware Design and Verification Methods*, ser. LNCS, T. Margaria and T. F. Melham, Eds., no. 2144. Springer-Verlag, Sept. 2001, pp. 386–402.

[29] M. S. Abadir, K. L. Albin, J. Havlicek, N. Krishnamurthy, and A. K. Martin, "Formal verification successes and motorola," *Formal Methods in System Design*, vol. 22, no. 2, pp. 117–123, Mar. 2003.

[30] D. Peled, "Partial order reduction: Linear and branching temporal logics and process algebras," in *POMIV'96, Partial Orders Methods in Verification*. American Mathematical Society, 1996.

[31] Y.-A. Chen and R. Bryant, "Verification of floating-point adders," in *Workshop on Computer-Aided Verification*, A. J. Hu and M. Y. Vardi, Eds., 1998, pp. 179–196.

[32] P. R. Halmos, *Naive Set Theory*. Springer-Verlag, 1987.

[33] B. A. Davey and H. A. Priestley, *Introduction to Lattices and Order*. Cambridge University Press, 1990.

[34] A. Jain, "Formal hardware verification by symbolic trajectory evaluation," Ph.D. dissertation, Carnegie Mellon University, August 1997.

[35] R. E. Bryant, "A methodology for hardware verification based on logic simulation," *Journal of the ACM*, vol. 38, no. 2, pp. 299–328, April 1991.

[36] R. E. Bryant, D. E. Beatty, and C.-J. H. Seger, "Formal hardware verification by symbolic ternary trajectory evaluation," in *ACM/IEEE Design Automation Conference*, 1991, pp. 297–402.

[37] C.-T. Chou, "The mathematical foundation of symbolic trajectory evaluation," in *Computer Aided Verification: 11th International Conference, Trento, Italy, July 6-10 1999: Proceedings*, ser. Lecture Notes in Computer Science, N. Halbwachs and D. Peled, Eds., vol. 1633. Springer-Verlag, 1999.

[38] 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, December 1993.

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

[40] Z. Zhu and C.-J. Seger, "The completeness of a hardware inference system," in *Computer Aided Verification*, 1994, pp. 286–298.

[41] S. Hazelhurst and C.-J. H. Seger, "Symbolic trajectory evaluation," in *Formal Hardware Verification*, T. Kropf, Ed. Springer-Verlag, 1997, ch. 1, pp. 3–78.

[42] M. D. Aagaard, R. B. Jones, and C.-J. H. Seger, "Formal verification using parametric representations of Boolean constraints," in *ACM/IEEE Design Automation Conference*, July 1998.

[43] D. M. Russinoff, "A case study in formal verification of register-transfer logic with ACL2: The foating point adder of the AMD Athlon processor," in *Formal Methods in Computer-Aided Design: Third International Conference, FMCAD 2000: Austin, November 2000: Proceedings*, ser. Lecture Notes in Computer Science, J. W. A. Hunt and S. D. Johnson, Eds., vol. 1954. Springer-Verlag, 2000, pp. 3–36.

[44] M. N. Velev and R. E. Bryant, "Efficient modeling of memory arrays in symbolic ternary simulation," in *Tools and Algorithms for the Construction and Analysis of Systems (TACAS '98)*, ser. Lecture Notes in Computer Science, B. Steffen, Ed., vol. 1384. Springer-Verlag, 1998, pp. 136–150.

[45] M. Pandey, R. Raimi, R. E. Bryant, and M. S. Abadir, "Formal verification of content addressable memories using symbolic trajectory evaluation," in *ACM/IEEE Design Automation Conference (DAC)*. ACM Press, June 1997, pp. 167–172.

[46] M. Pandey and R. E. Bryant, "Exploiting symmetry when verifying transistor-level circuits by symbolic trajectory evaluation," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 18, no. 7, pp. 918–935, July 1999.

[47] T. F. Melham and R. B. Jones, "Abstraction by symbolic indexing transformations," in *Formal Methods in Computer-Aided Design: 4th International Conference, FMCAD 2002: Portland, November 2002: Proceedings*, ser. Lecture Notes in Computer Science, M. D. Aagaard and J. W. O'Leary, Eds., vol. 2517. Springer-Verlag, 2002, pp. 1–18.

[48] R. B. Jones, *Symbolic Simulation Methods for Industrial Formal Verification*. Kluwer Academic Publishers, 2002.

[49] V. Bertacco, M. Damiani, and S. Quer, "Cycle-based symbolic simulation of gate-level synchronous circuits," in *ACM/IEEE Design Automation Conference*. ACM Press, June 1999, pp. 391–396.

[50] P. P. Chauhan, E. M. Clarke, and D. Kroening, "A SAT-based algorithm for reparameterization in symbolic simulation," in *ACM/IEEE Design Automation Conference*. ACM Press, June 2004, pp. 524–529.
[51] C. Wilson and D. L. Dill, "Reliable verification using symbolic simulation with scalar values," in *ACM/IEEE Design Automation Conference*. ACM Press, June 2000, pp. 124–129.
[52] L. Augustson, "A compiler for Lazy-ML," in *ACM Symposium on Lisp and Functional Programming*, 1984, pp. 218–227.
[53] M. J. Gordon, R. Milner, and C. P. Wadsworth, *Edinburgh LCF: A Mechanised Logic of Computation*, ser. Lecture Notes in Computer Science. Springer-Verlag, 1979, vol. 78.
[54] S. Owre, J. M. Rushby, and N. Shankar, "PVS: A prototype verification system," in *Automated Deduction - CADE-11, 11th International Conference on Automated Deduction*, ser. Lecture Notes in Artificial Intelligence, D. Kapur, Ed., vol. 607, 1992, pp. 748–752.
[55] R. L. Constable, S. F. Allen, H. M. Bromley, W. R. Cleaveland, J. F. Cremer, R. W. Harper, D. J. Howe, T. B. Knoblock, N. P. Mendler, P. Panangaden, J. T. Sasaki, and S. F. Smith, *Implementing Mathematics with the Nuprl Development System*. Prentice-Hall, 1986.
[56] S. L. P. Jones, *The Implementation of Functional Programming Languages*. Prentice-Hall, 1987.
[57] A. Church, "A formulation of the simple theory of types," *Journal of Symbolic Logic*, vol. 5, pp. 56–68, 1940.
[58] A. Trybulec and H. Blair, "Computer assisted reasoning with Mizar," in *International Joint Conferences on Artificial Intelligence*, A. K. Joshi, Ed., vol. 1. Morgan Kaufmann, Aug. 1985, pp. 26–28.
[59] J. Harrison, "A Mizar mode for HOL," in *Theorem Proving in Higher Order Logics*, J. von Wright, J. Grundy, and J. Harrison, Eds. Springer-Verlag, Aug. 1996, pp. 203–220.
[60] D. Syme, "Three tactic theorem proving," in *Theorem Proving in Higher Order Logics: 12th International Conference: Proceedings*, ser. Lecture Notes in Computer Science, Y. Bertot, G. Dowek, A. Hirschowitz, C. Paulin, and L. Théry, Eds., vol. 2144. Springer-Verlag, 1999, pp. 203–220.
[61] K. Slind, "Derivation and use of induction schemes in higher-order logic," in *Theorem Proving in Higher Order Logics: 10th International Conference, Murray Hill, USA, August 19–22, 1997: Proceedings*, ser. Lecture Notes in Computer Science, E. L. Gunter and A. Felty, Eds., vol. 1275. Springer-Verlag, 1997, pp. 275–291.
[62] T.-Y. Yeh and Y. N. Patt, "Two-level adaptive branch prediction," in *Proceedings of the 24th Annual ACM/IEEE Intl. Symposium on Microarchitecture*, 1991, pp. 51–61.
[63] N. C. Ip and D. L. Dill, "Better verification through symmetry," *Formal Methods in System Design*, vol. 9, no. 1/2, pp. 41–75, Aug. 1996.
[64] K. McMillan, "Verification of an implementation of Tomasulo's algorithm by compositional model checking," in *Computer Aided Verification (CAV)*, ser. LNCS, A. J. Hu and M. Y. Vardi, Eds., vol. 1427. Springer-Verlag, 1998, pp. 110–121.
[65] "IEEE standard for binary floating-point arithmetic," ANSI/IEEE Std 754-1985.
[66] J. L. Hennessy and D. A. Patterson, *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann, 1990, third edition, with D. Goldberg, 2002.
[67] J. Feldman and C. Retter, *Computer Architecture: A Designer's Text Based on a Generic RISC*. McGraw-Hill, 1994.
[68] *Intel Architecture Software Developer's Manual, Volume 2: Instruction Set Reference*, Intel Corporation, 1997, order Number 243191.

Carl-Johan H. Seger is a Principal Engineer at Intel's Strategic CAD Labs in Hillsboro, Oregon. He received the Ph.D. in Computer Science from the University of Waterloo. His research interests include formal hardware verification and asynchronous circuits.

Robert B. Jones is a Principal Engineer at Intel's Strategic CAD Labs in Hillsboro, Oregon. He received the B.Sc. from Brigham Young University and the M.Sc. and Ph.D. from Stanford University, all in Electrical Engineering. His dissertation on microprocessor verification was chosen as the 2000 ACM Outstanding Ph.D. Dissertation in Electronic Design Automation. His research interests include practical application of formal methods to hardware systems specification, architecture, and verification. He is a member of the IEEE and ACM.

John W. O'Leary leads the formal verification research group at Intel's Strategic CAD Labs. He received the B.Sc. from the University of Calgary, the M.Sc. from Carleton University and the Ph.D. from Cornell University, all in Electrical Engineering. From 1987 to 1990 he worked in the CAD group at Bell-Northern Research, Ottawa. His research interests are in formal specification and verification of hardware, particularly in verification approaches that combine theorem proving and model checking. He is a member of the IEEE.

Tom Melham is a Professor of Computer Science at the University of Oxford and a Fellow of Balliol College. He received his Ph.D. from the University of Cambridge for his early research in formal hardware verification and was a co-developer of the HOL theorem prover. His research interests include industrial-scale formal verification, abstraction, combined model checking and theorem proving, and integrating verification into design methodologies. He was elected a Fellow of the Royal Society of Edinburgh in 2002.

Mark D. Aagaard is an Associate Professor in the Department of Electrical and Computer Engineering at the University of Waterloo, in Waterloo, Canada. He received his M.Sc. and Ph.D. degrees from Cornell University. His research area is the design and verification of digital systems and circuits at the register-transfer-level and above. Aagaard was a visiting scholar at the University of British Columbia from 1994 to 1996. He was with Intel Corporation from 1996 to 2000, where he was a member of the Forte formal verification group and a member of formal verification team for the Pentium 4 processor. In 2003, Mark received an Ontario Premier's Research Excellence award. He is a member of the IEEE.

Clark Barrett is an Assistant Professor of Computer Science at the Courant Institute of New York University. He received his bachelor's degree in Mathematics, Computer Science, and Electrical Engineering from Brigham Young University and his Ph.D. in Computer Science from Stanford University. He is a co-author of the Stanford Validity Checker (SVC), and its successor, the Cooperating Validity Checker (CVC). His research interests include automated theorem proving and applications of theorem proving to hardware and software verification.

Don Syme is a researcher at Microsoft Research, Cambridge, UK. His Ph.D. from the University of Cambridge was on novel techniques for interactive theorem proving and their application to large semi-automated proofs of the type soundness of subsets of the Java language. In 1998, he worked for six months at Intel's Strategic CAD Labs on some of the theorem proving aspects of the work described in this paper. More recently he has been the co-architect and a lead implementer of the support for 'generics' in C# and the .NET Common Language Runtime, and also the designer of the language F#.