Enhancing ABC for LTL Stabilization Verification of SystemVerilog/VHDL Models

Jiang Long, Sayak Ray, Baruch Sterin, Alan Mishchenko, Robert Brayton
Berkeley Verification and Synthesis Research Center (BVSRC)
Department of EECS, University of California, Berkeley
{jlong, sayak, sterin, alanmi, brayton}@eecs.berkeley.edu

***Abstract*—We describe a tool which combines a commercial front-end with a version of the model checker, ABC, enhanced to handle a subset of LTL properties. Our tool, VeriABC, provides a solution at the RTL level and produces models for synthesis and formal verification purposes. We use Verific (a commercial software) as the generic parser platform for SystemVerilog and VHDL designs. VeriABC traverses the Verific netlist database structure and produces a formal model in the AIGER format. LTL can be specified using SVA 2009 constructs that are processed by Verific. VeriABC traverses the resulting SVA parse trees and produces equivalent LTL formulae using the F,G, Until and X operators. The model checker in ABC has been enhanced to handles LTL stabilization properties, an important subset of LTL. The paper presents VeriABC's implementation strategy, software architecture, tool flow, environment setup for formal verification, use model, the specification of properties in SVA and translation into LTL. Finally the properties are translated into safety properties that can be verified by the ABC model checker.**

I. Introduction

We present an integrated tool flow for liveness model checking using industry hardware description languages (HDLs) and SystemVerilog Assertions: (i) VeriABC: a front-end to read in hardware models expressed in HDLs, and (ii) capability of model checking a subset of liveness properties. VeriABC is able to read in hardware models expressed in SystemVerilog or VHDL. SystemVerilog and VHDL languages are the most popular HDLs being used in industry today for digital designs. VeriABC generates a formal model in the AIGER[2] format and relies on a commerical front-end, Verific, to build a generic parser platform for HDLs. This allows down-stream tool flows in synthesis and verification. A version of ABC was enhanced from a safety-only verification engine to allow both safety and liveness verification. Our current version supports a particular subset of liveness properties called *stabilization properties* or *generalized fairness constraints* (defined in Section IV).

In a typical use model, a user will develop a hardware design in SystemVerilog or VHDL, and specify its correctness requirements in the property specification language *SystemVerilog Assertion* (SVA). SVA has been adopted into IEEE SystemVerilog standard and is supported by major commercial tools in simulation, synthesis and verification. The SVA language in SystemVerilog 2009 standard contains liveness constructs that allow full specification of liveness properties as those defined in LTL formulas. In our framework, a user can specify both safety properties and liveness properties (stabilization properties, to be precise). In this paper, we detail its liveness capabilities.

After reading in a design, VeriABC bit-blasts it into a bit-level netlist and converts the SVA stabilization properties into an intermediate LTL representation. Then the LTL properties are folded into the bit-level netlist in an appropriate way (using an extended *liveness-to-safety* conversion, explained later in Section IV). The resulting bit-level netlist represents a formal model of the design, represented as an and-inverter graph (AIG). And-inverter graphs are concise representations of finite state machines. The AIGER[2] format is a prevalent format for AIG representation. supported by various academic tools and used in annual hardware model checking competitions. Since our liveness verification is based on liveness-to-safety conversion, eventually the safety verification backend in ABC[1] is called which works on the bit-level netlist produced by the VeriABC front-end. We have used this methodology to verify liveness in the context of compositional verification of deadlock freedom of micro-architectural communication fabrics. Preliminary experimental results are encouraging.

A. Related work

Although parsing and elaborating RTL languages are a standard practice for commercial EDA products, it is a daunting task for academics due to language complexity and continuous language evolution over the years. Although, *vl2mv*[12] was an academic tool that attempted to treat a significant subset of the Verilog language for synthesis and verification purposes, it was not maintained and language support was not complete. Tools like ABC[1], VIS[9] parse subsets of Verilog language too strict and not applicable in a broad setting. Freely accessible tools like icarus[3] contain Verilog languages front-end but are not up-to-date with newer SystemVerilog features.

Our choice of a commercial and stable front-end Verific, allows academics to get around the language barrier to access real-world designs.

Liveness-to-safety conversion was first proposed in [7], [24]. They demonstrated that verification problems for any ω-regular property can be converted into a verification problem of an equisatisfiable safety problem. Their algorithm has been deployed successfully in industrial setups and used to verify liveness properties of microprocessor designs [6]. Our liveness-to-safety conversion algorithm for stabilization is essentially an extension of the algorithm proposed in [24] and is broader than discussed in [6].

B. Contribution

As illustrated in Figure 1, we combine a commercial front-end, Verific, with a version of our model checker, ABC, enhanced to handle a subset of LTL properties. Our tool processes the Verific output, conducts various modeling procedures on the design, compiles SVA into LTL formulas, then the enhanced ABC processes the LTL formulas for liveness model checking. We detail the software architecture, tool flow, formal model construction, SVA compilation and downstream LTL modeling checking.

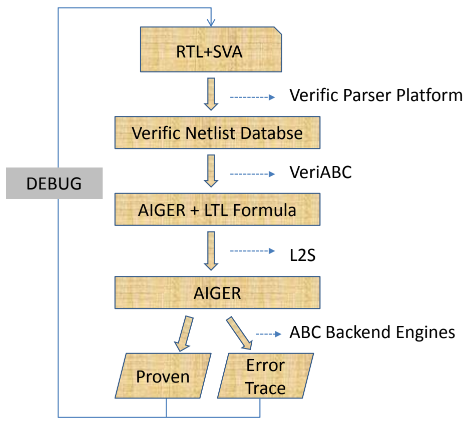

Fig. 1. Complete Tool Flow

C. Organization of the Paper

We first discuss the capabilities of the Verific parser platform. In Section III we describe the architecture, formal modeling of VeriABC and translation of SVA into LTL. In Section IV the stabilization properties are described in further detail. Section V describes the liveness-to-safety conversion for stabilization properties. Experimental results are presented in Section VI.

II. Background: Verific Parser Platform

Verific Design Automation[4] builds SystemVerilog and VHDL Parser Platforms which enable its customers to develop advanced EDA products quickly and at low cost. Verific's Parser Platforms are distributed as C++ source code or library and build on all 32 and 64 bit Unix, Linux, and Windows operating systems. Applications vary from formal verification to synthesis, simulation, emulation, virtual prototyping, in circuit debug, and design-for-test. We chose Verific as our front-end for its commercial success and stability in supporting the latest language constructs in SystemVerilog.

Figure 2 shows the architecture of the Verific parser front-end. The main commands we use in Verific library are $analyze$ and $elaborate$. $Analyze$ creates parse-trees and performs type-inferencing to resolve the meaning of identifiers. The Parser/Analyzer supports the entire SystemVerilog IEEE 1800, VHDL IEEE 1076-1993, and Verilog IEEE 1364-1995/2001 languages, without any restrictions. The resulting parse tree comes with an extensive API.

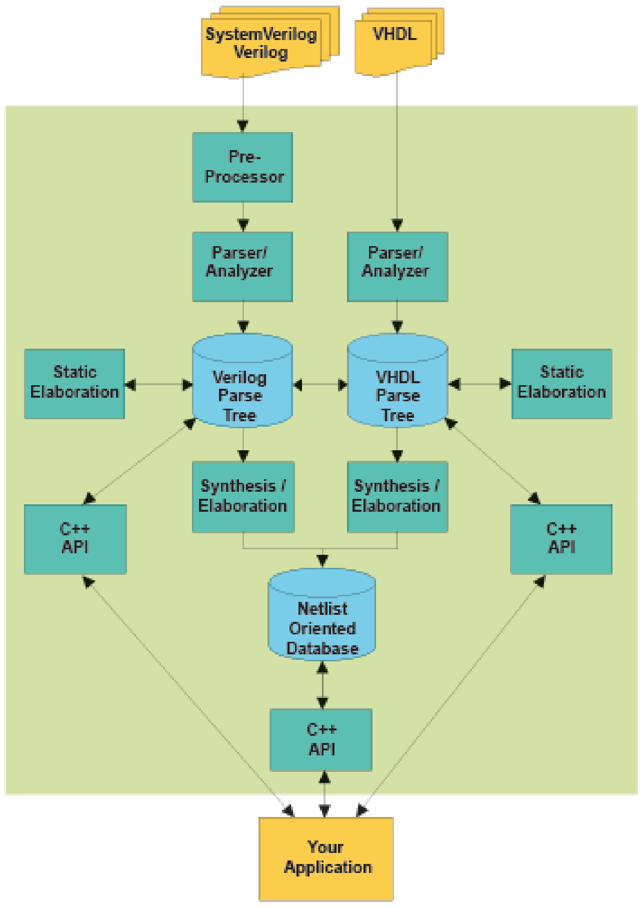

Fig. 2. Verific Parser Flow

$Elaborate$ supports both static elaboration and RTL elaboration. Static elaboration elaborates the entire language, and specifically binds instances to modules, resolves library references, propagates defparams, unrolls generate statements, and checks all hierarchical names and function/task calls. The result after static elaboration is an elaborated parse tree, appropriate for simulation like applications. RTL elaboration is limited to the synthesizable subset of the language. In addition to the static elaboration tasks for this subset, it generates sequential networks through flipflop and latch detection, and Boolean extraction. The result after RTL elaboration is a netlist database, appropriate to applications such as logic synthesis and formal verification. This netlist database is the starting point of VeriABC and we utilize Verific provided C++ APIs to access the database.

A. Verific Netlist Database Structure

In this Section, we use Verilog terminology to present Verific's netlist database structures. The netlist database is rather intuitive and adheres to the module definitions. Shown in Table I, there is a one-to-one correspondence between the C++ API class definitions and Verilog constructs.

A $Netlist$ corresponds to $module$ definitions in Verilog while an $Instance$ object corresponds to module instantiation,

Verific Database C++ API Class	Verilog RTL Objects
Netlist	*Module definition*
Instance	*Module instantiation*
Port	*Module port declarations*
Net	*wire/reg/assign*
PortRef	*Port to Net connectivity*

TABLE I
VERIFIC NETLIST OBJECTS

after the module's parameters have been characterized. An $Instance$ is a thin copy of the $Netlist$ plus a pointer to its parent netlist. A $Netlist$ contains a set of $Ports$, $Nets$ and $Instances$ for its internal logic structure. A $Port$ corresponds to the Verilog port definitions which can be *input*, *output* or *inout*. A Net is a named component, intuitively a *wire*. $PortRef$ contains the connectivity between a $Port$ and a Net. The direction of the $PortRef$ can be *in*, *out*, or *inout* depending on the type of $Port$ it contains. Using these C++ objects, the Verific netlist database defines a directed hyper-graph and encapsulates the following types of information:

Design Hierarchy
: Design hierarchy is captured as an instance tree by the parent pointers in the $Instance$ with a top-level netlist as the root.

Unique Hierarchical Name
: Following the design hierarchy through the instance tree, each internal object is assigned a unique hierarchical name.

Connectivity
: A directed hyper-graph is defined through $Port$, Net and $PortRef$: $Port$ being the node, Net being the edge, and $PortRef$ containing the connectivity and direction information between pairs of a $Port$ and a Net. As an edge in the hyper-graph, a Net can be referenced in multiple $PortRef$ objects.

Logic Definition
: At the leaf of the design hierarchy, a $Netlist$ of primitive operator types such as *and*, *or*, *adder*, flipflop, latches etc defines the basic logic operators.

Recursively, the functional behavior of the design is captured through the directed hierarchical hyper-graph with basic logic operators at its leaf level.

III. VERIABC

VeriABC traverses the above netlist database and transforms it into a monolithic AIG which can be treated as a directed acyclic graph (DAG). The AIG contains primary input, primary output, register nodes and and-inverter nodes. Each named $Port$ and Net in the Verific netlist has a mapped node in the AIG graph. Additional book-keeping information such as hash tables are created that map the hierarchical name to the corresponding AIG node. The down-stream model checker ABC then reads in this AIG to conduct formal analysis.

A. Architecture

A hyper-graph is rather hard to traverse and conduct analysis/transformation at the same time.

As shown in Figure 3, we employ a two phase approach. First we construct an intermediate netlist graph, a DAG with extra annotated node types representing logic structure and connectivity of the flattened design. In addition to the simple node types in and-inverter graphs, extra node types contain annotations for language constructs such as tri states, flipflops and latches etc. For example, flipflops contain set/reset pins and driving d-pins; latches contain additional gated-clock definitions. By language definition, a design can specify any signal for the clock and reset signals. Design behavior is defined by event-driven semantics. Further analysis needs be done to determine if there is an AIG representation that can capture the original design semantics. The intermediate netlist is a DAG for which various traversal algorithms can be conducted for the later analysis. For this step, VeriABC only traverses the design hierarchy and hyper-graph in the Verific netlist database to gather information and construct the intermediate DAG representation without conducting any design style checking or transformation.

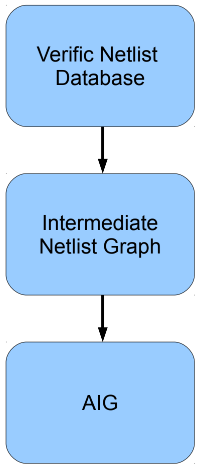

Fig. 3. VeriABC Architecture

B. Formal Modeling

The end result of VeriABC is an AIG model that is consistent with the original RTL design semantics. AIG is recognized as a finite state machine model. Compared to the event-driven semantics in HDL language, the execution semantics of an AIG is synchronous with an implicit universal clock that ticks at every step of the execution. The register loads in its driver value at the beginning of each step. The semantics inferred from the Verific netlist structure is more complex, such as its flip-flop can have arbitrary reset logic and clock network. The task in formal modeling is to transform the above design components into AIG registers with additional glue logic so that it maintains the consistency. In its simplest form, the following check determines if the design can be readily represented by an AIG.

- All registers are clocked by the same primary input signal which does not fanout to other nodes.
- Reset/Set signals are primary inputs

- No combinational loops
- No multiple drivers per node

More complex design modeling and transformations can be achieved by identifying certain patterns by traversing the intermediate netlist graph. Our current implementation supports the above form and produces a design style summary for debugging purposes. Though capacity and performance depends on the type of individual transformation and analysis, for the above ones, the transformation and design style checking is very fast, as it conducts only a few traversals of the intermediate netlist graph. After design style checking, a final traversal of the intermediate netlist graph generates an AIGER file representing the formal model.

C. Commands Implemented

VeriABC implementation utilizes the Tcl command interface shipped with the Verific library distribution. The following is the list of commands implemented to manage the environment setup for formal verification.

veriabc_analyze
: This command constructs the intermediate netlist graph in Figure 3 and conducts design style checking.

veriabc_set_reset
: Although a *reset* signal can be automatically detected in certain situations, this command provides the user with the option to specify the length of the reset sequence and phase of the *reset* condition. A user can also specify the initial value of the registers through a VCD waveform or textual file. In the generated formal model, the initial value of the registers will be valued at the end of the reset sequence and the reset condition will be disabled after reset.

veriabc_set_clock
: A user can specify the clock periods and relationships in the situation when multi-clocks are in the design. A phase-counter network will be created in the formal model to generate the corresponding clock signals.

veriabc_set_cutpoint
: This command will prune the cone-of-influence at cutpoint signal and treat it as free input.

veriabc_set_constant
: This command sets either an input or a cut-point signal to a constant value.

veriabc_set_assume
: This constrains the design signal to be a constant.

veriabc_write
: write out the final formal model in AIGER format.

The above commands give the user flexibility to model the environment with constraints and conduct design abstractions during verification.

D. SVA to LTL Compilation

In SystemVerilg 2009 standard, a rich set of LTL operators are added into SVA language. The SVA properties shown in Figure 4 are the templates for matching the basic LTL operators used in the set of stabilization properties. For stabilization properties, in our current implementation, we restrict the p and q to be Boolean expressions which seems to be sufficient in practice. The SVA verification directive *assume* is used to specify a fairness constraint, while *assert* is used to specify the target LTL properties.

```
property Until(p,q);
  p s_until q;
endproperty

property GF(p);
  always (s_eventually p);
endproperty

property FG(p);
  s_eventually (always p);
endproperty

property X(p);
  s_nexttime (p);
endproperty
```

Fig. 4. SVA Liveness Template

Verific also processes SVA constructs into the netlist database structure, essentially a corresponding parse tree is integrated into the netlist. We conduct traversals of the parse tree, identify specific liveness constructs and map them into the corresponding LTL formula. At the end of the procedure, along with the AIGER file generated, a separate file containing the LTL formulas are generated indicating target liveness assertions and fairness constraints. The support signals referred to in the LTL formulas are named output signals in the AIGER file. Although we currently only support stabilization properties in LTL, the full LTL language using X, F, G, U operators can be specified fully and translated through SVA constructs. In doing so, this completes the formal model generation and SVA compilation at the RTL level.

IV. Liveness Model Checking in VeriABC

In the FSM modeling formalism, the most intuitive notion of stabilization states that the system will always reach a particular state and stay there forever, no matter which state the system started from or which path it took. Relaxing this notion a bit, stabilization means that the system will eventually reach and stay within a given subset of states. Also, stabilization may denote conditions on the input and output signals of a system when it attains a stable state. Applications of stabilization properties have been demonstrated in [14] and [18], to name a few. We review some basic temporal logic terminology and formally define stabilization properties using LTL below.

A. LTL, model checking and stabilization property

Familiarity is assumed with LTL, basic model checking algorithms, and related terminology like Kripke structures and

Büchi automata. For further details, see [13]. In our current context, we use LTL properties $\mathbf{GF}p$ and $\mathbf{FG}p$, and thus overview their semantics here: let π be a path in some Kripke structure K; $\pi \models_K \mathbf{G}p$ means property p will hold on every state along π; $\pi \models_K \mathbf{F}p$ means the property p will hold eventually on some state along π; $\pi \models_K \mathbf{GF}p$ means p will hold along π infinitely often, and $\pi \models_K \mathbf{FG}p$ means p will hold eventually on π forever. Since temporal operators $\mathbf{F}$ and $\mathbf{G}$ are dual (i.e. $\mathbf{F}p \equiv \neg\mathbf{G}\neg p$), operators $\mathbf{FG}$ and $\mathbf{GF}$ are also dual (i.e. $\mathbf{FG}p \equiv \neg\mathbf{GF}\neg p$).

Definition 1 (GF-atom): Any LTL formula of the form $\mathbf{GF}p$ or $\mathbf{FG}p$, where p is some atomic proposition or some Boolean formula involving atomic propositions only, will be called a 'GF-atom'.

Stabilization properties are defined as the family of LTL formulas that are Boolean combinations of GF-atoms. Formally:

Definition 2 (Stabilization Property): The set of stabilization properties is the syntactic subset of LTL defined as follows:

- any GF-atom is a stabilization property
- if ϕ is a stabilization property, then so is $\neg\phi$
- if ϕ and ψ are stabilization properties, then so are $\phi \wedge \psi$ and $\phi \vee \psi$

Example 1: $\mathbf{FG}p$, $\mathbf{GF}p \Rightarrow \mathbf{GF}q$, $\mathbf{FG}p \wedge \mathbf{FG}q \Rightarrow \mathbf{FG}r$, and $\mathbf{FG}p \Rightarrow \mathbf{FG}q \vee (\mathbf{FG}r \wedge \mathbf{GF}s)$ are stabilization properties where p, q, r, and s are atomic propositions or Boolean formulas involving atomic propositions only and $a \Rightarrow b$ is the usual shorthand for $\neg a \vee b$. However, $\mathbf{G}(r \Rightarrow \mathbf{F}g)$ is an LTL liveness property but not a stabilization property.

Needless to say, these are all liveness properties. But not all of them specify so-called system stabilization directly. Properties like $\mathbf{FG}p$ and $\mathbf{FG}p \wedge \mathbf{FG}q \Rightarrow \mathbf{FG}r$ (or its generalization $\wedge_{i=1}^{k}\mathbf{FG}p_i \Rightarrow \mathbf{FG}q$) are perhaps the most elementary stabilization properties. $\mathbf{FG}p$ means that the system eventually will reach a state from where p will always hold, i.e. the system will eventually 'stabilize' at p. $\mathbf{FG}p \wedge \mathbf{FG}q \Rightarrow \mathbf{FG}r$ means that if the system stabilizes at p and also at q (at perhaps some other time), then it will stabilize eventually at r. Hence, semantics of these properties are close to the intuitive notion of stabilization. [14] demonstrates the use and significance of stabilization properties in the context of biological system analysis. However, our definition of stabilization captures a broader family of specifications. It includes $\mathbf{FG}p \Rightarrow \mathbf{FG}q \vee (\mathbf{FG}r \wedge \mathbf{GF}s)$ which may look contrived, but for example, [18] uses many such complicated stabilization properties for compositional deadlock analysis of micro-architectural communication fabrics. On the other hand, our definition includes many properties not intended to specify so-called stabilization behavior. For example, $\mathbf{GF}p$ or $\mathbf{GF}p \Rightarrow \mathbf{GF}q$.

The main motivation behind considering this broader subset of LTL is that we offer a short-cut L2S conversion, avoiding Büchi automaton construction, in a uniform way (due to the duality between $\mathbf{FG}$ and $\mathbf{GF}$ operators). The most significant applications of this class that we have encountered is "stabilization verification", and hence the name is coined for the family. (This name was inspired by [14]). Thus the L2S conversion proposed here may be applied for proving properties beyond the context of stabilization verification (eg. $\mathbf{GF}p \Rightarrow \mathbf{GF}q$).

The class of LTL properties defined as stabilization properties in this paper is a very important class of temporal properties extensively studied in the literature. It is related to so-called *fairness* specifications. Operators $\mathbf{GF}$ and $\mathbf{FG}$ are often called *infinitary operators* [19] and symbols F^{∞} and G^{∞} are used respectively instead [15]. The class itself (i.e. Boolean combination of GF-atoms) has been called *general fairness constraints* [16], [19]. As shown in [16], various notions of fairness like *impartiality* [21], *weak fairness* [20](also called *justice* [21]), *strong fairness* [20] (also called *compassion* [21]), *generalized fairness* [17], *state fairness* [22] (also known as *fair choice from states* [23]), *limited looping fairness* [5], and *fair reachability of predicate* [23] can be expressed by stabilization properties. These properties are used to exclude "unfair" counterexamples in liveness verification in both linear time and (fair) branching time paradigms. For liveness verification, we usually have a liveness property (the actual proof obligation) along with a set of fairness constraints. Liveness properties may not be stabilization properties. In that case we may need to construct the product of the system and the Büchi automaton of the (negation of the) liveness property before performing the L2S conversion. Interestingly, for many interesting applications as in [14] and [18], the liveness verification obligations fall entirely in the family of stabilization properties. For these applications, the simple L2S scheme proposed in this paper works. Note that some liveness properties like $\mathbf{G}(request \Rightarrow \mathbf{F}grant)$ are not stabilization properties, but also have a direct L2S conversion [24]. It is, therefore, an interesting question that under what more general conditions there exists a direct L2S conversion.

V. L2S Conversion for stabilization properties

It is important to understand that any counterexample to a liveness property (which must be an infinite trace) can be seen as a "lasso" like configuration with a finite handle and a finite loop. Therefore a liveness counter-example is a lasso which does not satisfy the property on the loop but satisfies all imposed fairness constraints on the loop.

In general, a liveness problem is converted to a safety problem by adding a loop-detection logic and property-detection logic on top of the product of the FSM of the original system and the Büchi automaton of the property to be verified. The loop-detection logic consists of a set of shadow registers, comparator logic, and an 'oracle'. The oracle saves the system state in the shadow registers at a non-deterministically chosen time. In all subsequent time frames, the current state of the system is compared to the state in the shadow registers. Whenever these two states match, the system has completed a loop. The non-deterministic nature of the oracle allows all such loops to be explored. The property verification logic checks if any of the liveness conditions are violated in any such loop and all fairness conditions always hold in the loop. This check is done as a safety obligation. For a more detailed exposition, see [24].

As mentioned, for some simple properties L2S conversion can be achieved while avoiding explicit Büchi automata construction. This is done by adding more functionality to the property detection logic. As presented in [24], these properties are $\mathbf{F}p, \mathbf{GF}p, \mathbf{FG}p, p\mathbf{U}q, , \mathbf{G}(r \Rightarrow \mathbf{F}q)$, and $\mathbf{F}(p \wedge \mathbf{X}q)$ (Table 1 of [24]). This approach, reviewed in Figure 5, depicts an L2S converted circuit for verifying the LTL property $\mathbf{F}p$.

In the next paragraph, we describe how this construction verifies $\mathbf{F}p$. In Section V-A we explain how to extend the ideas of Figure 5 for stabilization properties. Instead of presenting the liveness-to-safety conversion through Kripke structure-based representations (i.e. through explicit state machines based representations), we present the idea in terms of an actual circuit construction (i.e. through symbolic representation of the state space). Also, although we do not discuss it further, the same mechanism handles fairness constraints, which are always stabilization properties, so they just entail adding additional logic to the circuit for the monitor. For Kripke structure-based descriptions of liveness-to-safety conversion, see [24].

In Figure 5, `save` represents an additional primary input added to the circuit. This plays the role of the 'oracle'. When `save` is asserted for the first time, the current state of the circuit is saved in the set of shadow registers, and register `saved` is set. `saved` thus remembers that input `save` has been asserted and allows any further activity on `save` to be ignored. For subsequent time frames, `saved` enables the equality detection between the current state of the circuit and the state in the shadow registers. Clearly, signal `looped` is asserted iff the system has completed a loop. Signal `live` remembers if the signal p has ever been asserted. The safety property that the circuit verifies is, therefore, `looped` $\Rightarrow$ `live`. (In general this would be `looped` & `fair` $\Rightarrow$ `live`.) The block marked with "$\Uparrow$" represents this logical implication - the direction of the arrow distinguishes the antecedent signal from the consequent signal of the implication.

A. L2S *for stabilization properties*

In [24], the authors show how to do the L2S conversion for $\mathbf{GF}p$ and $\mathbf{FG}p$, which are GF-atoms. We demonstrate how to extend this to any Boolean combination of GF-atoms using an example, omitting a formal proof of correctness.

Consider a simple stabilization property ϕ of the form $\mathbf{FG}a \Rightarrow \mathbf{FG}b + \mathbf{FG}c$. An L2S converted circuit for this is shown in Figure 6. (For simplicity, we do not show any fairness constraints in the example.) Note that, signal `live` in Figure 5 monitors if signal p has ever been asserted from the very initial time frame. But for verifying $\mathbf{GF}p$, we need to monitor whether signal p has been asserted between the time when `saved` is set and the time when `looped` becomes true. Using this fact, the duality between $\mathbf{FG}$ and $\mathbf{GF}$ operators, and the Boolean structure $X_a \Rightarrow X_b + X_c$ of the given formula, we can derive the circuit of Figure 6. Logic that captures the Boolean structure of ϕ is marked with a dotted triangle in Figure 6. Hence, for any arbitrary stabilization property, we need to create monitors for individual GF-atoms and a crown of combinational logic on top of these monitors that captures the Boolean structure of the property. We can formulate the following theorem.

Theorem 1: For any stabilization property, the given procedure finds one counter-example if one exists.

(Proof Sketch) Any stabilization property can be transformed into another stabilization property with $\mathbf{GF}$ operators only. Let f be the Boolean structure in the negation of the given stabilization property. The procedure described above will create a monitor that will search for a lasso-loop where f is violated inside the loop. Since the procedure implicitly enumerates all possible cycles in the state space, it will detect a violating cycle if one such exists.

VI. Experimental Results

We implemented our L2S scheme for general stabilization properties in ABC and experimented with several designs of communication fabrics from industry. Our objective was to verify all stabilization properties defined for every structural primitive of the xMAS framework [18]. The properties, though local to each component, are verified in the context of the whole design in order to avoid explicit environmental modeling. BLIF models of the communication fabrics were generated by the xMAS compiler [11] from high-level C++ models. The L2S monitor logic was then created by ABC on these BLIF models. The xMAS compiler also generates SMV models from C++ models so that the LTL encoding of the stabilization properties can be verified directly on the SMV models using the NuSMV model checker.

We found that the ABC based L2S implementation has much better scalability than NuSMV. NuSMV can solve only toy designs while on the large designs of interest, it fails to produce a result. On the other hand, our tool works well even on large designs. For most cases, it produces a result almost immediately. For a few cases, initial trials could not produce a proof, but with the latest version of ABC using simplification, abstraction, speculative reduction, and property directed reachability (PDR) analysis [8], the proofs were completed. This observation supports the premise that the use of highly developed safety techniques can pay off for liveness verification.

Experimental results are shown below. Among all the local properties that the xMAS compiler generated, we provide results for the most challenging one. Call this property ψ; it is defined for a FIFO buffer, and has the following LTL form

$$\psi := \mathbf{FG}(\neg a) \Rightarrow \mathbf{FG}(\neg b) \vee \mathbf{FG}(c)$$

where a, b, and c are appropriate design signals (i.e. interface signals of a FIFO buffer). Table 1, 2, and 3 compare the performance of ABC with NuSMV on small examples. These examples are instances of communication fabrics or submodules thereof, and are explained in full detail in [10]. *simple_credit* and *simple_vc* (Table 1 and 2, respectively) are designs corresponding to Figure 4 and 5 of [10], and *simple_ms* (Table 3) is a much simpler version of the design shown in Figure 6 of [10]. Note from the table, how performance of NuSMV degrades even for small designs. For large designs, NuSMV could not finish for any single instance of ψ.

Since ψ is defined for a FIFO buffer and the xMAS compiler created one instance of ψ for each FIFO buffer, the

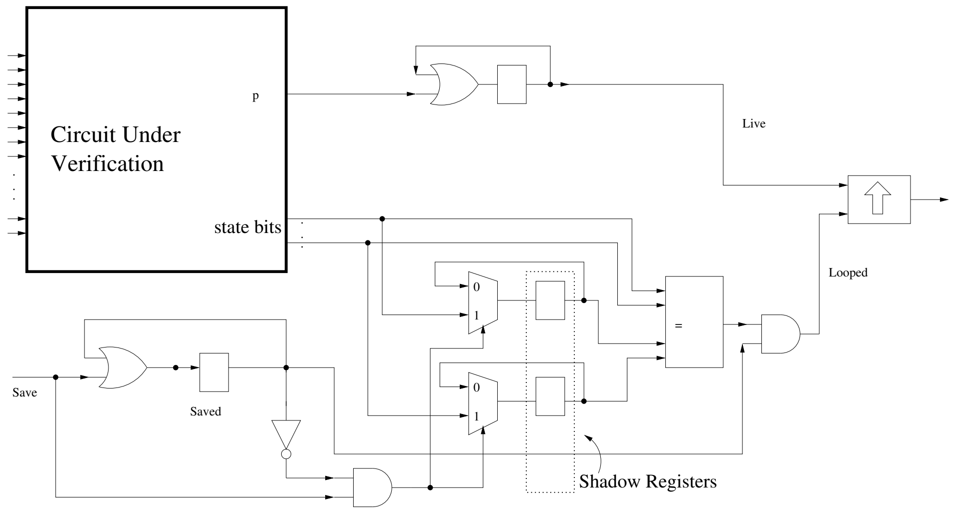

Fig. 5. Liveness-to-safety transformation for $\mathbf{F}p$

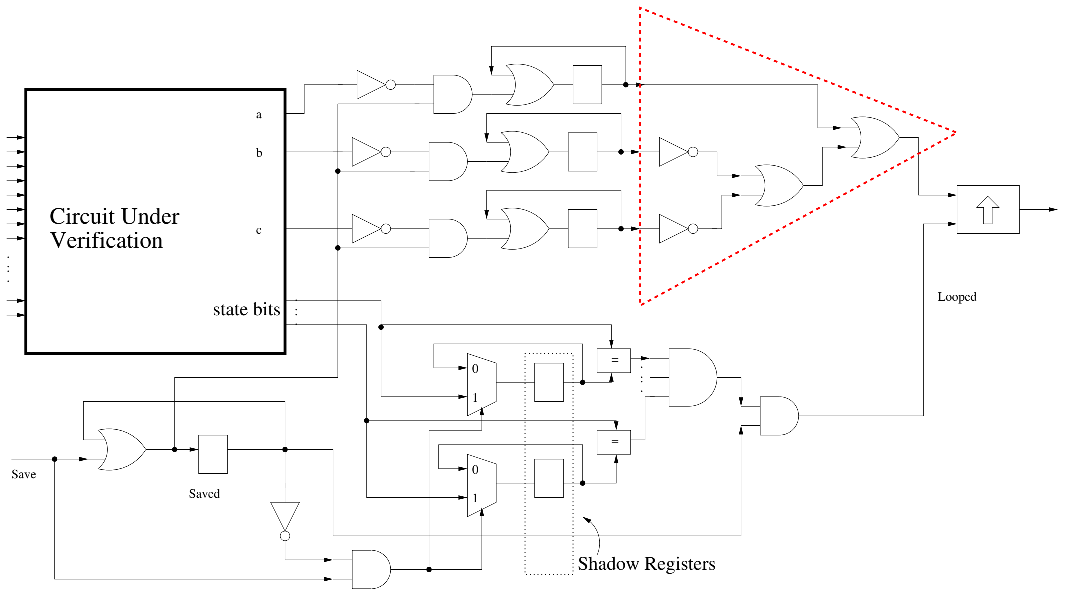

Fig. 6. L2S for stabilization property $FGa \Rightarrow FGb + FGc$

Prop #	ABC	NuSMV
	(sec)	(sec)
0	0.25	0.115
1	0.05	0.14
2	0.02	0.09

TABLE II
SIMPLE_CREDIT

Prop #	ABC	NuSMV
	(sec)	(sec)
0	0.09	33.23
1	0.07	31.8
2	0.06	39.57
3	0.03	16.46
4	0.5	41.37
5	0.03	16.89

TABLE III
SIMPLE_VC

Prop #	ABC	NuSMV
	(sec)	(sec)
0	0.03	431.5
1	0.12	379.59
2	0.8	471.36
3	0.8	385.67

TABLE IV
SIMPLE_MS

number of ψ instances is the same as the number of FIFO buffers. For example, the designs corresponding to Table 1, 2, and 3 above have 3, 6, and 4 FIFO buffers, respectively.

We also experimented on two large communication fabrics of practical interest [10], [18]. One has 20 buffers and the other has 24 buffers. 19 out of 20 of the first design and 23 out of 24 from the second design were proved by ABC by a light-weight interpolation engine within a worst case time of 5.83 seconds (most were proved in less than a second). Light-weight interpolation could not prove one instance from each design. These were proved using advanced techniques from ABC's arsenal of safety verification algorithms. For example, ABC took a total of 217.2 seconds to prove one of these harder properties. In this time span, ABC first did some preliminary simplification, then it tried interpolation, BMC, simulation and PDR in parallel for a time budget of 20 seconds. But this attempt failed and it moved on to further simplification by reducing the design using localization abstraction and speculation. It ran interpolation, BMC, simulation, BDD-based reachability and PDR engines in parallel both after abstraction and speculation, using an elevated time budget of 100 seconds and 49 seconds respectively. The iteration after abstraction could not prove the property, but the iteration after speculation managed to prove it with the PDR engine, which produced the final proof in 7 seconds.

VII. Conclusion & Future Work

We have developed a tool, VeriABC, which allows us to access real industrial designs written in SystemVerilog or VHDL and to process them into the AIGER format. The result can be used for synthesis and verification using a tool like ABC. We described how the RTL processing is done using the commercial front-end, Verific. SVA assertions are also processed by Verific, and VeriABC creates a separate file of equivalent LTL formulas. We showed an application of this to property checking, where ABC was enhanced to convert a subset of LTL into a circuit structure, thus effectively allowing liveness checking in ABC.

The use of a stable, supported and complete language processing tool like Verific, allows academics access to real industrial designs, without going through the hassle and daunting task of building their own equivalent tool. Liveness property checking is a growing interest in industry, and our enhanced ABC with a front end that automatically converts to a circuit structure for liveness checking, can use the advanced safety property methods of ABC.

In the future, the development of VeriABC will allow us to extract higher level constructs from SystemVerilog and VHDL by accessing Verific's parse trees. These constructs can be passed on using an extended AIGER format to an enhanced ABC, which will use this information in synthesis and verification.

VIII. Acknowledgment

This work has been supported in part by SRC contract 2057.001 and our industrial sponsors: Abound Logic, Actel, Altera, Atrenta, IBM, Intel, Jasper, Magma, Oasys, Real Intent, Synopsys, Tabula, and Verific.

References

[1] ABC - a system for sequential synthesis and verification. Berkeley Verification and Synthesis Research Center, http://www.bvsrc.org.
[2] Aiger, http://fmv.jku.at/aiger/.
[3] Icarus verilog, http://iverilog.icarus.com.
[4] Verific Design Automation: http://www.verific.com.
[5] K. Abrahamson. Decidability and expressiveness of logics of processes. In *PhD Thesis, University of Washington*, 1980.
[6] J. Baumgartner and H. Mony. Scalable liveness checking via property-preserving transformations. In *DATE*, pages 1680–1685, 2009.
[7] A. Biere, C. Artho, and V. Schuppan. Liveness checking as safety checking. In *In FMICS02: Formal Methods for Industrial Critical Systems, volume 66(2) of ENTCS*. Elsevier, 2002.
[8] A. Bradley. Sat-based model checking without unrolling. 2011.
[9] R. Brayton, G. D. Hachtel, A. Sangiovanni-vincentelli, F. Somenzi, A. Aziz, S. tsung Cheng, and S. Edwards. Vis : A system for verification and synthesis. pages 428–432. Springer-Verlag, 1996.
[10] S. Chatterjee and M. Kishinevsky. Automatic generation of inductive invariants from high-level microarchitectural models of communication fabrics. In *CAV*, pages 321–338, 2010.
[11] S. Chatterjee, M. Kishinevsky, and U. Ogras. Modeling communication micro-architectures (with one hand tied behind your back). *Intel Technical Report*, 2009.
[12] S.-T. Cheng and R. K. Brayton. Compiling verilog into automata, 1994.
[13] E. Clarke, O. Grumburg, and D. Peled. Model checking. 2000.
[14] B. Cook, J. Fisher, E. Krepska, and N. Piterman. Proving stabilization of biological systems. 2011.
[15] E. A. Emerson. Handbook of theoretical computer science (vol. b). chapter Temporal and modal logic, pages 995–1072. MIT Press, Cambridge, MA, USA, 1990.
[16] E. A. Emerson and C.-L. Lei. Modalities for model checking: branching time logic strikes back. *Sci. Comput. Program.*, 8:275–306, June 1987.
[17] N. Francez and D. Kozen. Generalized fair termination. In *Proceedings of the 11th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, POPL '84, pages 46–53, New York, NY, USA, 1984. ACM.
[18] A. Gotmanov, S. Chatterjee, and M. Kishinevsky. Verifying deadlock-freedom of communication fabrics. 2011.
[19] R. Hojati, R. K. Brayton, and R. P. Kurshan. Bdd-based debugging of design using language containment and fair ctl. In *Proceedings of the 5th International Conference on Computer Aided Verification*, CAV '93, pages 41–58, London, UK, 1993. Springer-Verlag.
[20] L. Lamport. "sometime" is sometimes "not never": on the temporal logic of programs. In *Proceedings of the 7th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '80, pages 174–185, New York, NY, USA, 1980. ACM.
[21] D. J. Lehmann, A. Pnueli, and J. Stavi. Impartiality, justice and fairness: The ethics of concurrent termination. In *Proceedings of the 8th Colloquium on Automata, Languages and Programming*, pages 264–277, London, UK, 1981. Springer-Verlag.
[22] A. Pnueli. On the extremely fair treatment of probabilistic algorithms. In *Proceedings of the fifteenth annual ACM symposium on Theory of computing*, STOC '83, pages 278–290, New York, NY, USA, 1983. ACM.
[23] J. P. Queille and J. Sifakis. Fairness and related properties in transition systems a temporal logic to deal with fairness. In *Acta Informat*, pages 195–220, 1983.
[24] V. Schuppan and A. Biere. Efficient reduction of finite state model checking to reachability analysis. *Int. J. Softw. Tools Technol. Transf.*, 5(2):185–204, 2004.