Towards Verifying Large(r) Systems: A strategy and an experiment

P.A.Subrahmanyam
AT&T Bell Laboratories, Rm 4E-530, Holmdel, N.J. 07733
electronic-mail: subra@research.att.com

Abstract

Despite some of the impressive results quoted in recent verification literature, the verification of even modestly sized "real" industrial designs is not yet feasible on a routine basis. The goal of the work discussed here is to enable the verification of large(r) real systems than currently feasible with any one of the available techniques, and to dovetail the verification methodology with the an underlying design methodology. The specific design methodology considered here is targeted towards the custom design of digital signal processing architectures. Two important attributes of this class of designs are (1) custom-crafted leaf cells, and (2) significant data path components. Our strategy is to partition the subsystems involved into different categories that can be handled by different techniques, and use a divide-and-conquer paradigm. The complexity introduced by data paths is addressed by automating the abstraction of some of the natural equivalences induced, and exploiting this in the context of an extended finite state machine formalism. Specifically, we illustrate how it is possible to exploit the distinction between data and control in implicitly specified state machines (HDLs), and comment on useful abstractions in the presence of symmetry. We illustrate some aspects of the strategy via an example. We also suggest some evolutionary (rather than revolutionary) changes in the design methodology that enable the existing state of art in verification to be better exploited in practise.

1 Introduction

Over the last few years, there has been a rapid (even surprising) evolution in the capabilities of techniques for formal verification of hardware. Examples include: state machine equivalence checking[6], symbolic simulation[2], automata-theoretic language containment[10], theorem proving based on various flavors of logic/algebra[8], and model checking[4]. It is reasonable to expect that the maturity and domain of applicability of most techniques will continue to improve, albeit at different rates.

Despite some the impressive results quoted in literature, the verification of even modestly sized real industrial designs is not yet feasible on a routine basis. For example, state-based methods that use binary decision diagrams[1] (BDDs) can handle some state spaces having on the order of 2^{120} states[4]; a more accurate metric in practice is the space needed by the BDD representation for a particular circuit rather than the size of the state space. However, the observation that one requires merely four 32-bit registers in a design to approach such a limit is rather sobering. More generally, it is recognized that each of the techniques listed above has its strengths and weaknesses. Often, some aspect of even a modest sized real system tends to defeat the specific technique being applied. It is therefore necessary

to evolve more sophisticated verification methodologies to address practical verification tasks.

We believe that it is important for any practically viable verification methodology to be closely coupled to (as opposed to being completely divorced from) a "real" design methodology i.e., one that is in actual use, or that can be reasonably enforced in practice. One implication of this observation is that a verification methodology that assumes a purely top-down approach to design is unlikely to be widely applicable in the near term: this is because a strictly top-down development methodology is not widespread in practise (despite several papers over the last 20 years that have advocated it, and several more that continue to advocate it).

Further, we believe that in the short to intermediate term, fully automated verification techniques will be absorbed into design methodologies and associated tools and environments more readily in comparison to interactive verification techniques, e.g., theorem proving based techniques.

The overall goal of the work discussed here is the verification of large(r) "real" systems than currently feasible with any one of the existing techniques. The underlying design methodology considered here is targeted towards the custom design of digital signal processing architectures[14]. Two important attributes of this class of designs are (1) the use of custom-crafted circuits at the lower levels e.g., leaf cells; and (2) significant data path components.

1.1 Strategy and Approach

The objective of a verification task is usually either (1) to determine the consistency between two descriptions of a design (usually generated independently), or (2) to ascertain whether a given design description satisfies a specified property. The approach discussed here is motivated by the following observation: many medium-to-large scale systems often consist of subsystems that can be categorized as being control logic, data paths, regular iterative subsystems (e.g., systolic arrays), or specialized subsystems (e.g., memories). These subsystems have very different "verifiability" attributes in that different techniques are useful for each of these classes of subsystem. Our strategy is to leverage the strengths of different approaches to verify larger systems than each can individually verify, and to appropriately couple this into the underlying design process.

In brief, given a layout, and the associated structural hierarchy, we delineate a strategy for abstracting the behavior of the components of the layout into modules that have increasing size (number of transistors). The abstraction process consists of combinational logic extraction, state machine abstraction, and the computation of state machine products. When the computational complexity of such an abstraction process exceeds the available resources, we establish a correspondence between the abstracted description and a second description arrived at by a top-down decomposition process. In order for this correspondence to be established, the two descriptions are required to have isomorphic submodule boundaries.

More specifically,

- The complexity introduced by data paths is addressed by automating the abstraction of some of the natural equivalences induced by data paths, and exploiting this in the context of an extended finite state machine formalism. Such a formulation corresponds closely to many hardware description languages (HDLs) in common use. If applicable, algebraic structure can be used to further reduce the complexity.
- Modified forms of symbolic simulation[2] are used to verify iterative subsystems and memories, and
- FSM-based verification, language containment and model checking techniques are used for verification of the control logic and the abstracted models.

Thus, our strategy is to analyze the subsystems involved, partition them into different categories that can be handled by different techniques, and use a divide-and-conquer paradigm.

The verification methodology developed here enables independent (and concurrent) development of the top-down and bottom-up design threads if needed. By not *requiring* a strictly top-down "one-shot" path to the design, it supports design methodologies that more closely reflect practise. Additionally, it enables "property analysis" to be done at any level of the design hierarchy, and preserves the integrity of the analysis done at the higher levels (by virtue of the coupling of the descriptions arrived at by the top-down and bottom-up design processes).

We also suggest some evolutionary (rather than revolutionary) changes in the design methodology that enable the existing state of art in verification to be better exploited in practise. The techniques advocated are targeted at (1) evolving the bottom-up, middle-out, and iterative design processes, and (2) synergistically merging bottom-up design and verification techniques with top-down design and verification techniques.

The rest of this paper is organized as follows. Section 2 summarizes a methodology used within AT&T for the design of a subclass of digital signal processing (DSP) circuits, and outlines a verification methodology that dovetails with this design methodology. Section 3 illustrates the use of this approach via an example. Section 4 elaborates on the formalism used for reducing the complexity of analyzing circuits that have large data paths, and comments on the computation of a minimized transition system described by an HDL program. Specifically, we illustrate how it is possible to exploit the distinction between data and control in implicitly specified state machines (HDLs), and comment on useful abstractions in the presence of symmetry in systems. Finally, Section 5 summarizes some of the main points of the paper, and mentions ongoing work.

2 The Design/Verification Methodology

In this section, we first highlight some of the distinguishing features of an existing methodology that is used for designing and debugging a subclass of custom signal processing circuits within AT&T. We then discuss a verification strategy for such designs. The

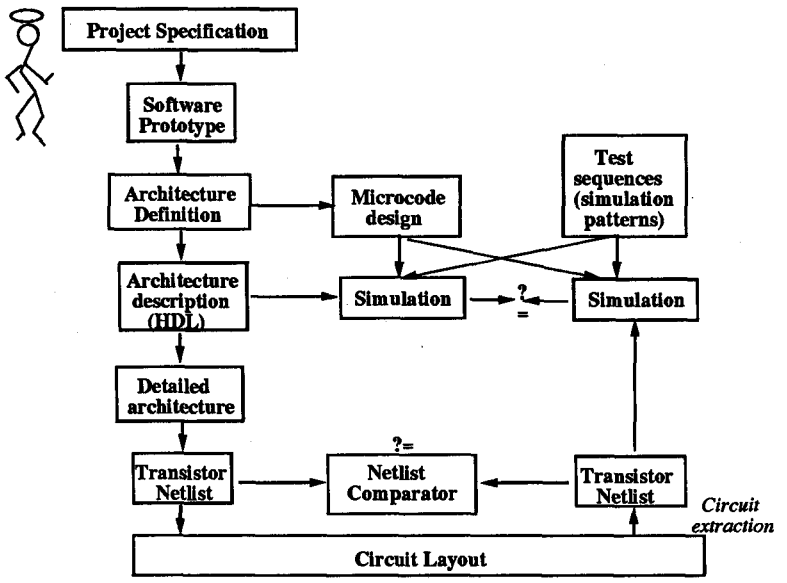

Figure 1: A prototypical design scenario motivating formal verification

verification strategy in turn suggests some alterations to the extant design methodology that can assist certain verification tasks, while preserving the key strengths of the methodology.

2.1 Current Design/Debugging Paradigm

The design/debugging methodology we will use as a starting point here is used for developing custom DSP systems and assessing their logical "correctness" within AT&T (see Figure 1). It is essentially simulation-based, and as follows. A high level (C) program is used to model and prototype the functionality of the overall system. This model is also used to generate the test vectors that are used for simulating/debugging more detailed hardware models, e.g., at the register-transfer level (RTL). Such models are usually expressed in a C-based hardware description language (HDL). Many of the component chips are custom designed, and incorporate hand-crafted cells. A low-level structural description (called an LSL description) is (manually) generated to reflect the cell-level structure of the actual design. This LSL description is graph-matched to the netlist extracted from layout so as to establish a 1-1 correspondence; heuristic graph matching techniques are used in this task. The correspondence between the LSL-model and clock-level accurate HDL descriptions is approximated by running through a set of simulation vectors on the two descriptions, and checking for the consistency (equivalence) of the outputs. Additionally, and somewhat independently, lower-level timing simulations are done at the chip level to reduce the probability of timing errors in the fabricated chip.

Since the overall systems and component chips are quite complex, two familiar problems are manifest in such a simulation-based debugging paradigm. First, it becomes

quite cumbersome to generate/obtain a reasonable set of test vectors. Secondly, it is quite difficult to obtain a reasonable degree of coverage with a set of vectors that can be simulated in a reasonable time span. Formal verification is therefore obviously appealing in this context, but not yet practical using existing techniques and tools.

2.1.1 Distinguishing Characteristics of the Design Paradigm

There are some aspects of the design process described above that provide significant leverage[14]. One of its distinguishing traits, pertinent to the development of a synthesis tool, is the (human) exploration of an application specific solution space that spans algorithms, architectures and circuit design/layout. Another feature, pertinent to the development of a verification methodology, is the use of custom circuit design and layout at the lower levels. The compactness of the resulting custom crafted layouts often enables associated improvements in both latency and throughput, and potentially a decrease in power consumption. A third noteworthy characteristic of this class of signal processing applications is that their data path component often tends to be significant and iterative.

In formulating a verification methodology, the guideline we follow is that any modification required of the underlying design methodology should preserve its key strengths. It is permissible, however, to incrementally alter other (i.e., "non-key") aspects in order to better assist the verification task. In this instance, human intervention/involvement in low-level design is a key component of this technique. Consequently, *completely* automated synthesis and layout are unlikely to be deemed acceptable, since this would arguably compromise the quality of the resulting designs (given available synthesis and layout tools).

2.2 A Formal Verification Methodology

We next outline a verification methodology that preserves the main strengths of the design methodology discussed above. For each step, we comment on the input information required, and indicate whether this is already available in the existing design process, or whether modifications are required.

Note that the primary aim of the verification process is (1) to verify that the functionality of the circuit layout is consistent with the clock accurate RTL-level C(HDL) model and (2) to verify that the circuit/system description has a specified set of properties. The discussion below has a "bottom-up" flavor, with the primary focus being on the first mentioned task.

Comparing the netlist extracted from the layout with the specified (low level) LSL netlist. As a first step, the netlists extracted from the layout and the hierarchical (low-level) structural description (manually generated) are compared for one-to-one correspondence.

The inputs required for this process are: the circuit layout, the extracted netlist and the low-level structural description (henceforth referred to as the hierarchical LSL description).

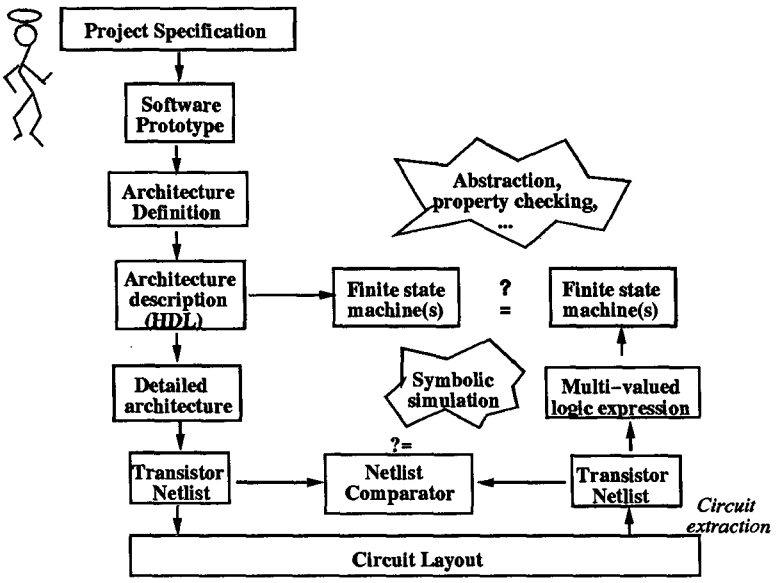

Figure 2: Some aspects of the Verification Methodology

These components are available in the current design process, and the netlist comparison task is done automatically using heuristic graph-isomorphism checkers.

Methodological Modifications. It is possible to avoid the labor-intensive human generation of low-level structural descriptions that is required in the current design methodology. This is because the extraction of the gate level equivalents of the layout (and/or their behavioral description) can largely be automated.[1] Such behavior extraction, however, benefits from and sometimes might require a demarcation of the structural hierarchy of the layout. It is noteworthy that this is painless for the designer to provide as input, since both the layout tool and/or the designer have ready access to this information. It should also be noted that such information (relating to the structural hierarchy) is often difficult to reverse-engineer i.e., to extract unassisted from the layout.

Comparing the behavior of the specified structural description with the extracted netlist behavior. In the next step, the *behaviors* of the cells in the structural hierarchy and the circuit layout are compared.

The inputs required for this step are (1) the LSL description for the cell and (2) the layout fragment corresponding to this LSL description. The first item is available as part of the specification. While the overall layout is always available, the extra information required by second item is the demarcation of the cells in the layout that correspond to the LSL description. Such identification may be done without much overhead by either a tool

[1]The only exception to this are layouts for hand-crafted circuits that defeat analysis using switch-level models. Such circuits need lower levels of simulation/analysis in order for their functionality to be extracted.

(in case the layout is automatically generated) or by the designer (who is cognizant of the structural hierarchy).

The behavior of each cell is extracted at the combinational or finite state machine level[13]. This is formally verified against the behavior corresponding to the LSL description. The comparison is performed automatically.

Moving up in the hierarchy by behavioral abstraction. Using a technique such as that described in [13], the state-machine behavior of increasingly larger clusters of transistors in the structural hierarchy can be abstracted. The behavior of this collection of transistors can be verified against a corresponding structural (LSL) or behavioral HDL description.

This step yields a partition of the layout into a set of medium grained modules. By a "medium grained module" we are here alluding to a collection of a few hundred to a few thousand transistors having meaningful behavior.

Matching up the Behavioral and Structural Hierarchy. The preceding step yields a description consisting of a system of interacting finite state machines (FSMs) that are medium grained. This description can be compared against a corresponding HDL model (having no direct structural correspondence, but possessing identical submodule boundaries) that is generated by a "top-down" design development process. This step thus provides the meeting ground for the top-down and bottom-up design paths.

More specifically, the layout/netlist segments of the design obtained as a result of the preceding step can be divide into two classes: (1) segments that have a corresponding (but independently specified) behavioral description; and (2) those that do not. In the first case, we consider the "immediate" verification problem as having been solved, since equivalence has been established between the specified behavior (HDL model) and the associated layout segment. In the second case, a combination of alternative techniques can be used for verification (including variants of symbolic simulation, symmetry annotations, and automatic abstractions). If the set of available techniques proves inadequate, then we require the top-down description to be refined further to provide behavioral descriptions for modules that correspond to the structural hierarchy that has been abstracted. This then completes the immediate verification task in both cases.

*Useful Methodological Modifications.*As an aid to the verification process, it is useful to replace/augment the individual simulation vectors that are used in traditional simulation by symbolic expressions capturing the property being checked for. Such properties can then be verified using variants of symbolic simulation techniques.

Higher level verification/Property checking. At this stage, the behavioral description can be compared with other descriptions at higher levels of abstraction, and/or can be checked for properties using a variety of techniques. Such techniques include automated model checking and automata-theoretic approaches, as well as more interactive methods. A discussion of these tasks can be found elsewhere, and is beyond the scope of this paper.

However, we will comment on the techniques used for addressing the complexity that arises due to the presence of a large state space.

Useful Methodological Modifications. As widely advocated, integrating the top-down design and verification processes expedites the overall development cycle.

2.3 Property Verification

In the early stages of the verification process discussed above, the focus is on checking for the consistency between an HDL specification and the layout. Once the system behavior is abstracted (e.g., as a set of interacting state machines, or some alternate form), it is often useful to check if the system description satisfies some set of properties that are not usually explicit in the HDL description. Examples of such properties include liveness, safeness, or fairness properties. Model checking techniques enable/accomplish this by exploring (searching) the state space of the system description. The property to be checked can be specified, for example, using a temporal logic formula[4], an automaton[10], or some other means.

One of the main problem that arises in this context is that the complexity of naively exploring the state space can easily overwhelm the available computational resources. A common way to combat this problem relies on user-supplied homomorphisms to abstract the initial description to another description that has a much reduced state space, but such that a search of this reduced search space is still sufficient. The drawback with this approach is that it requires the user to provide the homomorphisms: this is an added burden placed on an "engineer" not necessarily well-versed in formal methods. It is therefore desirable to automate their discovery to the extent feasible.

Here, we cope with the problem using a variety of techniques:

- Some forms of data-path homomorphisms are automatically extracted.
- An efficient algorithm for computing the *reachable* state space, given a partition of the states induced by the equivalence relations on states/inputs is used to reduce the complexity of the problem.
- User annotations related to the symmetry of data path components are used to reduce the state space that needs to be searched.

We will briefly comment on some of the techniques used in Section 4. Of course, it is still possible to overwhelm the available resources, in which case additional user guidance is needed.

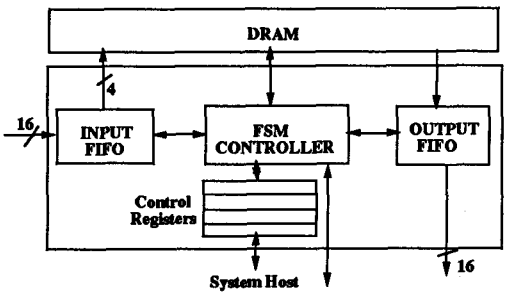

Figure 3: A FIFO Controller

3 An Example: A FIFO Controller

The example we use here to illustrate some of the concepts discussed in this paper is a FIFO controller (see Figure 3), which in turn is a subsystem of a larger video-codec system. The FIFO controller (denoted FIFO-C) consists of

- an input FIFO of size 4K (1024 * 4) bits,
- an output FIFO of size 2K (512 * 4) bits,
- an external DRAM (1 MB) that serves as a FIFO buffer,
- a finite state controller (denoted FC-FSM) that interfaces to the host, the input and output FIFOs, and the DRAM.

The task of the FIFO controller is to accept 16-bit data from the input bus, store it into a local input FIFO (V1), use an off-chip DRAM (with a 4-bit wide I/O bus) as an overflow FIFO (supplying refresh strobes for the DRAM at the required intervals), transfer data from the DRAM into the output FIFO (V3), and eventually provide the data to the system host when requested.

The first point to observe is that both the control logic and the data path taken together have approximately 7000 bits of memory. Thus, at first blush, the FIFO controller has a state space on the order of 2^{7000} nodes. However, despite its modest size, the FIFO controller system description cannot be directly verified (or analyzed) by any verification tool that we are aware of.

3.1 Desired I/O behavior of the FIFO Controller

The basic property to be proved about the FIFO controller behavior is that data read into the input FIFO will eventually be output at the output FIFO when requested by the system (i.e., after an appropriate sequence of system commands).

Additionally, several secondary properties specific to the architecture can be verified. Such properties include, for example,

- "the DRAM is refreshed at the required intervals";
- "the read, write and refresh modes alternate as prescribed by the DRAM interface";
- etc.

There are some caveats. The input/output data rate is assumed to be such that the input FIFO/DRAM/Output FIFO combination do not overflow. The actual sizes of the FIFOs are based on some application specific information which ensures, with high probability, that such overflow will not occur. However, an additional safety catch has been engineered in. When the DRAM is half full, an interrupt signal is sent to the external host that results in either the input data rate being reduced or the output FIFO being drained. If this interrupt is ignored, the FIFOs can overflow, but this is not construed to be an error.

3.2 Identifying Subsystems

The first step in the verification strategy involves identifying the various components of the system and partitioning them into categories that will be addressed using distinct techniques.

The major subsystems of the FIFO controller system are the 2 internal FIFOs (having significant data path components), and the controlling finite state machine. Since the input and output FIFOs are somewhat similar, we will here consider only the input FIFO and the FSM controller.

- The FSM controller FC-FSM is basically a random collection of logic with no particular structure. The behavior of this logic collection is extracted hierarchically, first at the cell level, and then at the module boundary level. This is verified against the corresponding behavior expressed in an HDL, and translated into an FSM representation.
- The input FIFO description has data paths that can be abstracted out for the purposes of reasoning at the level of the overall controller. The abstracted machine description can also be used for analysis using higher level tools, e.g., model checking and language containment.

3.3 Verification of the Controller FSM

The steps described in Section 2.2 are used in the verification of the FSM controller FC-FSM:

- A bottom-up abstraction of the behavior of the layout is performed, conforming to the hierarchical specification.
- The equivalence of cell behaviors is established using FSM equivalence techniques.

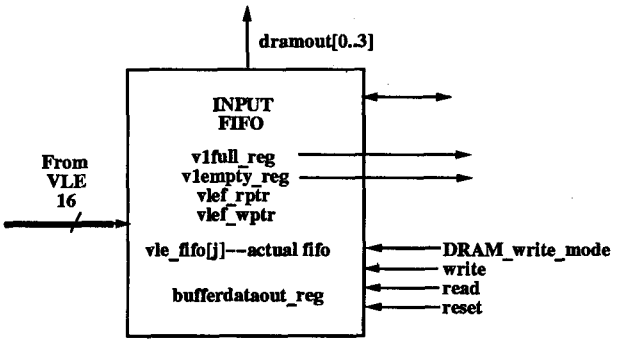

Figure 4: Input FIFO: Block diagram

While many of the cells abstracted did not pose any problems, some points are worth noting.

- A few of the cells have a behavior that is sensitive to the relative transistor sizes. This was discovered only as a consequence of the verification task having *failed* in an instance when an input having default transistor sizes was used.
- A set of cells that are designed to function as edge detectors have a behavior, which, when viewed in isolation, causes the output pulse to be unobservable when the output is sampled only after the rising clock edge. The expected behavior is indeed observed when the output is observed on the falling clock edge. However, when this cell is used as part of a larger circuit, the singularity does not cause any problems.

The BDD was successfully constructed for the initial stage of the abstraction of the overall controller, but there was inadequate memory available for the computation to complete. Certain optimizations in the BDD package are expected to enable this computation to proceed to completion. The FSM-based verification of the controller then follows this stage.

3.4 Input FIFO: Modeling and Verification

The input FIFO has a structure that is shown in Figure 4. In essence, it consists of

- a 16-bit wide input bus;
- a 4-bit wide output bus;
- status registers indicating whether the FIFO is `full` or `empty`;
- Internal read and write pointers. Each 16-bit value input is read into 4 words of the FIFO, each 4-bits wide.

The block diagram for the input FIFO is shown in Figure 4. The initial HDL-model of this description yields the following variables as being "control" variables: the read and write command lines, the full/empty status registers, the write control for the DRAM, and the clock. This identification of control variables used in the construction discussed in Section 4.

3.4.1 FIFO Properties

The properties verified of the FIFO are of the form:

Initialization. If the reset signal is raised, then the full, empty, data out, lines are set appropriately;

Data Delivery. If the data is input (written into the FIFO), then it is output (in the same order);

Fullness, Emptiness. That the fullness and emptiness of the FIFO are properly indicated (as a function of the size of the FIFO and the number of data items held).

There are a few peculiarities of the input FIFO that are complicating factors. The input data is 16 bits wide, while the output data is 4 bits wide. This can be handled in several ways. One alternative is to view the 16-bit input data as being made up of four 4-bit pieces, and then view a single write into the FIFO as a series of 4 writes. (Other alternatives are different variants of this scheme.)

3.4.2 FIFO Verification

We will here restrict our comments to the FIFO behavior without the 4 way interleaving mentioned above. The initial HDL specification of the FIFO is analyzed to yield the control variables. The resulting specification has a form that can be cast as an automaton analyzable by Cospan[10]. Appropriate abstractions were then used to perform the needed property verifications.

Examples of FIFO Abstractions A FIFO consisting of N words each of width W bits would yield a state space of size 2^{N*W}. The proof of correctness of a FIFO benefits from abstraction. Some the (somewhat standard) abstractions of a queue that were used in different contexts of the FIFO-C verification are listed below[10].

- To verify data-delivery in a data-independent system, we verify first the validity of collapsing the data items to 2 tokens (one modeling the item to be delivered, the second modeling all others), and establish this as a homomorphism. This model can then be reduced to yield a model having a much smaller number of states.

- At the level of the overall system, when the functionality of a FIFO, say Q, is only peripheral to the property which is to be verified, we replace the model Q with a 1-state model Q' which non-deterministically outputs from its single state all the output tokens of Q. Let L(Q) denotes the language accepted by the automaton Q. As $L(Q) < L(Q')$, any (regular) property we can verify using Q' therefore holds with Q as well; however, Q' is *too small* to hold the information necessary to verify, for example, data-delivery.

In the next section, we indicate how we can automatically extract some of the data path abstractions based on the structure of an HDL description.

4 Dealing with data paths

4.1 HDLs and Extended Finite State Machines

Systems are described (in the context of this paper) either by using an HDL, or are represented internally, e.g., using transition relations. An HDL program is expressed compactly in a notation that uses (typed) variables, and operations (e.g., +, *) on the variables. When such a description is converted into an "explicit" state machine representation, for example by enumerating all of the possible values for the variables, the size of the resulting representation (i.e., the state space) can increase significantly; indeed, it may even become infinite (depending upon the variable types allowed by the HDL). This is widely referred to as the "state explosion problem". Thus, while the reachable space may itself be small, there may still be severe limitations in the analysis that is feasible.

A common reason for the "non-essential" growth of the state space is the presence of a significant data path component in a circuit. The data path component of some circuits can be significant. However, in many cases, the actual data values in the "data path" of a hardware system are of no consequence to the property being analyzed. Thus, it is often possible to abstract the large set of values 2^n in an n-bit data path to a small set (perhaps even one) for the purposes of analysis. (Note that this is not directly feasible for arithmetic circuits.)

More generally, while the actual state space may be large, many states may be equivalent to each other (where the actual definition of equivalence is context dependent). In principle, the complexity of the analysis should at least be bounded by the number N of reachable equivalence classes. (The problem in general is PSPACE-hard for finite domains and undecidable for infinite domains.) Often N tends to be much smaller than either the total number of reachable states or the total number of equivalence classes.

There are several ways to combat the increase in the state space caused by the presence of data paths. These include:

- Writing the HDL model so as to explicitly separate the data path computations and the underlying control logic, and then dealing with the two components independently.

This option can exploit model checking techniques, for example. The drawback with this approach is that the rewriting of the HDL model is typically manual.

- Using user specified abstractions (homomorphisms) in the context of a language/automata theoretic framework[10].
- Using an extended finite state machine formalism (suited for HDLs) for analyzing the reachable space, and exploiting data path equivalences to reduce the size of the space to be explored.

We will briefly elaborate on the third option below.

4.2 Reducing State Transition Systems

A state transition system M is defined by a tuple $< S, R, I, T, O, Out >$ consisting of a set of states S, an equivalence relation R on S (that serves to induce a partition P on S), a finite set of input actions I, and a set T of transition relations $N_a \subseteq S \times S$ for each action $a \in I$. This definition allows the system to be nondeterministic. In the case of a deterministic system, the next state for a specific input is uniquely defined, and the transition relation can be defined by a map $T : S \times I \rightarrow S$. The other components of M optionally include an output domain O and output function Out that can be specified either as a deterministic map $Out : S \times I \rightarrow O$ or as a nondeterministic relation $Out_a \subseteq S \times O$ for each input $a \in I$. Often, the relation Out can be treated in a manner similar to the relation T (it may even be merged with T for certain computations).

The set of states in this definition may be potentially infinite, although in the case of hardware systems the set is large but not infinite. The equivalence relation R on S is intended to capture the set the equivalences relevant to a specific application context. For example, if a set of system states $\{s_1, ..., s_n\}$ differ only in the value of data associated with a data path component (e.g., values on a bus), and we do not care about the actual values, then this set of states may be coalesced by R.

We are interested in the reduced or "minimized" system whose states are given by the quotient S/R. Further, while S may be potentially infinite, we require S/R to be finite. If R preserves the transition relation for M, the quotient algebra M/R is well defined. Given an initial partition of S, we can iteratively converge to S/R by the algorithm discussed below. The primary advantage in doing this lies in the fact that the representation complexity is significantly reduced, since M need not be constructed in its entirety.

Pictorially, a transition system can be though of as a labeled graph. The nodes of this graph correspond to the states S of the system. There is an arc from state s_i to s_j labelled with input $a \in I$ if, the state s_i can transition to s_j on input a.

Correlating implicit and explicit representations

State Space interpretations. The state space of an HDL program can be viewed as being defined by a vector of variables taken from a domain $D = D_1 \times D_2 \times ... \times D_n$. The actual

interpretation of the domains typically depends upon the HDL (and the program) being considered. We give 2 examples.

- In the case of HDL such as VHDL[11], the implicit state space is defined by the location (program counter, or labelled statement), the values of variables that denote internal registers, buses, etc., and the inputs of this process (equivalently, the outputs of the processes that communicate with this process).
- In case of a language such as S/R[10], where the state space is explicit, the state may be interpreted as $< \textit{symbolic state}, \textit{variable values} >$, where the variable values can refer to both internal states as well as inputs (output selections of other processes). In this case, we will denote the state space by $SS \times D$ for convenience.

The data type domains D_i may have further algebraic structure that can at times be exploited. For example, integers modulo 8 can be represented using the set of values 0..7. Further, if 0 is an initial value, and only increments of 4 are allowed, then the reachable values in this domain can be represented by {0,4}.

Computing the state transition relation When the state machine specification is not explicit, the state transition relation can be derived from the program specification. In general, a state transition $s_i \rightarrow s_j$ is triggered by some predicate P_{ij} on the internal state being true. Such a predicate can be separated into (1) a triggering *event* on the input(s), such as a clock tick (e.g., for synchronous clocked systems), or signal transitions (e.g., for asynchronous systems) and (2) an associated predicate on the variable values.

In VHDL for instance, the conjunction of predicates along a path from one `wait` statement to another, comprises the predicate corresponding to a state transition. (Such a correspondence is also useful in a tool for correlating error traces arising out of an execution of the explicit state model with the programmatic representation that only has implicit states.)

Iteratively computing M/R.

Given an equivalence relation R, it is easy to compute the reduced transition system M/R whose states are the partitions induced by R. The initial partition of S is derived from the HDL description. If B, C are blocks of states, then (in the reduced transition system) there is a transition from B to C labelled by an input action $a \in I$ iff each state in block B has an a-arc to some state in block C. If this condition is satisfied, then this arc labelled a from B to C is said to be *stable*, otherwise it is said to be *unstable*. If an arc is unstable, then the block B can be split into two blocks B' and B'' such that B' contains all of the states that have $a-$transitions leading to a state in C, while the states in B'' have $a-$transitions that do not lead to C. This procedure can be iterated until no further splitting of the blocks is necessary i.e., until no new blocks are being formed and all arcs are stable.

The sets of states can be compactly represented by using BDDs for their characteristic functions. The manipulations needed for the minimization task are: representation of the intersection of two blocks; the inverse of a block B; set difference, and test for emptiness.

An algorithm for computing the reachable part of a minimized system was proposed in [12]. The algorithm arrives at this reachable minimal graph in $O(NM)$ operations, where N is the number of nodes (state partitions), and M is the number of edges in the graph. However, in the absence of any other information, the algorithm may not know that it has found the final graph, and may continue working. In general, therefore, it is necessary to verify that a given graph is the desired one. In certain special cases, it is possible to efficiently answer this question. This method can be applied to extended finite state machines with separable affine transformations, i.e., machines with variables that are modified via separate affine transformations.

An affine transformation f from D^n to D^n is of the form $y_i = f_i(x) = \sum a_{i,j}x_j + b_i$. The transformation is separable if there is at most one variable on the RHS of each equation. It is strongly separable if the $i-$th variable maps to the $i-$th variable i.e., $y_i = \sum a_i x_i + b_i$, otherwise it is weakly separable. In the case where the coefficients are integers, the termination problems can be solved in linear time. In the case of arbitrary real coefficients, the termination problem of verifying the result graph can be reduced to a Linear Programming problem with two variables per inequality, which admits a strong polynomial solution. These results can be applied in the context of the hardware/system verification tasks of interest here.

4.3 Some useful examples of R (abstractions)

We now list some useful examples of the equivalence relation R. The first example captures the notion of the separation of control and data in an HDL description. The second example is aimed at exploiting the existence of symmetry in a system.

4.3.1 Equivalences induced by data path abstraction

Consider the case where the variables comprising the state are conceptually partitioned into a "symbolic state" SS of an underlying finite state machine, and the remaining components of the state are variable values over a domain D. (The overall state space is $SS \times D$ as discussed earlier.) The initial partition of the state space is induced by the homomorphism h defined as follows: $\forall s \in SS\ \forall d_i, d_j \in D\ h(< s, d_i >) = h(< s, d_j >)$ This is tantamount to *initially* merging all states that have the same "symbolic state" value into one equivalence class, independent of the remaining variable values.

The iterative refinement performed by the stabilization procedure will refine the initial partitions. At each step in the refinement, if a transition $[s_i] \rightarrow [s_j]$ depends on a predicate P_{ij} on D, where P_{ij} is not true on all the elements in D, this causes $[s_i]$ to be split into parts $[s_i, P_{ij}(D)]$ and $[s_i, \neg P_{ij}(D)]$ which correspond to a partitioning of the domain D by P_{ij}. More generally, we require that $\nexists d \in D(P_{ij}(d) \wedge P_{ik}(d))$. This notion can be

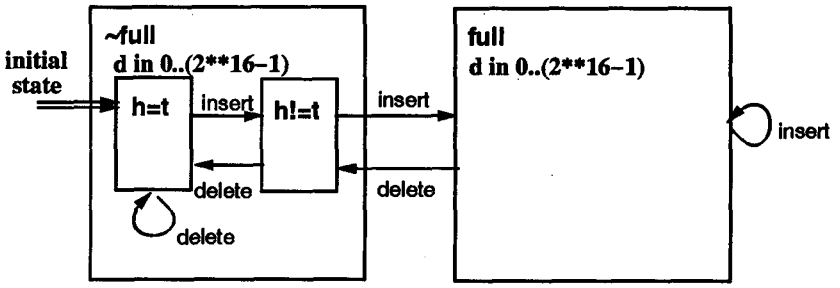

Figure 5: Refined transition structure for a FIFO of capacity 2

generalized to the case where the underlying types are user defined, in which case their semantics can be captured by either the initial or final algebra defined on the underlying Herbrand universe. The initial algebra captures the notion of congruence defined by a set of equations on the terms in the Herbrand universe, while the final algebra captures the notion of externally observable behavior.

As an example, in the case of a FIFO description, the refinement procedure does not cause the data values to be expanded. However, the pointer values used for the head and tail, and the status bit indicating whether the FIFO is full or empty are indeed expanded. The equivalence classes resulting from a refinement of the a FIFO of size 2 are shown in Figure 5. Note that the states $< \texttt{full}, \texttt{h} \neq \texttt{t} >$ are unreachable. This partitioning exploits the structure on the domains of `h` and `t` indicated earlier, viz., they are integers modulo 8, with initial values of 0, and updated only in increments of 4.

4.4 Equivalences induced by the presence of symmetry

On occasion, a system (or part thereof) may embody some form of symmetry. Examples include register files, memory and cache structures, regular iterative structures (e.g., systolic arrays), buses, etc. The presence of symmetry in a system implies that some sets of states or structural configurations are equivalent for the purpose of analyzing some of the system properties. Such equivalences can potentially lead to a reduction in the complexity of the associated analysis.[2]

Note that a system is typically associated with both a structure and a behavior. Symmetry can correspondingly be described at, these two levels. At the structural level, symmetry may be described by stating that the circuit behavior is invariant under certain permutations of the components. The state space of the transition system characterizing the behavior of the circuit is related to the structure. Alternatively, symmetry may be described by permutations on the state space, which leave the transition relation invariant.

[2]The notion of exploiting symmetry by explicit specification in an HDL is also independently being explored by Dill et al. and Clarke et al. in the context of model checking. Unfortunately, at the time of writing this draft we are unaware of enough details to make a detailed comparison.

```
-----------------------------------------------------------
SIZE = 8; /* initialize FIFO size */
/* initialize head (h), tail (t), and fullness status */
h=0; t=0; full=0;
/* main loop */
case (op) {
  insert:
   if (!full) {
     FIFO[h] = d; /*read in data from bus */
     h = (h+4) mod SIZE ; /* a datum takes 4 FIFO slots */
     if (h==t) full=1; /* update status */
   }
  delete:
    if ((h!=t) | (full)) {
        t = t+4 mod SIZE;
        full = 0; /* FIFO no longer full */
    }
}
-----------------------------------------------------------
```

Figure 6: Skeletal FIFO description (cf. Figure 5).

A permutation σ is a 1-1 onto map $S \rightarrow S$. (Note that a set of permutations $\Sigma = \{\sigma_i\}$ that is closed under composition forms a group.) Consider now the equivalence relation R on S defined by $< s_i, s_j > \in R$ iff $\sigma(s_i) = s_j$ for some permutation $\sigma \in \Sigma$. If the R is a congruence with respect to the transition relation T, i.e., the permutations in Σ do not change the transition structure, then the quotient algebra M/R is well defined. Formally, the requirement that R be a congruence with respect to T states that $(\forall s, s' \in S)(\forall \sigma \in \Sigma)((s, s') \in R \; iff \; (\sigma(s), \sigma(s')) \in R)$.

For example, let D represents the data domain for the values on a bus. An abstraction at the level of an HDL specification stating that $h : D \rightarrow d$, where d is a distinguished representative of D, is a way of stating that the actual values on the bus are indistinguishable for the behavior under consideration.

This abstraction at the HDL level is tantamount to stating that the wires in the bus representing D can be arbitrarily permuted without affecting the behavior being examined. While this may have been intended, a wiring interconnection error may have indeed rendered this untenable. It is therefore necessary to verify that the relevant group of permutations leaves the behavior of the circuit invariant. (Formally, a behavior under investigation can be expressed in terms of an appropriate set of operations (functions) defined on a universal algebra[9]. If an equivalence relation R induced by the generators of the permutations is a congruence with respect to the generators of the algebra, then the quotient M/R is well defined.) Unfortunately, the BDD representation of the equivalence classes induced by symmetries as defined above can still be quite large, and the computational complexity of

this problem is still quite hard.

Thus, while such abstractions are easy to make at the level of an HDL specification, they are not always computationally easy to verify at the level of the circuit. The way we currently address this complexity is by combining inductive reasoning based on a representation of the iterated structure of the circuit. However, this area needs further research.

5 Conclusions

We have described a divide-and-conquer paradigm to scale up the size of systems that can be verified. We are experimenting with the viability of this paradigm in the context of a design methodology that is targeted at the design of custom, high-performance, high-throughput digital signal processing circuits and systems. While the focus on this class of circuits does not restrict the applicability of the concepts, it highlights the challenge of dealing with bottom-up design methodologies in the context of verification, and the necessity of coping with the data path problem. Although we have had some initial success in our experiments, a significant amount of work is needed before the techniques described here can be considered mature. Further, while functionality is obviously of primary importance, issues related to the timing of such circuits and systems increasingly become the bottleneck in the design cycle. Ongoing work is directed at alleviating these bottlenecks.

Acknowledgements. I wish to acknowledge Sriram Narayanan for his ungrudging enthusiasm in discussions relating to the video-codec models.

References

[1] R. E. Bryant. Graph-Based Algorithms for Boolean Function Manipulation. In *IEEE Transactions on Computers*, volume C-35, pages 677--691, August 1986.

[2] R. E. Bryant. Formal Verification of Memory Circuits by switch-level simulation. In *IEEE Transactions on Computer-Aided Design*, 10(1), pages 94-102, January 1991.

[3] R. E. Bryant, D. L. Beatty, and C.H.Seger. Formal Hardware Verification by symbolic ternary evaluation. In *Proc. ACM/IEEE 23rd Design Automation Conference*, pages 397-402, June 1991.

[4] J. Burch, E. Clarke, K. McMillan, and D. Dill. Sequential Circuit Verification Using Symbolic Model Checking. In *Proceedings of the 27th Design Automation Conference*, pages 46--51, June 1990.

[5] L. Claesen, F. Proesmans, E. Verlind, and H. De Man. SFG-Tracing: a Methodology for the Automatic Verification of MOS Transistor Level Implementations from High Level Behavioral Specifications. In *Proceedings of International Workshop on Formal Methods in VLSI Designs*, January 1991.

[6] O. Coudert, C. Berthet, and J. C. Madre. Verification of Sequential Machines Using Boolean Functional Vectors. In *IMEC-IFIP International Workshop on Applied Formal Methods for Correct VLSI Design*, pages 111--128, November 1989.

[7] S. Devadas, K. Keutzer, A.S. Krishnakumar. Design verification and reachability analysis using algebraic manipulations. In *Proc. ICCD*, pages 250-258, October 1991.

[8] M. Gordon. HOL: A Proof Generating System for Higher Order Logic. In G. Birtwistle and P. A. Subrahmanyam, editors, *VLSI Specification, Synthesis and Verification.* Kluwer Academic Publishers, 1988.

[9] G. Gratzer. *Universal Algebra.* Second Edition, Springer Verlag, 1979.

[10] R. Kurshan. Analysis of Discrete Event Coordination *Springer Verlag LNCS*, 1990.

[11] VHDL. IEEE Standard 1076. VHDL Language Reference Manual. IEEE Press.

[12] D. Lee and M. Yannakakis. Online Minimization of Transition Systems. Personal Communication. (Also, preprint, Proc. Symp. Theory of Computing, 1992.)

[13] T. Kam and P. A. Subrahmanyam. Comparing Layouts with HDL Models: A Formal Verification Technique. In *Proceedings of International Conference on Computer Design*, pages 588--591, October 1992.

[14] S. Rao, M. Hatamian, and B. D. Ackland. A Design Environment for High Performance VLSI Signal Processing Circuits. In *Proc. ICCD*, pp. 147-152, October 1990.