Automatic Datapath Abstraction In Hardware Systems

Ramin Hojati, Robert K. Brayton

Abstract

The biggest stumbling block to make formal verification widely acceptable is the state space explosion problem. Abstraction is used to simplify a design so that the number of reachable states is reduced. In this paper, we first introduce a concurrency model, called integer combinational/sequential (ICS), capable of describing hardware systems at high and low levels of abstractions. ICS uses finite relations, interpreted and uninterpreted integer functions and predicates, interpreted memory functions, and supports non-determinism and fairness constraints. As a subset, it includes finite-state systems with general fairness constraints. Verification in this framework is performed using language containment as follows.

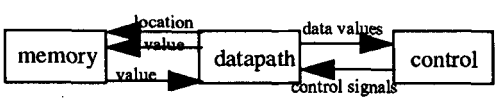
1. For a subclass of "control-intensive" ICS models, we prove that finite small instantiations can be used to decide the properties without sacrificing accuracy. A linear time algorithm for recognizing these subsets is given. These results also hold for the standard finite-state systems and thus also provide some generic methods for automatic data abstraction for such systems. Using these results, we are able to verify a memory model by reducing integer data values to binary, and unbounded memory addresses to a small number .

2. For verifying properties of circuits with complex datapaths, the model can be executed symbolically to find the reachable states. In some cases, the set of reachable states is finite, and the verification can be completed exactly. In other cases, given n, the verifier checks that no errors of length less than n exist.

1 Introduction

Formal design verification (or verification) is the process of validating a design by proving properties of it. Validation of a design is currently done using simulation, which is the bottleneck in many designs. Verification has the promise of reducing the simulation time, as well as increasing the level of confidence in the design.

A hardware system can be divided into three major components: control, datapath, and memory (see Figure 1). The ***control*** part consists of a set of interacting FSM's which, depending on data values and their internal states, produce a set of control signals for the datapath. The ***datapath*** consists of functions, predicates, and registers, which based on control signals, operate on data. The data often consists of integers. The ***memory*** acts as a container for values, and communicates with the datapath.



A hardware system consists of three major components: control, datapath, and memory. Figure 1

Properties can be classified as control, data, and data/control. ***Control properties*** are those which involve only the control signals. An example is that no two control signals of a bus are asserted at the same time. Most existing automatic verification techniques are suitable only for verifying such properties. Therefore, a verification expert needs to manually abstract the datapath. This process is time-consuming and often involves a third-party's understanding of the design.

Data properties are those which the datapath must satisfy. For example, for a pipelined datapath, one verifies that values appear on time where they are needed. The most successful technique to attack such properties has been theorem-proving. The data properties, addressed so far, fall into those for which automatic or almost automatic theorem-proving is possible. The theorem-prover PVS ([PVS93J) has been used to prove such properties almost

automatically (the proofs involve a few routine proof commands).

Data/Control properties involve both control signals and datapath variables. An example is "if an add command is issued, and no exceptions occur, then in 3 time steps, location c contains $a + b$." In general, such properties are the most difficult to verify, and only ad hoc techniques have been used to verify them.

Abstraction is the process of reducing the proof of a property on an infinite or large state space (***concrete model***) to a proof on a smaller state space (***abstract model***). Based on how well the abstract model approximates the concrete one, abstractions can be divided into three categories. ***Exact abstractions*** are those, where the property holds in the concrete model iff it holds in the abstract one. [MPS92] provides an example of exact abstractions is provided, where n -state counters whose data values are not used by other FSMs are reduced to 3-state machines. ***Conservative abstractions*** are those where if the property holds in the abstract model it holds in the concrete model. The homomorphism theory of [Kur92] is an example of (manual) conservative approximations. ***Aggressive abstractions*** are those where if the property does not hold in the abstract model then it does not hold in the concrete model. If a conservative abstraction fails, an aggressive abstraction may confirm the property does not hold. Similarly, if an aggressive abstraction holds, conservative abstractions may be used to possibly confirm the property holds.

Abstraction of hardware systems can be classified by their level of granularity. ***Datapath abstractions*** are those which eliminate portions of the datapath or reduce the number of bits in the datapath. ***Control abstractions*** are those which reduce the number of output values or states of a state machine based on equivalence with respect to a property. For example, some property may only be sensitive to two values of a multi-valued variable. In such cases, other values may be collapsed together. Control properties may not be behavior-preserving. For example, [HKB94] presents a technique for reducing the states of a FSM which does not preserve the behavior: if the original machine can produce an output in n steps, then the reduced machine can also produce it in n steps.

In this paper, we are concerned with datapath abstractions. We first introduce the ***integer combinational/sequential (ICS)*** concurrency model, which can describe hardware systems in a high-level of abstraction. This concurrecny model is based on the combinational/sequential (C/S) semantics ([HB95]), and extends it by 1) introducing integer variables, 2) interpreted and uninterpreted predicates and functions on integers, and 3) interpreted memory functions. ICS easily supports non-determinism and fairness constraints, and if the extensions are not used, it reduces to C/S. One can extend the general theory of verification using language containment ([Kur92]) to ICS models. Again if the ICS extensions are not used, the theory reduces to the theory of language containment on C/S models.

Verification using ICS models is performed in two ways. When verifying properties of control-intensive systems, i.e. those with simple datapaths, small instantiations can be used. In cases where the property cannot be decided using finite instantiations, the finite sized models can be used to either find an error, or give strong indications that the property holds. Using these results, we were able to verify a memory model by reducing integer data values to binary, and integer memory addresses to a small number ([HMLB95]).

The first kind of circuits we consider are ***data insensitive controllers***. Intuitively, these only move data around, and are not sensitive to the value of the data. It appears that many communication protocols fall into this category. We prove that for verifying certain types of safety properties, a single bit of data for each variable is sufficient. ***Data sensitive controllers***, on the other hand, interrogate data values, i.e. apply predicates to them, as well as move them around. A typical memory controller with respect to operations on memory addresses is an

example. We show that, depending on the predicates applied (we allow comparisons, equality, sign, and mod), a few bits can suffice to check the property. We prove a negative result which states when finite instantiations cannot be used. Since we use ICS models and the language containment paradigm, all our results unless otherwise stated, apply to liveness as well as safety properties. Algorithms for automatically recognizing data sensitive and insensitive controllers are given.

We also use ICS models and language containment to verify data and control/data properties. Fairness constraints are added to the ICS model, the property is specified, and its complement is computed. The set of reachable states of the composition of the model and the complement of the property is then computed, and it is verified that no fair path (a path satisfying all fairness constraints) exists. ICS is designed so that checking the equivalence of two states is decidable. However, the set of reachable states may be infinite, in which case language containment is undecidable. Language containment can then be approximated in the sense that it can be checked that no errors of a given length n exist. If all reachable states are computed within n steps, this verification is exact. We present a BDD-based algorithm for symbolic verification of ICS models.

Two previous works are most relevant to our results regarding finite instantiations. [Wol86] introduced the notion of data independent protocols, which are basically the same as data-insensitive controllers, and showed how finite instantiations can be used to decide properties expressed in a variant of linear temporal logic. [ID93] proved a result, called data saturation, which is similar to our Lemma 4.1, but does not handle liveness. Regarding symbolic execution of ICS models, [CN94] introduced a temporal logic, augmented with uninterpreted functions and predicates to express properties of pipeline circuits. [BD94] devised a decision procedure for deciding whether two terms built from uninterpreted functions and memory operations are equal. These formulas arise in the context of verifying pipelined circuits. We will give a more thorough comparison of our work with the previous ones in the later sections.

The flow of the paper is as follows. Section 2 presents the syntax of ICS, its operational semantics, its derivation from descriptions in hardware description languages (HDL's), how fairness constraints are specified, and how language containment is done. Data insensitive and sensitive controllers, theorems about their exact abstractions, and algorithms for recognizing them are described in sections 3 and 4. Section 5 gives a BDD-based algorithm for the symbolic execution of ICS models. Section 6 concludes the paper. Due to lack of space, most proofs have been eliminated.

2 Integer Combinatorial/Sequential (ICS) Concurrency Model

In this section, the integer combinational/sequential concurrency model is described. ICS is designed to represent systems composed of control, datapath, and memory. Some machinery is given to reason about integers, and how they affect the state spaces.

2.1 Syntax

The primitives are: variables, tables, interpreted functions and predicates, uninterpreted functions and predicates, constant creators, latches, and memory functions.

Variables. Variables are of two types: ***finite*** (or ***enumerated)*** and ***integer.*** Enumerated variables take values from some finite domain, whereas integer variables take integer values $(0, 1, 2, \ldots)$.

Tables. A table is a relation defined over a set of finite variables, divided into inputs and outputs. A table is a ***function*** if for every possible input tuple there is at most one output tuple. Otherwise it is a ***relation.*** If a table has only one binary output, and is a function, then it is a ***predicate.***

Interpreted Functions and Predicates. A set of functions and relations over integers has

been pre-defined. The ***interpreted functions*** are: $z := x + y$, $y := x$, $y := x + c$, $y := if(b, x)$, $z := mux(b, x, y)$, where x, y, and z are integer variables, b a binary variable, and c an integer. The ***interpreted predicates*** are $y = x$ (equality), $y < x$, $x = c$, $(x \bmod m) = r$, and $(x \bmod m) < r$.

Uninterpreted Function and Predicates. These are a set of function and predicate symbols with their arities and domain variables given. For example, if x_1 and x_2 are variables, $f(x_1, x_2)$ may be the specification of an uninterpreted binary function defined over binary variables x_1 and x_2.

Constant Creators. A constant creator is a special element with no input, and an integer output. Intuitively, it is a higher-order function, which creates a new constant each time called, i.e. a fresh 0-ary uninterpreted function.

Latches. A latch is defined on two variables over the same domain: input (or ***next state***) and output (or ***present state***) of the latch. Present and next state variables may overlap, e.g. when an output of a latch is an input to another. Every latch has a set of initial values, which are a subset of the domain of its variables. If the latch is integer-valued, then the initial value set can either be a finite set of integers, a given constant, or the output of a constant creator. Predicates can be used to create infinite initial values. For example, the predicate $x > 5$ can be used to only allow integers greater than 5.

Memory Functions. Two functions ***read*** and ***write*** are provided with their usual interpretation; *read* is a unary function whose argument is a location; *write* is a binary function, whose arguments are a location and a value. Location and value are variables in the model. It is implied that *read* and *write* have an extra argument which is the memory itself[1]. Reading a location which has not been written, returns a new constant (like a constant creator).

Definition A ***generalized gate*** is a table, an interpreted or uninterpreted function or predicate, or a constant creator.

Every model has only a finite number of variables, latches, and generalized gates. Every variable is the output of exactly one generalized gate or latch. Hence, every input to a generalized gate or latch is the output of some other generalized gate or latch, i.e. our systems are closed. A variable can be input to many generalized gates or latches.

Definition An ***ICS term*** is built recursively from constants, interpreted and uninterpreted functions. Therefore, constants are ICS terms, and if f is an n-ary function and $t_1, \ldots, t_n$ are ICS terms, then $f(t_1, \ldots, t_n)$ is also an ICS term. Note that ICS terms do not involve any variables or predicates.

Definition A ***state*** is a triple $(latch, memory, predicates)$, where,

a. *latch* is an assignment of values to the latches. For finite valued latches, the value comes from the domain. For integer valued latches, the value is an ICS term.

b. *memory* is a set of pairs of ICS terms, the first denoting a location and the second a value.

c. *predicates* is a set of atomic formulas (i.e. interpreted and uninterpreted predicates applied to ICS terms).

---

1. One can easily extend ICS models to allow for more than one memory, in which case *read* and *write* take an extra argument, which specifies which memory the operation is on.

Definition Given two ICS terms t_1 and t_2, and two sets of atomic formulas $P = \{P_1, ..., P_n\}$ and $Q = \{Q_1, ..., Q_m\}$, t_1 ***is equal to*** t_2 ***subject to*** P ***and*** Q, denoted as $t_1|_P = t_2|_Q$ iff the formula $t_1 \wedge P_1 \wedge ... \wedge P_n = t_2 \wedge Q_1 \wedge ... \wedge Q_m$ is valid. For example, if $t_1 = x$, $P = \{x > 7, x \leq 8\}$, $t_2 = 8$, and $Q = \emptyset$, then $t_1 = t_2$ subject to P and Q since $x \wedge \{x > 7, x \leq 8\} = 8$ is valid.

Definition Let states $s_1 = (L_1, M_1, P_1)$ and $s_2 = (L_2, M_2, P_2)$ be given. $s_1 = s_2$ if the following both hold.

a) Let l_1^i and l_2^i denote the values of the i-th latch in L_1 and L_2, respectively. If the i-th latch is finite, then $l_1^i = l_2^i$; otherwise, $l_1^i|_{P_1} = l_2^i|_{P_2}$ must hold.

b) Let $m_k^i = \left(a_k^i, v_k^i\right)$ denote the i-th address/value pair in M_k. Then, for each m_1^i there exists m_2^j such that $a_1^i|_{P_1} = a_2^j|_{P_2}$ and $v_1^i|_{P_1} = v_2^j|_{P_2}$. Similarly, for each m_{26}^j there must exist m_1^i such that $a_1^i|_{P_1} = a_2^j|_{P_2}$ and $v_1^i|_{P_1} = v_2^j|_{P_2}$.

Definition An ***initial state*** is a state $(latch_{init}, \emptyset, \emptyset)$, where $latch_{init}$ is an assignment of an initial value to each latch. The constants assigned to those latches whose initial value is constant must be distinct.

Lemma 2.1 It is decidable whether two states are equal.

Proof (sketch) The problem reduces to deciding whether two ICS terms are equal subject to predicate constraints. ICS terms are combinations of ground Presburger arithmetic and uninterpreted functions and predicates. [Sho79] shows how such formulas can be decided (QED).

2.2 Operational Semantics of ICS

The operational semantics of ICS describes how a transition between two states of an ICS model occurs. Since the development here parallels the one in [HB95], most proofs are omitted.

Definition A ***gate graph*** G is a directed graph where every node is a generalized gate. $(u, v) \in G$ if some output variable of generalized gate u is an input to generalized gate v. A cyclic gate graph is said to contain a ***combinational loop (or cycle).***

Remark For an acyclic gate graph G a root node either has no inputs or each input is a latch output.

Our operational semantics is restricted to acyclic graphs and is defined in terms of a ***configuration*** (or ***state) graph*** and its transition relation. Every node in the configuration graph is a pair (s, n), where s is a state of the model, and n represents the number of constants created so far. An initial state of the configuration graph is of the form (s_{init}, k), where s_{init} is an initial state of the model, and k the number of constants in s_{init}.

We present an algorithm which, given a state $u = ((L, M, P), n)$ in the configuration

graph, assigns a new value to all next state variables (creating L'), and creates a new memory M', a set of predicates P', and a counter n'. In this case, we say there is an edge (u, v) in the configuration graph, where $v = ((L', M', P'), n')$. We show the configuration graph is well-defined by proving our algorithm is well-defined.

Notation If $P = \{P_1, ..., P_n\}$ is a set of predicates, we will use P in atomic formulas to denote $P_1 \wedge ... \wedge P_n$. For example, $P \rightarrow (b = 1)$ is the atomic formula $(P_1 \wedge ... \wedge P_n) \rightarrow (b = 1)$.

0. Let $P' = P$, $n' = n$.

1. Choose a topological sort of the gate graph.

2. Assign to the outputs of the latches the values given by L.

3. Assign values to the outputs of each generalized gate consistent with its inputs, processing the generalized gates in the topological order. More precisely, let a generalized gate $g(i, o)$ be given, where i represents the inputs to the gate and o its output.

a. If g is a table representing the relation $R(i, o)$, then $(i, o) \in R$.

b. If g is an interpreted or uninterpreted function, then $o = g(i)$, where o and i are ICS terms.

c. If g is an interpreted or uninterpreted predicate, if $P' \rightarrow (g = 0)$ is valid, let $g = 0$; if $P' \rightarrow (g = 1)$ is valid, let $g = 1$; otherwise let $o = 0$ or $o = 1$, and $P' = P' \cup \{g = o\}$.

d. If g is a constant creator, then $o = c_{n'}$, where $c_{n'}$ is a fresh constant, and $n' = n' + 1$.

Step 3 is referred to as ***value propagation***. Note that the configuration graph is finite-branching, i.e. for every state, there are a finite number of next states. We prove the algorithm is well-defined by proving it is not sensitive to the topological sort chosen. It is possible that a table is not complete, i.e. there are inputs for which there are no outputs. Then, the set of values assigned to an output of a table may be empty. The empty values propagate, i.e. if one of the inputs to a table is empty, then the output is empty as well.

Lemma 2.2 When a generalized gate is processed in step 3 each of its inputs has been assigned a value.

Lemma 2.3 The above procedure assigns values to all next state variables (inputs of the latches).

Lemma 2.4 Let L be an assignment of values to all latch outputs. Let L' and n' be obtained by value propagation given topological sort O_1 and value n. Then, given topological sort O_2, L' and n' can be obtained by value propagation from L and n.

2.3 Creating ICS Models From the HDL Verilog

In the HSIS environment, Verilog descriptions are compiled into BLIF-MV, an intermediate format which implements combinational/sequential concurrency model. BLIF-MV has a library of pre-defined functions and predicates on finite-sized integers. For example, $add4(x, y, z)$ may mean that two 4-bit integers x and y are added to get the 5-bit integer z. We can easily extend this to compile a Verilog description into an ICS model.

2.4 Fairness Constraints and Language Containment

In this section, we describe how to construct an automaton from an ICS description, how to define fairness constraints on this automaton, what its language is, how to specify properties, and how language containment is performed by checking language emptiness.

2.4.1 Fairness Constraints in ICS Models

Definition Let an acyclic ICS model M be given. The ***symbolic automaton*** of M, A_M, is the configuration graph defined by the operational semantics of ICS. The ***alphabet*** of A_M is the Cartesian product of the alphabets of the non-state variable, and is denoted by Σ_M. The alphabet of the integer variables is the set of the ICS terms of M (a countable set), whereas the alphabet of the finite variables is their domains. Let an ω-string x over the alphabet Σ_M be given. r is a run of x in A_M if r_0 is an initial state of A_M, and for all i, there is a transition from r_i to r_{i+1} assigning x_i to non-state variables.

If the model has inputs, we close it by allowing the finite inputs to take any value in their domains, while the integer inputs are driven by constant creators. Hence, the notion of symbolic automaton carries to models with inputs. We allow edge-Streett ([HB95]) fairness constraints[1] to be placed only on finite valued latches. For example, one can restrict the acceptable runs to be such that a finite-valued latch l takes some value a in its domain infinitely often. Fairness constraints are generally placed on the control part. Since control circuits are finite state, the restriction of fairness constraints to finite latches is expected not to be severe in practice.

Definition A run r is fair if all fairness constraints are satisfied. More specifically, for each finite latch l, the set of infinitely occurring values which l takes in the run r satisfies all fairness constraints placed on l. The ***symbolic language*** (or just ***language***) of symbolic automaton M, $L_S(M)$, is the set of all strings which have a fair run in A_M.

Definition The ***concrete language*** of a symbolic language L with respect to an interpretation I (an interpretation of all uninterpreted functions and predicates), denoted by $L_C(L, I)$, is obtained by allowing the constants to take any possible integer value for all $x \in L$. The concrete language of M with respect to interpretation I is $L_C(L_S(M), I)$.

Definition Let symbolic languages L and L' be given. L is ***contained*** in L', denoted as $L \subseteq L'$, if for every interpretation I, $L_C(L, I) \subseteq L_C(L', I)$.

Definition Let symbolic languages L, L', L'' be given. L'' is the ***intersection*** of L and L', denoted as $L'' = L \cap L'$, if for every interpretation I, $L_C(L'', I) = L_C(L, I) \cap L_C(L', I)$. Similarly ***union*** and ***complementation*** of symbolic languages are defined.

Definition Let models M and N be given. The ***composition*** $M \bullet N$ is defined as follows. If M and N have no variables in common, or the common variables are not outputs in both

---

1. One can allow other types of fairness constraints as well. The language emptiness check for edge-Streett fairness constraints is polynomial, while their next natural extension has an NP-complete languages containment check.

models, the composition of the two models is their syntactic composition. For each common variable which is an output in both M and N, rename one of them, and add a one state automaton which checks that the two outputs are equal.

Lemma 2.5 Let models M and N be given. Then, $L(M \bullet N) = L(M) \cap L(N)$, i.e. composition of models corresponds to taking the intersection of their languages.

2.4.2 Property Checking

Lemma 2.5 implies that the classical results of verification using language containment are valid. For example, to do property checking, the complement automaton of the property can be composed with the system, and the language of the composed system can be checked for emptiness. For hierarchical verification, if the language of the refined model is contained in the language of the abstract model, the properties proved on the abstract model are guaranteed to hold for the refined model.

Definition A ***property automaton*** is a complete (i.e. for every symbol there is a transition) edge-Rabin ([HB95]) automaton with no outputs. Note that property automata can have integer inputs. All latches of a property automaton are finite valued (i.e. a property automaton has a finite number of states), and the only allowed integer operation are integer predicates which are applied to integer inputs, but the alphabet of a property automaton can take an infinite number due to integer inputs.

Definition Let A be a property automaton. The ***finite encapsulation*** of A, A_F, is obtained by replacing each integer predicate (applied to integer inputs) with a binary input variable. Note that A_F is a regular edge-Rabin automaton with finite alphabet and finite number of states.

Lemma 2.6 Property automata can be complemented.

2.4.3 Hierarchical Verification

Besides property checking, language containment appears in hierarchical verification. Hierarchical verification is the process of checking that the language of an abstract model contains the language of a detailed model. Since we don't know how to complement general symbolic automata, we restrict hierarchical verification to a subset of symbolic automata, called control automata. Control automata are where non-determinism (and hence refinement) occurs; hence, this may not be too restrictive in practice.

Definition A ***control automaton*** is a property automaton which is allowed to have only finite outputs. The finite encapsulation of a control automaton can be defined similar to the case of property automaton.

Lemma 2.7 Control automata can be complemented.

2.4.4 Checking Emptiness of ICS Models

Verification using language containment involves complementing the property automaton, and checking whether there is a fair path in the composition of the system and the complement of the property. In order to do this, we need to compute the set of reachable states, and check whether there are any fair paths. However, the set of reachable states may be infinite (checking language emptiness of ICS models with fairness constraints is undecidable). Hence, we compute the set of states reachable in at most n steps, for some given n. We then check that no fair path in this subset of the reachable states exists. In practice, since many error traces are short (if they exist), this technique should be quite effective.

Lemma 2.8 The language emptiness check of ICS model is undecidable.

3 Data Insensitive Controllers

In this section, we formalize the notion of data insensitive controllers (DICs). We then prove

a theorem which can be used to verify the property that DICs correctly move data around on a system where the integer variables are replaced by single bit binary variables. A practical application of this result is then described, and an algorithm for automatically recognizing DICs is given.

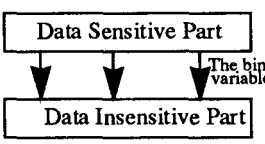
3.1 0-1 Theorem for Data Insensitive Controllers

Definition Let the ***data movement operations*** be $x := y$, $z := mux(b, x, y)$, and $y := if(b, x)$, where x, y, z are integer variables, and b is binary.

Definition Let an ICS model M be given. M is a ***data insensitive controller (DIC)*** with respect to a set of variables X, called ***data insensitive variables (DIV),*** if the following holds.

1. The only operations allowed on the variables in X are data movement operations and $x :=$ constant-creator where all variables involved in these operations belong to X.
2. If $l \in X$ is a state variable, then the initial value of l is a fresh constant.
3. If some constant creator C is input to some $x \in X$, then C is not input to any other variable $z \notin X$.

The (generalized) gates in a DIC are naturally divided into two disjoint parts. The first, called the ***data sensitive part***, contains the gates which drive the variables not in DIVs, whereas the second called the ***data insensitive part***, contains those gates which drive the DIVs. The binary variables which appear in the data insensitive part are driven by gates in the data sensitive part. No gate in the data sensitive part has a DIV as an input.



The data sensitive part generates the control signals for the data insensitive part. Changing the values of variables in the data insensitive part does not affect the values of the data sensitive part.

Figure 2

Definition Let M be a DIC with respect to X. The ***binary instantiation*** of M, M_2, is formed by replacing all variables in X by binary ones, replacing each constant creator driving a DIV by a gate which produces 0 or 1 non-deterministically, and replacing the initial values of all latches in X by the set $\{0, 1\}$.

0-1 Theorem for Data-Insensitive Controllers Let an ICS model M be data insensitive with respect to a set of variables X. Let property P specify that when binary variable b becomes 1, then $x = y$, where $\{x, y\} \subseteq X$. Then, P holds for M iff P holds for M_2.

Historical Remark [Wol86] introduced the notion of data-independent controllers for simple reactive programs, which are conceptually very similar to our data-insensitive controllers. However, the results of [Wol86], specifically theorem 5.4, are essentially different than our 0-1 theorem. [Wol86]'s theorem 5.4 basically applies to properties which can be written as "for all tuples $i_1, \ldots, i_n$ with distinct values, $P(i_1, \ldots, i_n)$ should hold", where P is a linear temporal logic formula over the variables $i_1, \ldots, i_n$. Such a property can also be expressed as an infinite conjunction of linear temporal logic formulas of the form $P(v_1, \ldots, v_n)$, where $v_1, \ldots, v_n$ are value assignments to $i_1, \ldots, i_n$. However, our 0-1 theorem is equivalent to an infinite disjunction, i.e. "when b occurs either $x = y = 0$, or $x = y = 1$, or $x = y = 2$, etc."

One in general may be interested in proving the property "repeatedly whenever binary variable b becomes 1, then $x = y$." To do so, create M' from M by adding a two-state

FSM, which outputs a variable b'. This FSM stays in its first state for an arbitrary amount of time, outputting $b' = 0$. It can non-deterministically make a transition to its second state, outputting $b' = 1$. It then stays in its second state forever outputting $b' = 0$. Let P' be the property "when binary variable b' becomes 1, then $x = y$." It is easy to see that P holds of M iff P' holds of M'. However, the 0-1 theorem applies to the latter case.

We also emphasize that this theorem also applies to finite state systems and can be used to abstract datapaths from a large number of bits to a single bit.

3.2 An Application of the 0-1 Theorem

Omitted due to lack of space. See [HMLB95].

3.3 Recognizing Data Insensitive Controllers

DICs can be automatically recognized with a linear time algorithm (in the size of the integer operations) which finds the largest set of variables according to which a model is DIC.

Lemma 3.1 If an ICS model M is a DIC with respect to X and a DIC with respect to Y, then it is a DIC with respect to $X \cup Y$. Hence, there is a largest set of variables $DIV(M)$ according to which M is a DIC.

The following algorithm finds $DIV(M)$. The algorithm proceeds by marking off any variable which cannot be a DIV. The remaining variables are in $DIV(M)$.

1. Mark each integer variable involved in any operations other than data movement, i.e. violating condition 1 of DICs.

2. Mark any variable whose input is a constant creator with more than one fanout, i.e. violating condition 3 of DICs.

3. Let X be the of marked variables in steps 1 and 2. For each $x \in X$, mark off any other variable which is involved in some integer operation (including data movement) with x. Continue the process with the set of newly marked variables.

4. Return all non-marked integer variables as $DIV(M)$.

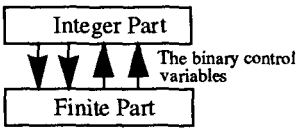
4 Data Semi-Sensitive Controllers

In this section, we present results about a class of circuits, called data semi-sensitive circuits (DSSC), which disallow function application to integer variables. We identify subclasses of these circuits, where properties can be proved by replacing the ranges of integer variables by a small finite range. Recognizing DSSCs and any of the subsets we present in this section is straight-forward. In what follows, checking a property on a system M is done by checking for the language emptiness of the composition of the complement of the property and M.

Definition The predicates $x < c$, $x = c$, $x < y$, $x = y$, $(x \bmod m) = d$, and $(x \bmod m) < d$ are called ***integer predicates.*** Note that the predicates $Even(x)$ and $Odd(x)$ are special cases of $(x \bmod m) = d$.

Definition An ICS model M is a ***data semi-sensitive controller (DSSC)*** if the only operations on integer variables are data movement, $x :=$ constant-creator, and integer predicates. The initial value of all integer latches is a fresh constant.

A DSSC can be pictured as shown in Figure 3, where the model is divided into two parts: integer and finite. The integer part contains the gates which drive the integer variables, whereas the finite part contains those which drive the finite variables. The communication between the two parts is restricted to a set of binary control variables. The control variables produced in the integer part are the outputs of the integer predicates. The gates in the finite part take only finite variables as inputs.



The communication between the two parts is restricted to a set of binary control variables, where the binary variables output from the integer parts are the output of the integer predicates. Figure 3

Definition Let M be a DSSC. The ***n-ary instantiation*** of M denoted by M_n, is formed by replacing all integers variables by finite variables taking values from $0, 1, \ldots, n-1$. The initial value of an integer is the set $0, 1, \ldots, n-1$.

4.1 Data Comparison Controllers

Data comparison controllers are a subclass of DSSCs which move data around, and compare them. We prove finite instantiations can be used to verify language emptiness for these circuits. A practical example of where data comparison controllers come up is given.

Definition A DSSC which only involves data movement operations, the predicate $x = y$, and constant creators is called a ***data comparison controller***.

Lemma 4.1 Let M be a data comparison controller with n integer variables, and M_n its n-ary instantiation. Then, $L(M) = \emptyset$ iff $L(M_n) = \emptyset$.

A result similar to Lemma 4.1 appeared in [ID93]. The main difference is that Lemma 4.1 applies to both liveness and safety properties. Also our bound is a bit better (the bound in [ID93] for safety properties was $n+2$). The fact that Lemma 4.1 applies to liveness properties comes basically for free as a consequence of dealing with ICS models. As an example, the property "if request and $x = y$, then eventually acknowledge and $z = w$" can be proved in our framework, but is not dealt with in [ID93].

Lemma 4.2 There is a data comparison controller M having n integer variables such that $L(M) \neq \emptyset$ and $L(M_i) = \emptyset$ for $i < n-2$, i.e. the bound given by Lemma 4.1 is not off a tight bound by more than 2.

One application of data comparison controllers is in memory systems of multi-processors. These memory controllers have a buffer for each processor where the memory instructions (load and store) are stored. To perform verification, we assume the sizes of the buffers and the number of processors are finite ([HMLB95]). However, it is assumed that the number of memory locations is infinite. If the memory controller only compares memory addresses to each other, which is what one might expect, then it is a data comparison controller with respect to the addresses. Hence, a small number of addresses suffice to prove the properties. This number is proportional to the sizes of the buffers. Combining this result with the 0-1 theorem for data insensitive controllers, we get that once the sizes of the instruction buffers are fixed, then a memory controller can be verified using binary data bits, and a finite (and small) number of memory locations without losing any accuracy in verification.

4.2 Other Types of Data Semi-Sensitive Controllers

We present results on circuits which use more sophisticated integer predicates (such as $x < c$, $(x \bmod m) = r$, and $x < y$). Our results show that the only situation where finite instantiations cannot be used is when both data movement and the predicate $x < y$ are used.

Lemma 4.3 There exists a DSSC controller M involving data movement operations and predicate $x < y$ such that $L(M) \neq \emptyset$ but $L(M_n) = \emptyset$ for all n.

Lemma 4.4 Let M be a DSSC with n integer variables, p predicates of the form $x = c_i$

for $0 \le i \le p-1$, and using only data movement operations, and the predicate $x = y$. Then, $L(M) = \varnothing$ iff $L\left(\hat{M}_{n+p}\right) = \varnothing$, where $\hat{M}_{n+p}$ is M_{n+p} with the predicates of the form $x = c_i$ changed to $x = i$.

Remark In Lemma 4.4, if there are no predicates of the form $x = y$, then $p+1$ values suffice. This is the core of [Wol86]'s techniques, where the predicates $x = c_i$ represent the atomic formulas of the temporal logic formula. Note that as [Wol86] argues, for such a circuit M (involving data movement and $x = c_i$'s), if emptiness holds for a set of distinct constants $c_1, \ldots, c_p$, then it holds for any circuit obtained from M by replacing $c_1, \ldots, c_p$ by another set of distinct constants $c_1, \ldots, c_p$.

Lemma 4.5 Let M be a DSSC with n integer variables, p predicates of the form $x = c_i$, and using only data movement operations, and the predicates $x = y$, $x = c_i$, and $x < d_j$. Let $C = MAX(MAX(c_i), MAX(d_j))$. Then, $L(M) = \varnothing$ iff $L\left(M_{\tilde{n}}\right) = \varnothing$, where $\tilde{n} = C + n + 1$.

Remark One can get a bound for Lemma 4.5 which is independent of the values of the constants. Let p and q predicates of the forms $x < d_j$ and $x = c_i$ respectively be given. Further assume $d_j \le d_{j+1}$. Let h_j be the number of c_i's which occur between d_j and d_{j+1}. Intuitively, between d_j and d_{j+1}, $n + h_j$ values are needed. By re-assigning value to $x < d_j$'s and $x = c_i$'s as in the proof of Lemma 4.4, the emptiness check can be done on a system where the integer variables take on $(p+1)(n+1)+q$ values (there are $p+1$ intervals between d_j's each requiring $n + h_j$ values, and $\sum h_j = q$).

Lemma 4.6 Let M be a DSSC with n integer variables, using only data movement operations, and the predicates $x = y$, $x = c_i$, $x < d_j$, $(x \bmod m_k) = r_k$, $\left(x \bmod m_{k'}\right) < r_{k'}$. Let $C = MAX(MAX(c_i), MAX(d_j))$, and N be the common multiple of all the m_k's and $m_{k'}$'s. Then, $L(M) = \varnothing$ iff $L\left(M_{\tilde{n}}\right) = \varnothing$, where $\tilde{n} = C + Nn + 1$.

The following lemma states that although the predicate $x < y$ cannot be used with data movement operations, but it can be used with all other predicates so long as there are no data movement operations.

Lemma 4.7 Let M be a DSSC with n integer variables using no data movement operations. Let m be,

a. n, if M involves only the predicates $x = y$ and $x < y$.

b. $C + n + 1$, if M involves the predicates $x = y$, $x < y$, $x = c_i$, and $x < d_j$, where $C = MAX(MAX(c_i), MAX(d_j))$.

c. $C + Nn + 1$, if M involves the predicates $x = y$, $x < y$, $x = c_i$, $x < d_j$,

$(x \bmod m_k) = r_k$, $\left(x \bmod m_{k'}\right) < r_{k'}$, where $C = MAX(MAX(c_i), MAX(d_j))$, and N is the least common multiple of all the m_k's and $m_{k'}$'s.

Then, $L(M) = \emptyset$ iff $L(M_m) = \emptyset$.

5 Verifying Properties Involving Data

We now describe techniques which can be used when finite instantiations cannot. An example is proving data or data/control properties of a system whose datapath is complex, e.g. it contains interpreted and uninterpreted functions. First, we describe a BDD-based algorithm for verifying ICS models. We then compare our work with the previous ones. Finally, we describe how a mixed verification/simulation strategy can be employed to find bugs quickly.

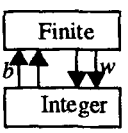
5.1 Symbolic Execution of ICS Models

The idea for symbolic execution of ICS models (which can be infinite) is to compute the set of states reachable in n steps, and check whether some infinite accepting path exists in this subset. An overview of a symbolic algorithm using BDDs for an acyclic ICS model M is given below.

1. *Compute the set of reachable states, or a subset of it reachable in n steps. Let this set be S, and represent it using BDDs*
2. *Let T_S be the transition relation of model M restricted to S. Represent T_S using BDDs.*
3. *Using techniques of [HB95], check whether there are any accepting runs in T_S.*

Step 3 of the above algorithm is well-defined; we explain steps 1 and 2 in detail.

Let a model M with finite latches $l_F = \left(l_1, \ldots, l_{f_i}\right)$, integer latches $l_I = \left(l_1, \ldots, l_{i_i}\right)$, and integer predicates $P_1, \ldots, P_p$ be given. Let l'_F and l'_I represent the set of next state variables corresponding to l_F and l_I. M can be partitioned into two sets, finite and integer. Let $b = (b_1, \ldots, b_p)$ be the binary variables corresponding to the outputs of $P_1, \ldots, P_p$. Let $w = (w_1, \ldots, w_q)$ be the set of finite variables which are inputs to the integer parts. Usually $w_1, \ldots, w_q$ are also binary.



An ICS model can be divided into two parts: finite and integer. Finite valued variables, in general binary, are used to communicate between the two parts.

Figure 4

To compute the set of reachable states, three auxiliary tables are used. The ***ICS table*** H_I is a table of all ICS terms which have been stored in latches in states visited so far. The ***predicate table*** H_P is a table of all integer predicates enumerated so far. The ***memory table*** H_M is a table of memories, where a memory is a table from ICS terms to ICS terms. One can encode a variable ranging over a finite domain, using a set of binary variables. We call all binary variables corresponding to a finite-valued variable a BDD variable. Let S_i be the set of states reachable in i steps, represented by a BDD, where there are BDD variables for all finite latches, BDD variables for integer latches ranging over all ICS formulas in H_I, one BDD variable for each predicate in H_P, and one BDD variable m ranging over all memories in

H_M. Note that the number of BDD variables as well as their ranges are finite.

Let p denote the set of binary variables corresponding to the predicates in H_P. Let T_i represent the transition relation of M restricted to S_i, represented as a BDD. Let T_F be the BDD of the transition relation of the finite part, which is obtained by taking the intersection of the BDDs of all finite tables. The following algorithm describes how S_{i+1} and T_{i+1} are computed. The algorithm maintains the invariance that in every state (l_F, l_I, p, m) the set of predicates in p imply that the addresses of the memory m are distinct.

1. Let $G(b, w, l_I, p, m) = \exists x (T_F \wedge S_i)$ *, where* x *is the set of all variables* T_F *depends on except for* b *and* w.

2. For each $(b, w, l_I, p, m) \in G(b, w, l_I, p, m)$,

2a. Given (w, l_I, p, m) *, propagate values through the integer partition, choosing the values given by* b *to integer predicates if possible. If not, stop the value propagation.*

2b. Add the new ICS terms assigned to next state variables to H_I*, and extend the range of integer latches to accommodate this.*

2c. Add the new integer predicates to H_P*, and introduce new binary variables for them. Set these variables to* 0 *for the states in* S_i.

2d. Let $\hat{T}_{i+1}\left(b, w, l_I, p, m, \tilde{l}_I, \tilde{p}, \tilde{m}\right) = \sum \{b \wedge w \wedge \left((l_I, p, m), \left(\tilde{l}_I, \tilde{p}, \tilde{m}\right)\right)\}$ *, where* $\left(\tilde{l}_I, \tilde{p}, \tilde{m}\right)$ *are the new values assigned to integer latches, predicates and memory as the result of propagating* (w, l_I, p, m) *. As explained below there may be several such choices due to memory operations.*

3. Let $T_{i+1} = T_i \vee \exists z\left(T_F \wedge S_i \wedge \hat{T}_{i+1}\right)$*, where* z *are all variables except for the state variables. Let* $S_{i+1} = \exists ns (T_{i+1})$ *, where* ns *are the set of next state variables.*

The details of step 2 of the above algorithm are as follows.

1. In step 2a, given a predicate Q and an assignment to it b_Q, the validity of $Q_1 \wedge \ldots \wedge Q_k \rightarrow (Q = Q_b)$ and $Q_1 \wedge \ldots \wedge Q_k \rightarrow \left(Q = \overline{Q_b}\right)$ are first checked, where $(Q_1, \ldots, Q_k)$ are all the predicates in H_P. If the former is valid, value propagation is continued. If the latter is valid, value propagation is stopped. If neither is valid, then Q is added to H_P, and a new binary variable is created for it which is set to Q_b.

2. If during value propagation, the memory operation $read(x)$ is encountered, find an address y in M_H, such that $Q_1 \wedge \ldots \wedge Q_k \rightarrow (x = y)$ is valid, and return the value pointed to by y. If no such address is found, for each address y in M_H, check whether $Q_1 \wedge \ldots \wedge Q_k \rightarrow (x \neq y)$ is valid. If this is not valid, introduce a new predicate $x = y$, and a new binary variable for this predicate. If this binary variable is true, then return the value pointed to by y. If all such binary variables are set to false, return a fresh new constant. This situation corresponds to reading a location which has not been written before.

3. If during value propagation, the memory operation $write(x, d)$ (write the value d to address x) is encountered, check for any address y such that $Q_1 \wedge \ldots \wedge Q_k \rightarrow (x = y)$ is valid. If such a y is found, change the value of y to d. If not, check the validity of $Q_1 \wedge \ldots \wedge Q_k \rightarrow (x \neq y)$ for each address y. If none is valid, for each address y, create a new predicate $x = y$, and a new binary variable corresponding to this variable. If this binary variable is set to true, change the value of y to d. If all such binary variables are false, add a new entry (x, d) in m. This situation corresponds to writing to a new address.

4. After value propagation, it is checked whether the created memory has been encountered before. If not, the range of m is extended.

The complexity of the algorithm is based on the complexity of the finite partition, the number of predicates in the datapath, the amount of communication from the finite partition into integer partition (the set w), and the number of distinct pairs of (l_I, m) at every point in time. If the datapath is not very complex, we expect the algorithm to be dominated by the complexity of the finite partition, which in general is efficiently represented using BDDs.

Remark If only integer functions and predicates are used, there are no constant creators, and the integer latches are initialized to a finite set of values, then the ICS terms obtained during value propagation are integers, and can be evaluated using the computer's arithmetic functions. This situation might for instance come up when handling a complete microprocessor. In this case, the above algorithm can be used to generate all states reachable within n steps. If there is no non-determinism, then the algorithm reduces to a simulation algorithm, which uses BDDs to represent the control part, and uses the computer's arithmetic functions to compute the values encountered in the datapath.

5.2 Comparison With Previous Work

Two previous works are most relevant.

1. *Functional Equality*. [BD94] introduced techniques for checking whether two functions corresponding to two implementation of a datapath, one pipelined and the other not, are equivalent. This kind of verification can be done using ICS models, and its verification is decidable, since the pipeline is executed only for a finite number of steps. Compared to the ICS approach, [BD94] is very specialized. It appears that any verification which cannot be posed as a function equality check cannot be performed in this framework; hence, general property checking cannot be done. Also, non-determinism and fairness constraints are not supported. As for efficiency, it is hard to know a priori, which method would be better for those problems where both are applicable.

2. *Extended Temporal Logic*. [CN94] introduced an extended temporal logic, called ground temporal logic, with pretty much the same expressiveness as ICS models. They suggested that transition diagrams be translated into this logic, and the validity of the formulas be checked. This approach has the same drawbacks as when validity of linear temporal logics is used as the computational means for verification; there are no known efficient procedures for model checking a linear temporal formula by checking the validity of an LTL formula expressing both the transition structure and the property. In particular, one can expect that systems with large control circuitry cannot be efficiently model checked using this approach.

5.3 Mixed Simulation/Verification Strategy

As a mixed simulation/verification strategy, one can give a finite instantiation to integer variables, and arbitrary interpretations to uninterpreted functions and predicates. This is what is done today in an ad hoc manner. It appears that this method is very useful in finding bugs. To get good coverage, a sufficiently large finite instantiation must be used. The results of

sections 3 and 4 can be applied to get a heuristically good bound on the size of the finite instantiation to be used.

6 Conclusions

A concurrency model was presented which extends the regular synchronous concurrency models naturally, and can be obtained easily from user's descriptions with little effort. An extension of the language containment paradigm to this concurrency model was described.

Results were also presented which allow verification on finite instantiations of some subset of models whose state spaces are infinite. This was used to verify a memory model with a finite number of processors and buffers, integer data, and unbounded memory space ([HMLB95]). Only binary data and a few memory locations were used without losing any accuracy in verification. For cases, where finite instantiations cannot be used, we gave an algorithm for symbolic enumeration of the state space using BDDs.

Acknowledgments

Discussions with David Cyrluk and David Dill were very helpful. The first author is also thankful to the PVS team at SRI International, N. Shankar, D. Cyrluk, M. Srivas, S. Owre, and John Rushby for help in using PVS, and to SRI International for facilitating an extended visit. This work was supported by SRC under grant DC-94-008.

References

[Bry86] R. E. Bryant, "*Graph Based Algorithms for Boolean Function Manipulation*", IEEE Trans. on Computers, C-35(8):677-691, August 1986.

[BD94] J. Burch, D. Dill, "*Automated Verification of Pipelined Micro-processors*", Computer-Aided Verification, 1994.

[CN94] D. Cyrluk, P. Narendran, "*Ground Temporal Logic: A Logic for Hardware Verification*", Computer-Aided Verification, 1994.

[HB95] R. Hojati, R. K. Brayton, "*An Environment for Formal Verification Based On Symbolic Computations*", Journal of Formal Methods, 1995.

[HBK94] R. Hojati, S. Krishnan, R. K. Brayton, "*Heuristic Algorithms for Early Quantification and Partial Product Minimization*", ERL Memorandum M94/11, March 1994, UC Berkeley.

[HMLB95] R. Hojati, R. Mueller-Thuns, P. Lowenstein, R. K. Brayton, "*Automatic Verification of Memory System Using Language Containment and Abstraction*", to be submitted to CHDL 95.

[ID93] C. N. Ip, D. Dill, "*Better Verification through Symmetry*", Symp. on Computer Hardware Description Languages and Their Application, 1993..

[MPS92] E. Macii, B. Plessier, F. Somenzi, "*Verification of Systems Containing Counters*", IEEE/ ACM International Conference on Computer-Aided Design, 1992.

[Kur92] R. P. Kurshan, "*Automata-Theoretic Verification of Coordinating Processes*", UC Berkeley notes, 1992.

[PVS93] N. Shankar, S. Owre, J. M. Rushby, "*The PVS Specification and Verification System*", SRI International, 1993.

[Sho79] R. E. Shostak, "*A Practical Decision Procedure for Arithmetic With Function Symbols*", JACM Volume 26, No. 2, April 1979, pp. 351-360.

[Wol86] P. Wolper, "*Expressing Interesting Properties of Programs*", 13th Annual ACM Symp. on Principles of Prog. Languages, 1986.