FORMAL VERIFICATION USING THE INTEGER COMBINATIONAL/SEQUENTIAL (ICS) CONCURRENCY MODEL

by

Adrian J. Isles

Memorandum No. UCB/ERL M97/39

28 May 1997

FORMAL VERIFICATION USING THE INTEGER COMBINATIONAL/SEQUENTIAL (ICS) CONCURRENCY MODEL

by

Adrian J. Isles

Memorandum No. UCB/ERL M97/39

28 May 1997

ELECTRONICS RESEARCH LABORATORY

College of Engineering
University of California, Berkeley
94720

Formal Verification Using the Integer Combinational/Sequential (ICS) Concurrency Model

Adrian J. Isles

University of California
Berkeley, CA 94720

Department of Electrical Engineering
and Computer Sciences

Abstract

The Integer Combinational/Sequential Concurrency (ICS) Model *[HB95]* uses integer variables, interpreted and uninterpreted integer functions and infinite memory to represent datapaths. By modeling the datapaths of hardware systems such as microprocessors in this way, verifying the control logic of such systems becomes a simpler computational task. In this thesis, four new contributions are made, outlined below, for verifying ICS Models.

First, an overview of ICS Models are given. In its original definition, the operational semantics of ICS Models allowed invalid behavior to occur for which we were unable to remove in its symbolic language. A new definition of ICS symbolic languages is given such that this inconsistency is removed.

For a certain class of ICS models, the symbolic representation of the datapath can be replaced with a datapath whose width is as small as one or a few bits. *[HB95]* presented an abstraction technique, called finite instantiations, for performing this reduction, and gave a lower bound on the number of bits needed. A new abstraction technique, called *data shifting derivations*, is proposed, which has a state space smaller than what can be achieved using finite instantiations. In addition, an open problem is solved by showing that verification of a certain class of ICS Models, with predicate $x < y$, is decidable. In *[HB95]*, it was shown that no finite instantiation exists for this class. Finally, for cases where automatic abstraction cannot be performed, ICS reachability analysis can be used to perform verification. In this report, the implementation of our first generation ICS reachability tool is presented. Experimental results are given for verifying two simple microprocessors.

R. K. Brayton

Professor Robert K. Brayton
Research Advisor

Contents

List of Lemmas and Theorems

List of Figures

List of Tables

Acknowledgments

I would like to thank my advisor, Professor Brayton, for his support and guidance during this project. In addition, I would also like to thank Professor Henzinger for agreeing to be a member of my committee.

The work in this report was done through close interaction with Ramin Hojati. Ramin developed ICS Models and my research was built upon this and other mathematical foundations developed by him in his Ph.D. dissertation. I would like to thank him for introducing me to ICS Models and processor verification and for all the guidance and encouragement that he has given me throughout this endeavor.

I would also like to thank the Ford Foundation and the SRC (grant number DC-324-033) for financially supporting me. In addition, I would like to thank members of the U.C. Berkeley CAD Group with whom I have had the pleasure of interacting with and who have made my research at Berkeley a more enjoyable and intellectually rewarding experience. I would also like to thank Shelia Humphreys, the Black Graduate Engineering and Science Student Association (BGESS) and the Brothers of MTFO for their support. Last, but certainly not least, I would like to thank my parents, family members and friends for the moral support that they have given me. Their support played a crucial role in allowing me to finish this work.

Chapter 1

Introduction

As the engineer's quest to build a faster computer continues, she is forced to employ increasingly more advanced computer architectural techniques and algorithms in her design. These techniques, some of which were previously only implemented in expensive supercomputers, can now be found in most desktop microprocessors *[HP90]*. One of the most popular architectural approaches used to minimize the time it takes a computer to execute a sequential program, is to design computers that can take advantage of instruction level parallelism. Here, the intuition is that for some sequences of instructions, the order in which they are executed by the computer may not change the final result of the program. This added degree of flexibility can be exploited to optimize program execution time, since each instruction may require a different computational resource or may take a varying number of clock cycles to complete. Some common techniques that take advantage of this "fine grain parallelism" include executing multiple instructions simultaneously (superscalar microprocessors), executing instructions out of order, completing instructions out of order, and performing branch predication. Designs can be further complicated by the addition of other architectural features, such as precise interrupt handling. As a result of these complexities, the engineer typically has a very difficult time insuring that her design executes instructions in such a way that the correct result is always produced. In fact, insuring that a computer always sequences instructions correctly can dominate its design cycle, and usually results in a validation problem that is significantly more difficult (in practice) than simply checking the correctness of a particular computational element (i.e. checking if the arithmetic logic unit (ALU) in the datapath adds two numbers correctly). Thus, most bugs that occur during the design of a modern microprocessor are because of errors in control logic where these advanced architectural features are typically implemented *[Rho95]*.

The validation of modern microprocessors is currently performed via simulation. Here, the architectural model and the implementation are simulated in parallel using a common set of instructions. If, after executing the same program, the output of the architectural model differs from the output of the implementation, then a bug in the microprocessor exists[1]. It is, however, mathematically intractable to simulate all possible input sequences a design may be subject to in a reasonable amount of time. Thus, in reality, simulation cannot guarantee the absence of bugs even for small designs.

Formal verification is a body of techniques and algorithms that allows the engineer to mathematically prove whether certain properties hold for a design. Verification is ***complete*** since it has the ability to show that no bugs (with respect to the property being verified) exists in a design for all inputs and all sequences of inputs.

1. This may not be the case if the instruction set architecture allows for more than one acceptable behavior.

In addition, using ***Binary Decision Diagrams*** (BDDs) *[Bry86]* technology, designs that have very large state spaces (10^{20} states or more) can be verified in only a few CPU hours *[BCMD90, Cou89, Tou90]*. These techniques are limited, however, in that they can only handle relatively small designs. This is because most automatic verification algorithms work by visiting, either explicitly or implicitly, every state of the design. However, it can be shown that the size of the reachable state space of a design is, in the worst case, exponential in the size of its description. This "blowup" in the number of states is called the ***state explosion problem***. In microprocessors, this problem is particularly aggravated by the existence of wide datapath elements and memories. However, since most microprocessor bugs occur in the control part of the circuit, the computational complexity of the verification task can be significantly reduced by abstracting away the least-error prone portion of the design: the datapath. The Integer Combination/Sequential concurrency (ICS) Model, first introduced in *[HB95]*, is a paradigm for performing verification using an abstract representation of the datapath.

ICS Models use integer variables, interpreted and uninterpreted integer functions, integer predicates, and infinite memory to abstract away functional units, register files, and memory, thereby substantially reducing the complexity of the datapath. Because abstract data types and operators are used, the state space of systems modeled using ICS is, in general, infinite. Despite this, two approaches can still be used for verification. In certain cases, depending of the type of design and property being verified, a system modeled using ICS can be automatically abstracted into a equivalent system where the width of the datapath has been reduced to only one or a few bits. Traditional state exploration techniques can then be used to check properties on the design. For cases where this is not possible, ICS reachability analysis can be used for verification. This algorithm uses a novel hybrid approach for generating the set of reachable states. Here, BDDs are used to represent transitions in the control portion of a design, and a special form of symbolic simulation is used to represent transitions in the integer datapath. Since the state space may be infinite, reachability analysis may not always terminate. However, the reachable state set, or an approximation of it, can be used for complete or partial verification using model checking or language containment. This thesis makes new contributions to both approaches for verifying ICS models.

1.1 An Outline of this Work

In Chapter 2, ICS Models are formally defined. The original definition appeared in *[HB95]*. Subsequently, it was discovered that there is an inherent problem in the model. This problem occurs because the symbolic representation of the datapath used in ICS Models, allows behavior that may not have a concrete representation *[HIB97]*. A redefinition of symbolic ICS automata and languages is given in this chapter and it is proved that the language of the composition of two machines is equal to the intersection of the languages of the two machines. In addition, Boolean operations on languages, such as union and intersection, have also be redefined.

In Chapter 3, three new results are presented which give bounds on the minimum number of values needed in the datapath to accurately verify certain types of ICS Models. *[HB95]* introduced an automatic abstraction technique, called ***finite instantiations***, where the integer domains are automatically replaced with finite values consisting of only one or a few bits, without any loss of accuracy in verification. First, we show that for a class of ICS Models called ***Data Semi-Sensitive Controllers*** (DSSC's), where only integer predicates of the form $x = y$ are used, the number of bits needed in the integer datapath for verification can be reduced significantly from what was earlier reported *[HB95]*. Next, a new abstraction technique, called ***data shifting derivations***, is proposed that allows the number of bits needed in the datapath to be reduced even further, when compared to finite instantiations. Finally, a previously open problem in the verification of ICS Models is solved by showing that verification of DSSC's with predicates $x < y$ and $x = y$ is decidable, and an exact data shifting derivation exists. Previously, it was proved that ***no*** finite instantiations exist for this type of controller *[HB95]*. Proofs and bounds are given for all the approaches described in this chapter.

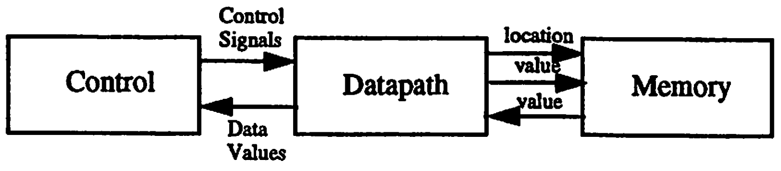
Finally, in chapter 4, the reachability algorithm, as first described in *[HB95]*, is explained with minor modifications to the original definition. Experimental results, produced by the first generation ICS reachability tool, is then presented. In particular, results are given for computing the set of reachable states on a simple design. A major bottleneck, associated with this algorithm, is also discussed. Finally, it is demonstrated how ICS reachability can be employed in solving the specific problem of pipelined processor verification. Here, one would like to verify that a pipelined microprocessor implements its instruction set architecture. In *[BD94]*, Burch and Dill presented a unique approach for solving this problem. ICS reachability can be used to approximate this approach. Experimental results are given for two simple pipelined microprocessors. Comparisons to other approaches are given.

Chapter 2

An Introduction to ICS Models

2.1 Introduction

In this chapter, an introduction is given to the mathematical framework upon which the key results in this thesis are based: ICS Models. ICS Models extend traditional C/S Models *[Hoj96]* by modeling hardware at a higher level of abstraction using integer variables, integer functions and infinite memory. Modeling hardware at this higher level simplifies the verification process by removing some of the lower level details of a design that may not be needed to verify certain properties. ICS Models represent hardware by dividing it into three separate components: control, datapath and memory. Control consists of finite gates and latches, and its semantics is very similar to C/S models. The datapath consists of gates and latches whose functionality is defined over integer variables with infinite domains. Finally, memory represents an infinite table of address/data pairs. As with datapath, addresses and data are represented using integer values. A diagram of these three components and their interactions is shown in figure 2.1.



ICS Models consist of three separate components: Control, Datapath and Memory. This closely resembles the structure of modern hardware systems.

Figure 2.1: ICS Models

The verification of ICS Models can proceed along two paths. Depending on the functionality of the system and the property to be verified, automatic abstraction techniques have been developed that reduce a system modeled using ICS to an equivalent system modeled using C/S semantics. A traditional state exploration based verification tool can then be used to verify the simplified design. This type of automatic abstraction, called finite instantiations, cannot work for all types of systems modeled with ICS. In these cases, an ICS reachability algorithm can be used to symbolically generate the state space. In general, since systems modeled using ICS have infinite state spaces, reachability will not always terminate. In cases, however, where reachability does terminate, a model checking or language containment algorithm can be used to prove properties on the state space. Otherwise, the reachability tool can be used to partially generate the state space for a finite number of cycles. Here again, model checking or language containment can be used to perform verification on the portion of the state space that has been explored. Hence, one can only guarantee that error traces of a certain length do not exist. Since, in practice, most error traces are short, this type of partial verification may still prove useful for catching bugs.

The outline of the rest of this chapter is as follows. In Section 2.2, ICS terms, which are a symbolic representation used to describe the behavior of the datapath and memory is defined. Section 2.3 discusses the syntax of ICS models and the types of operations and gates that are allowed in each of the three ICS components. Section 2.3 defines state space of ICS models, and Section 2.4 describes the ICS semantics. Finally, Section 2.5 discusses how ICS Models are verified. Note that the definition of ICS Models and how to verify them is slightly different than what was first given in *[HB95]*. The motivation for this re-definition is given and comparisons to the old definitions are made.

2.2 ICS Terms: A Symbolic Representation of the Datapath Behavior.

ICS Models consist of variables that are either ***integer*** or ***finite***. If a variable v is finite and has an n valued domain, then it can be assigned values $\{0, 1, \ldots, n-1\}$. If v is an integer variable, then it can take on an infinite number of values. Here, in order to efficiently represent behavior in the datapath, symbolic expressions called ***ICS terms*** are assigned to integer variables instead of concrete values. ICS terms are defined recursively and consist of symbolic constants, interpreted integer functions and uninterpreted integer functions applied to ICS terms. Each of these mathematical objects is described below:

Symbolic Constants. A symbolic constant is a symbol which represents any integer value. Intuitively, one can think of symbolic constants as symbols that can be assigned an integer value only once. Thus, the relationship between multiple symbolic constants can never change once a particular relationship has been assumed. For example, if c_1 and c_2 are symbolic constants, and in state s_i it is assumed that $c_1 = c_2$, then in all successor states of s_i, $c_1 = c_2$ must be true (this would also be the case if $c_1 < c_2$, $c_1 > c_2$ or any other relationship between c_1 and c_2 assumed in s_i).

Interpreted functions. Interpreted functions are integer functions that only have a single interpretation. ICS Models only permit a small subset of all such functions to be used. These functions are $z{:=}x + y$, $z{:=}x + c$, $x{:=}y$, $z{:=}mux(b, x, y)$, and $z{:=}if(b,x)$, where x, y are symbolic integer variables, c is an integer and b is a binary variable. Examples of ICS terms that contain interpreted functions include $c_0 + c_1$ and $mux(0, c_0, c_1)$, assuming that c_0, c_1 are also ICS terms.

Uninterpreted functions. An n-ary uninterpreted function, $f(x_0, \ldots, x_{n-1})$, takes n integer variables as input and returns a symbolic integer value. Thus, if i is an ICS term that is the input of an unary uninterpreted function symbol f, then the output would be the ICS term $f(i)$. Since no interpretation of f is given, it can be used to represent all possible interpreted functions. Uninterpreted functions are very useful for abstracting away details of a design that are not needed for verification. This is particularly important when verifying, for example, the control logic of a superscaler microprocessors. Here, verification is independent of the functional units in the datapath, since one only needs to check whether the processor executes each instruction in the proper sequence and writes back the result to the correct location. Therefore, the detailed functionality of the datapath is not relevant and can be replaced with uninterpreted functions.

An example of a property of a microprocessor that one may want to verify is that the control logic fetches, decodes and write backs the instruction $I{:}R_3 \leftarrow R_1\ sub\ R_2$ correctly. Here, I subtracts the values in the registers R_1 and R_2 and places the result in the register R_3. The control logic can be verified by replacing the subtraction hardware with an uninterpreted function f. Assuming that c_0 and c_1 are the correct values assigned to registers R_1 and R_2 when I is issued, then one only has to check that when I retries, R_3 contains the ICS term $f(c_0, c_1)$. Note that replacing a concrete function with an uninterpreted function is conservative, i.e. more behavior is modeled using uninterpreted functions.

In order to describe the relationship between ICS terms, ***predicates*** are used. Predicates are special functions that, when given ICS terms as arguments, return a binary value. Two examples of these functions are $c_0 = f(c_1)$ and $f(c_1) < g(c_2)$. Predicates can either be interpreted or uninterpreted. The only interpreted predicates allowed are $x = y$, $x<y$, $x = c$, $(x\ mod\ y) = r$ and $(x\ mod\ y) < r$, where x and y are symbolic integer variables and r and c are integers. An n-ary uninterpreted predicate, $p(x_0, \ldots, x_{n-1})$, takes n integer variables as inputs and returns a binary value. Note that since no interpretation of p is given, it can be used to represent all possible interpreted predicates.

A ***concrete interpretation*** of an ICS term t, is performed by replacing all symbolic constants in t with integer values, replacing all uninterpreted integer functions in t with interpreted integer functions and replacing all uninterpreted integer predicates with interpreted integer predicates. Here, any interpreted function or predicate can be used, not just those described above.

Given a set of ICS terms $\{t_1, \ldots, t_n\}$, and assuming that R_p is an interpreted or uninterpreted predicate, one can determine if $R_p(t_1, \ldots, t_n)$ is true with respect to another predicate p (denoted $R_p(t_1, \ldots, t_n)|_p$ or $(t_1\ R_p\ t_2)|_p$, if R_p is a binary predicate) if the formula $p \rightarrow R_p(t_1, \ldots, t_n)$ is ***valid***. Such a formula is valid, if it is true for all concrete interpretations. For example, if $t_1 = f(c_1)$, $t_2 = f(c_2)$ and $p \equiv (c_1 = c_2)$, then $t_1 = t_2|_p$ is valid since $(c_1 = c_2) \rightarrow (f(c_1) = f(c_2))$ is always true no matter what interpretations are given to f, c_1 and c_2. A formula $p \rightarrow R_P(t_1, \ldots, t_n)$ is said to be ***satisfiable*** if there exist an interpretation for which $p \rightarrow R_P(t_1, \ldots, t_n)$ is true. Determining if two ICS terms are equal, subject a predicate (or set of predicates), is decidable and can be determined using the algorithms given by *[Sho79]* and *[Sho82]*. Note that checking if the binary predicate $(t_1 \; R_P \; t_2)$ is true, with respect to two predicates p and q, denoted $t_1|_p \; R_P \; t_2|_q$ can be found by checking the validity of the formula $(p \wedge q) \rightarrow (t_1 \; R_P \; t_2)$.

2.3 The ICS Model Syntax

Systems modeled using ICS models are described by a network of circuit primitives. Using these primitives, a wide variety of hardware systems that operate on integer data can be modeled without having to specify the number of bits in the datapath of the actual hardware implementation. A circuit primitive can either be a ***generalized gate*** or a ***latch***. Generalized gates can be classified into those that control data, gates that allow integer data to be moved around in the datapath, gates that perform computation on data, data constant creator gates that simulate integer inputs, and memory gates that allow for the storage of data. Generalized gates can have multiple input variables but only a single output variable. ICS Models are ***closed***,i.e. they have no inputs. In the following, each of these types of gates is described in detail.

Control gates. A control gate defines a relation over a set of finite variables.These gates can be placed only in the control portion of the model. A gate is considered ***deterministic*** if there is only one output for any set of input variable assignments. For simplicity, it is assumed that each relation is over multiple input variables and a single output variable. ***Primary control inputs*** are represented using non-deterministic gates that have no inputs and a single output, and can produce any value in the domain of the input variable.

Data comparison gates. A data comparison gate can be either an interpreted or uninterpreted predicate. Since predicates take a set of symbolic integers and return a finite binary value, they provide the only mechanism by which the datapath can communicate with the control.

Data movement gates. A data movement gate can be used to move integer values around in the datapath. There are three data movement operations allowed in the datapath: $x{:=}y$, $z{:=}mux(b, x, y)$, and $z{:=}if(b,x)$. Here b is a binary variable and x, y and z are symbolic integer variables. The control can manipulate the flow of integer data by changing the value of b. The control inputs of the data movement gates provide the only means of communication from the control to the datapath.

Data computation gates. A data computation gate is used to represent a function that, given a set of symbolic integer variables as inputs, returns an integer value. Such a gate can consist of the interpreted integer functions $z{:=}x + y$, $z{:=}x + c$ or an uninterpreted function. These gates provide the only means by which computation can be performed on integer data in the datapath.

Data constant creator gates. A data constant creator gate has no input variables and a single symbolic integer output variable. This gate creates a new symbolic constant each time it is called. This constant is called a ***fresh constant***, and denotes a new symbolic constant that has never been previously assigned and does not currently occur as a formula or sub-formula in any of the integer variables in the model. Data constant creator gates are used to model unconstrained integer inputs.

Memory Gates. A memory gate can either be of type ***read*** or ***write***. As discussed in Section 2.1, in ICS models, memory represents an infinite set of address/data pairs[1]. A read takes a memory (ICS allows for circuits with multiple memories) and an ICS term representing an address and returns a ICS term that was last written to that location. If the location being accessed has never been written to before, a fresh constant is returned. Write takes a memory, an address, and a data value and sets the address in memory to hold that particular value.

Latches are defined using two variables called the ***present state variable*** and the ***next state variable***. The present and next state variables of latches must be over the same domain and can be either finite or integer. The initial state of a latch is a value or subset of values of the state variables domain. If a latch has an integer domain, then a constant and predicates can be used to describe a subset of the integer domain that are valid initial states.

2.4 The ICS State Space

A state $s = (latches, memories, predicates)$ is a triple of assignments to latches, memories and predicates. Since the set of all ICS terms that can be assigned to latches, memories or predicates in infinite, the state space may also be infinite. Two states $s_0 = \left(l_0, \left\{m_0^0, \ldots, m_0^{n-1}\right\}, p_0\right)$ and $s_1 = \left(l_1, \left\{m_1^0, \ldots, m_1^{n-1}\right\}, p_1\right)$ are equal in a given model M, denoted $s_0 = s_1$, if the following three conditions hold.

1. For the i-th latch in M, assume s_0 assigns it the value α_0^i and s_1 assigns it the value α_1^i. If α_0^i is finite

[1] *Although ICS memory represents an infinite set of address values pairs, the ICS operational semantics requires the explicit representation of only those memory locations that have been written to on a previous cycle. Thus, for a finite number of cycles, the number of memory locations that have to be stored is always finite.*

then $\alpha_0^i = \alpha_1^i$, otherwise $\alpha_0^i\big|_{p_0} = \alpha_1^i\big|_{p_1}$.

2. Assume $m_0^k = \left\{\left(a_0^k[0], v_0^k[0]\right), \ldots, \left(a_0^k[z], v_0^k[z]\right)\right\}$ is given in s_0, where $\left(a_0^k[i], v_0^k[i]\right)$ is an address/value pair in m_0^k. For each such pair, there exists $\left(a_1^k[j], v_1^k[j]\right)$ in m_1^k of s_1 such that $a_0^k[i]\big|_{p_0} = a_1^k[j]\big|_{p_1}$ and $v_0^k[i]\big|_{p_0} = v_1^k[j]\big|_{p_1}$. The reverse should also hold. For each $\left(a_1^k[j], v_1^k[j]\right)$ in m_1^k, there exists a $\left(a_0^k[i], v_0^k[i]\right)$ in m_0^k such that $a_0^k[i]\big|_{p_0} = a_1^k[j]\big|_{p_1}$ and $v_0^k[i]\big|_{p_0} = v_1^k[j]\big|_{p_1}$.

3. $p_1 \rightarrow p_2$ and $p_2 \rightarrow p_1$ are both valid.

The initial state of M is a state $s_{init} = (l_{init}, \varnothing, \varnothing)$, where l_{init} a set of initial assignments given to all latches in M.

2.5 The Operational Semantics of ICS Models

The operational semantics of ICS Models defines how transitions are made between states. The semantics is very similar to that of C/S models. Here, a topological sort of the gates in the model is first chosen. Then, starting from the primary inputs and present state latch variables, values are propagated through the network. The key difference between C/S and ICS occurs when simulating the datapath. Here, ICS terms are assigned to symbolic integer variables instead of concrete finite values. When an integer predicate is evaluated, if the set of predicates defined at the current state implies the gate output value, then it is assigned that value. Otherwise, two paths of propagation are created. One path assumes that the value of the predicate to be 0, the other assumes it to be 1.

2.5.1 Definition of ICS Operational Semantics

A ***gate graph*** G, of a ICS Model M, is a graph where each vertex v represents a gate in M and each edge (u, v) represents a variable that is the output of gate u and the input of a gate v. A topological sort is then created by starting from latches and gates that have no inputs (control primary inputs and constant creators) and visiting each vertex in breath first order. Since their can be no combinational cycles in M, G also contains no cycles. Given a topological sort O of G, and a state $s = (l, m, p)$, the operational semantics of ICS Models is given below, and defines a set S', where each $(l'_j, m'_j, p'_j) \in S'$ is a next state of s. In the following, let V be the set where each $v_j \in V$ is a set of assignments to a subset of state and non-state variables of M made during the symbolic execution of its gate graph. Intuitively, one can think of each v_j as an alternative "path" of execution starting from s.

1. Initially, let $z = 1$ and note that in the following that $z = |V|$. Create $v_1 \in V$ and assign to the present state variables in v_1 values assigned to the corresponding variables in s.

2 Propagate values through M, by assigning values to the output of each generalized gate, g, in the order given by O. In particular, for each $v_j \in V$, and for each generalized gate $o = g(i)$, where i is the set of input variables and o is the output variable of g, and α_i is the set of assignments to i by v_j ($(i = \alpha_i) \in v_j$), the following cases may occur.

2a. If g is a table that represents the relation $R(i, o)$, then assign o a finite value α_o such that $(\alpha_i, \alpha_o) \in R$ and let $v_j = v_j \cup \{o = \alpha_o\}$. If there are multiple values, α'_o, such that $(\alpha_i, \alpha'_o) \in R$, then for each such value where $\alpha'_o \neq \alpha_o$, let $z = z + 1$, and create a new set of assignments v_z such that $V = V \cup v_z$ and $v_z = v_j \cup \{o = \alpha'_o\}$. Finally, if there are no values α_o such that $(\alpha_i, \alpha_o) \in R$, let $v_j = v_j \cup \{o = \emptyset\}$. This corresponds to stopping value propagations along this path.

2b. if g is an interpreted or uninterpreted function, then let $\alpha_o = g(\alpha_i)$, where both α_i and α_o are ICS terms. Let $v_j = v_j \cup \{o = \alpha_o\}$.

2c. If g is an interpreted or uninterpreted predicate R_P, let p_j be the predicates associated with v_j. If $p_j \rightarrow (R_P(\alpha_i) = b)$ is valid, then assign $\alpha_o = b$ and let $v_j = v_j \cup \{o = \alpha_o\}$ (where α_i is a set of ICS terms and b is a binary value). Otherwise, let $z = z + 1$, and create a new set of assignments v_z such that $V = V \cup v_z$, $v_z = v_j \cup \{o = \alpha_o\}$, $p_z = p_j \wedge (R_P(\alpha_i) = \alpha_o)$. In addition, let

$$v_j = v_j \cup \left\{o = \overline{\alpha_o}\right\} \text{ and } p_j = p_j \wedge (R_P(\alpha_i) = \overline{\alpha_o}).$$

2d. If g is a constant creator, then let $\alpha_o = c_i$ and $v_j = v_j \cup \{o = \alpha_o\}$. Here c_i is a fresh constant: a symbolic constant that has never been assigned to an integer variable in v_j or any state reachable from s. A sufficient, but not necessary condition to guarantee this is to return a unique fresh constant each *time the gate is called.*

2e. If g is a read memory gate, let m_j be the memory associated with g and assigned by v_j, and $a \in \alpha_i$ the address being read. If there is a location a' in memory, such that $a = a'|_{p_j}$ is valid, then assign $\alpha_o = m_j[a']$ and $v_j = v_j \cup \{o = \alpha_o\}$. Otherwise, for each location a' in memory where $a \neq a'|_{p_j}$ is not valid, let $z = z + 1$, and create a new set of assignments v_z where $v_z = v_j$ (note that this implies that $m_z = m_j$ and $p_z = p_j$), $p_z = p_z \wedge (a = a')$, $\alpha_o = m_z[a']$ and $v_z = v_z \cup \{o = \alpha_o\}$. Let $V = V \cup v_z$. The case where $(a = a')$ is not valid for any a' also has to be considered. For each location a' in memory where $a \neq a'|_{p_j}$ is not valid, let $p_j = p_j \wedge (a \neq a')$, create a new fresh constant d, set $m_j[a] = d$, $\alpha_o = d$ and $v_j = v_j \cup \{o = \alpha_o\}$. This corresponds to reading a location that has never been written to before.

2f. If g is a write memory gate, let m_j be the memory associated with g and assigned by v_j. Also, assume that a is the address and d is the data. If there is a location a' in memory, such that $a = a'|_{p_j}$ is valid, then assign $m_j[a'] = d$. Otherwise, for each location a' in memory where $a \neq a'|_{p_j}$ is not valid, let $z = z + 1$ and create a new set of assignments v_z where $v_z = v_j$, $p_z = p_z \wedge (a = a')$ and $m_z[a'] = d$. Let $V = V \cup v_z$. The case where $(a = a')$ is not valid for any a' also has to be considered. For each location a' in memory where $a \neq a'|_{p_j}$ is not valid, let $p_j = p_j \wedge (a \neq a')$ and let $m_j[a] = d$.

3. For each $v_j \in V$, assign to $s_j' = (l_j', m_j', p_j')$ the corresponding values given to the next state latch values, memory and predicate variables in v_j.

Note that at the end of 2, z is finite, i.e. s always has a finite number of next states.

2.5.2 Comparison to ICS Operational Semantics Defined in *[HB95]*

The operational semantics of ICS Models given in *[HB95]* is defined in terms of configuration graphs. A configuration graph, G_M, for a model M, is an acyclic graph where each node (or state) is a pair $u = (s, n)$. Here, s was equivalent to the definition of a state of M, given in Section 2.3, and n is a defined as the number of fresh constants created in all states proceeding u. The initial state of M is represented by a vertex

(s_0, n_0), where s_0 is also defined as in Section 2.3 and n_0 is the number of constants in s_0. Given, a topological sort O of M, and a state with $u = ((L, M, P), n)$, a transition to a state $v = ((L', M', P'), n)$ is defined in a similar fashion to that in Section 2.5.1. The difference is that each state is defined as a pair (s, n) instead of just s as given in Section 2.3. In addition, every time a constant creator is called, a new fresh constant named $c_{n'}$ is returned and $n' = n' + 1$ (initially, $n' = n$). This definition insures that along any path of the configuration graph, no two fresh constants were ever given the same name. The operational semantics in Section 2.5.1 also guarantees that no two fresh constants are given the same name along the same path. However, it does not require a counter to be used nor does it specify a particular name.

2.6 The Verification of ICS Models.

The verification of ICS Models is performed using the language containment paradigm *([VW86],[Kur87])*. Language Containment allows a property to be described using an automaton called a property automaton. Verifying that an ICS Model M satisfies a property P can be performed by showing that the language of M is contained in the language of P, denoted $L(M) \subseteq L(P)$. In this section, the definition of ICS languages presented is different that was first given in *[HB95]*. It is then proved that for this new definition, that composition of two models has a language that is equal to the language intersection of both models

.

2.6.1 ICS Automata and Languages

The ***symbolic automaton*** of an ICS Model M, A_M, is a graph of states and transitions between states defined by the operational semantics described in Section 2.4. The ***alphabet*** of A_M, Σ_M, is the Cartesian product of the alphabets of non-state variables and the predicate state variable. Following *[Kur87]*, we refer to these variables as ***selection variables***. For finite variables, the assignments come from their finite ranges. For integer variables and predicates, this is infinite (but countable) set. An ***ω-string*** is an infinite sequence of assignments to the selection variables of M. Let the ***ω-string*** $x = \sigma_0, \sigma_1, \sigma_2, \ldots$ be given. An ***ω-run*** $e = s_0, s_1, s_2, \ldots$ of x is an infinite sequence of states in A_M, where s_0 is an ***initial state*** of A_M, and for every $i > 0$ there is a transition $(s_{i-1}, \sigma_{i-1}, s_i)$ in A_M. Edge-Street *[HSB94]* fairness constraints are used to define the set of acceptable runs of A_M, by placing restrictions on the values of finite latches. A run e of A_M is considered fair if all fairness constraints are satisfied. This implies that the set of assignments that occur infinitely often to the latch control variables in e satisfies all fairness constraints. The ***symbolic language*** of A_M, $L(M)$, is the set of all ω-stings that have fair runs.

In order to define boolean operations on symbolic languages of automata, it is necessary to first define a notion of equivalence between two ICS ω-strings.

Definition 2.6.1.1 Two ω-strings $x = \sigma_0, \sigma_1, \sigma_2, \ldots$ and $\tilde{x} = \tilde{\sigma}_0, \tilde{\sigma}_1, \tilde{\sigma}_2, \ldots$ are equivalent, denoted $x \approx \tilde{x}$, if the following two conditions hold for all i:

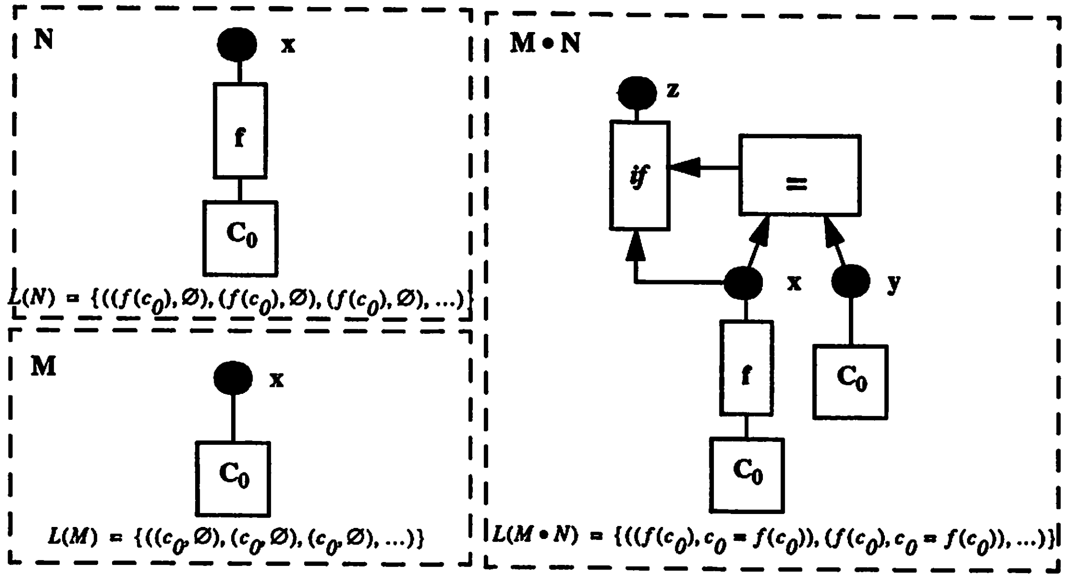
1. For each finite selection variable v_j, $\sigma_i[v_j] = \tilde{\sigma}_i[v_j]$ and

2. The formula $\prod_{\forall i, j} (p_i \wedge \tilde{p}_i \wedge (\sigma_i[v_j] = \tilde{\sigma}_i[v_j]))$ must be satisfiable. Here, v_j is an integer selection variable and ($\sigma_i[v_j]$ and $\tilde{\sigma}_i[v_j]$ are ICS terms). Also, p_i ($\tilde{p}_i$) represents assignments to the predicate selection variable by σ_i ($\tilde{\sigma}_i$).

The definition of ***containment*** for two languages follows naturally from the above definition. Let L_1, L_2, and L_3 be three symbolic languages. L_1 is contained in L_2, denoted $L_1 \subseteq L_2$, if for every ω-string $x \in L_1$, there exist an ω-string $y \in L_2$ such that $x \approx y$. L_3 is the ***intersection*** of L_1 and L_2, denoted $L_3 = L_1 \cap L_2$, if for every ω-string $x \in L_3$, there exists two ω-strings $y \in L_1$ and $z \in L_2$ such that $x \approx y$ and $x \approx z$. L_3 is the ***union*** of L_1 and L_2, denoted $L_3 = L_1 \cup L_2$, if for every ω-string $x \in L_3$, there exists an ω-string y such that $y \in L_1$ and $x \approx y$, or there exist an ω-string $z \in L_2$ such that $x \approx z$. If L_2 is the ***complement*** of L_1, denoted $L_2 = \bar{L}_1$, then L_2 contains the set of all ω-strings x such that there does not exist a $y \in L_1$ where $x \approx y$.

Given two ICS models M and N, their ***composition***, denoted $M \bullet N$, can be constructed as follows. If M and N have no selection variables in common, then their composition is simply their syntactic composition. If, however, M and N do have selection variables in common, then for each such pair of variables both named x_i, rename one of them y_i and add a new gate g and a new variable z_i, which implements the function $z_i := if(x_i = y_i, x_i)$. The initial states of $M \bullet N$ are composed in a similar fashion. Observe that except for g, the new variable z_i is not connected to any other gates. Also note that the behavior of the function $z := if(b, x)$ is empty if $b = 0$. The language of $M \bullet N$ is defined over the selection variables that were not renamed. An example is given in Figure 2.6.1.1.

In order for language containment to be useful in practice, it is important that the language of the machine created by the composition of two models be equal to the intersection of the languages of both models. Lemma 2.6.1.1 proves this.



N
x
f
C0
L(N) = {((f(c0),∅),(f(c0),∅),(f(c0),∅),...)}
M
x
C0
L(M) = {((c0,∅),(c0,∅),(c0,∅),...)}
M•N
z
if
=
x
y
f
C0
C0
L(M•N) = {((f(c0),c0 = f(c0)),(f(c0),c0 = f(c0)),...)}

M and N are both one state automata. The selection variable x of M is always assigned c_0, but the same variable x of N is always assigned $f(c_0)$. In the composition $M \bullet N$, the variable corresponding to M is renamed y and x is always assigned $f(c_0)$. Note, however, that the predicate $c_0 = f(c_0)$ is assigned in the only string in $L(M \bullet N)$

Figure 2.6.1.1: An Example of ICS Composition

Lemma 2.6.1.1 Given two ICS Models M and N and their composite machine $C = M \bullet N$, $L(C) = L(M) \cap L(N)$.

Proof $L(C) = L(M) \cap L(N)$ implies that for every string $x^C \in L(C)$, there exist two strings $x^M \in L(M)$, $x^N \in L(N)$ such that $x^C \approx x^M$ and $x^C \approx x^N$. The reverse has to hold, but will not be proved here, since its proof is similar.

Let $G^C = G^C_M \cup G^C_N \cup G^C_{EQ}$ where G^C_M are the gates in C that came from M, G^C_N are the gates in C that came from N, and G^C_{EQ} are the gates in C of the form $z_i := if(x_i = y_i, x_i)$ introduced by the composition. Define the set of variables in C, V^C, where $V^C = V_M^C \cup V_N^C \cup V_{EQ}^C$ in a similar fashion.

To prove the above result, it will be shown that for every accepting run $e^C = s_0^C, s_1^C, s_2^C, \ldots$ of x^C in C, there exist a string $x^M \approx x^C$ such that x^M has an accepting run $e^M = s_0^M, s_1^M, s_2^M, \ldots$ in M. Note that the same argument below can be used for the existence of x^N. Each state $s_j^M \in e^M$ has the following two properties.

(1) If $v_j^C \in V_M^C$ then $v_j^C = v_j^M$, where $v_j^M \in V^M$ is the variable corresponding to v_j^C. More specifically, if v_j^M is an integer variable, it will be assigned an ICS term that is syntactically equivalent to the one given to v_j^C. If the variables are finite, then they will be given the same value. Observe that if there is a syntactic equivalence, then $v_j^C = v_j^M\big|_p$ is valid under all predicates p.

(2) $p_i^M \rightarrow (P = b)$ implies that $p_i^C \rightarrow (P = b)$ where P is any predicate, $b = 0$ or $b = 1$, and p_i^M (p_i^C) are the state predicates assigned in s_j^M (s_j^C). Note that if this is true, then $p_i^M \wedge p_i^C$ is always satisfiable.

The existence of e^M can be shown by induction on its length.

For the initial state s_0^C, if a present state latch variable $v_j^C \in V_M^C$ is a finite variable, then by definition, there exist an s_0^M such that $v_j^C = v_j^M$. In addition, if a latch variable $v_j^C \in V_M^C$ is an integer, then clearly $v_j^C = v_j^M$ is also trivially valid, since when C was constructed, it was given initial constants with the exact same names as in M. Note that by the definition of initial states in ICS Models, $p_0^C = p_0^M = \emptyset$.

Now assume that at a given state s_i^C, there exists a corresponding state s_i^M, where the above two properties holds. We will show that these properties will also hold for all assignments going from s_i^M to state s_{i+1}^M. Choose a topological sort of M that is consistent with the ordering given to the gates G_M^C in the topological sort of C. In the following, $\tilde{g}^C$ denotes the last gate that was symbolically executed in C before g_j^C, and $\tilde{p}^C$ the predicate given after $\tilde{g}^C$ was executed. This distinction is necessary because it may be true that $\tilde{g}^C \in G_N^C \cup G_{EQ}^C$ and thus $\tilde{g}^C$ has no corresponding gate in M. Also, let g_{j-1}^M denote the last gate in M that was symbolically executed before g_j^M. Let's assume that for all gates g_k^M, g_j^M $(k < j)$ the above properties hold. Then, the properties must also hold after processing g_j^M, because of the following cases:

(1) g_j^M is a constant creator. Assign v_j^M the same fresh constant that was given to v_j^C. Also, let $p_j^M = p_{j-1}^M$. Note that C sets $p_j^C = \tilde{p}^C$. Clearly, property 1 holds. Property 2 also holds. To show this, let's assume the reverse: $p_j^M \to (P = b)$ valid and $p_j^C \to (P = \bar{b})$ is valid for some predicate P. This case cannot occur and can be argued as follows. If $p_{j-1}^M \to (P = b)$ is valid, then at the corresponding gate in C, g_{j-1}^C, $p_{j-1}^C \to (P = b)$ is, by inductive assumption, also valid. This means that there was some gate $\hat{g}^C \in G_N^C \cup G_{EQ}^C$ was symbolically executed after g_{j-1}^C, but before g_j^C, that eventually forced $p_j^C \to (P = \bar{b})$. This, however, is not allowed by the operational semantics of ICS Models, since it earlier was assumed at g_{j-1}^C that $p_j^C \to (P = b)$ was valid. The case where $p_j^M \to (P = b)$ valid but neither $p_j^C \to (P = \bar{b})$ nor $p_j^C \to (P = b)$ is valid can also not occur and can be argued in a similar fashion.

(2) g_j^M represents an interpreted or uninterpreted function. Just apply the function to the inputs and let $P_j^M = P_{j-1}^M$. By inductive assumption, since the ICS terms assigned at the inputs of g_j^M are syntactically equivalent to those given at the inputs of g_j^C, property 1 will be valid for v_j^C. Property 2 also holds by using the same argument given in (1).

(3) g_j^M is a table. Just apply the relation to the inputs and let $P_j^M = P_{j-1}^M$. Once again, by inductive assumption, since the finite values given at the inputs of g_j^M and g_j^C are the same, then so are the outputs. Property 2 also holds by using the same argument given in (1).

(4) g_j^M represents an interpreted or uninterpreted integer predicate, $P(i)$. When g_j^M is executed, several cases may occur. In the following, assume that $b = 0$ or $b = 1$

(4a) $\tilde{p}^C \rightarrow (P(i) = b)$ and $p_{j-1}^M \rightarrow (P(i) = b)$ are valid. Simply assign v_j^M the value b.

(4b) Neither $\tilde{p}^C \rightarrow (P(i) = b)$ or $p_{j-1}^M \rightarrow (P(i) = b)$ are valid, but both are satisfiable. Then assign $o = v_j^C$ and $p_j^M = p_{j-1}^M \wedge (P(i) = v_j^C)$. Note that the operational semantics gave $p_j^C = \tilde{p}^C \wedge (P(i) = v_j^C)$.

(4c) $\tilde{p}^C \rightarrow (P(i) = b)$ is valid, but neither $p_{j-1}^M \rightarrow (P(i) = b)$ or $p_{j-1}^M \rightarrow (P(i) = \bar{b})$ are valid. Assign $p_j^M = p_{j-1}^M \wedge (P(i) = b)$ and v_j^M the value b.

(4d) $\tilde{p}^C \rightarrow (P(i) = b)$ valid and $p_{j-1}^M \rightarrow (P(i) = \bar{b})$ is valid. This case cannot occur by using the same argument in (1).

(4c) Neither $\tilde{p}^C \rightarrow (P(i) = b)$ or $\tilde{p}^C \rightarrow (P(i) = \bar{b})$ is valid, but $p_{j-1}^M \rightarrow (P(i) = b)$ is valid. This case cannot occur by using the same argument in (1).

(5) g_j^M is a write of a value d and location a to a memory R_{j-1}^M. First, set $p_j^M = p_{j-1}^M$. By inductive assumption g_j^C also wrote the value d to a location a to a corresponding memory R_{j-1}^C. Also, by induction assumption, the ICS terms (both address and values) in R_{j-1}^M are syntactically equivalent to those in R_{j-1}^C. For each address a' in R_j^C for which $p_j^C \rightarrow a \neq a'$ is valid, set $p_j^M = p_j^M \wedge (a \neq a')$ if $p_j^M \rightarrow (a \neq a')$ is not valid (as with case (4), no other possibilities are allowed). There is one address a' in R_j^C for which $p_j^C \rightarrow (a = a')$. Let $R_j^M = R_{j-1}^M$. If a' does not exist in R_j^M then create it. Write d to location a'. Clearly both properties hold.

(6) g_j^M is a read. This case is similar to (5).

If both properties hold for the state s_i^M, then they must also hold for s_{i+1}^M. This is because the values assigned to the present state variables at s_{i+1}^M (s_{i+1}^C) are the same as the values assigned to the next state functions at s_i^M (s_i^C).

Finally, it needs to be shown that $x^C \approx x^M$. By property 1, the assignment of all finite selection variables is the same, thus part (1) of definition 2.6.1.1 is satisfied. If v_j^M is an integer selection variable in M and $v_j^C \in V_M^C$ is the corresponding selection variable in C, then $v_j^C = v_j^M$ is valid by property (2). Otherwise, the selection variable in C is some variable $\tilde{v}^C \in V_N^C$, and there is a gate in C of the form $z_j := if\left(\tilde{v}^C = v_j^C, \tilde{v}^C\right)$. Thus, if C assigns $\tilde{v}^C = \tilde{\alpha}$ and $v_j^C = \alpha$ at state s_i^C, then $\alpha = \tilde{\alpha}|_{p_{i+1}^C}$ is valid. As a result of this and property (2), part (2) of definition 2.6.1.1 also satisfied.
(QED)

2.6.2 Comparison to Language Definition Given in *[HB95]*

In *[HB95]*, languages, and operations on languages, are defined in terms of concrete interpretations. A ***concrete language*** of a symbolic language L, for a given interpretation I, is denoted by $L_C(L, I)$ and is obtained by giving a concrete interpretation to all constants, uninterpreted functions and predicates for each string $x \in L$. Given two languages L and L', $L \subseteq L'$ is only true if for all possible interpretations, I, $L_C(L, I) \subseteq L_C(L', I)$. This definition is problematic because the semantics of ICS Models allows for fair runs to have no concrete interpretations. Determining if a fair run has a concrete interpretation is still an open problem. Thus, the language and automata definitions given in 2.5.1 were redefined such that they would not be based on the existence of such interpretations. It is conjectured, however, that for models where all fair runs have concrete interpretations, the definition given in *[HB95]* is equivalent to the definition in section 2.5.1.

2.6.3 Property Automata and Language Containment

M can be verified using the language containment paradigm. Here, a property P is represented using an automaton, A_P. Checking $L(M) \subseteq L(A_P)$ for a model M can be performed by first complementing A_P and then checking for ***language emptiness***, $L(M) \cap \overline{L(A_P)} = \varnothing$. Unfortunately, there are no known technique that can always find the complement of any language in the class of all symbolic ICS Languages. Indeed, there is no result that shows that this problem is even decidable. *[HB95]*, however, defined a class of automata called ***property automata***, whose symbolic langauge can be complemented and can be used to describe useful properties of ICS systems. Here, a property P is represented using a property automaton, A_P, which is a complete edge-Rabin automaton. A_P can have integer constant creators but all state variables are finite. The only operations allowed on integer variables are interpreted predicates. *[HB95]* showed that property automata can be complemented by first replacing the integer predicates with binary variables, performing standard edge-Rabin automata complementation, and finally replacing the binary variables that were introduced with the corresponding integer predicates that were in the original automaton.

2.6.4 Automatic Abstraction of ICS Models

Depending on the types of gates and predicates in an ICS Model M and the type of property P specified, the integer datapath can be automatically replaced with a datapath circuit that only has finite variables. This technique is a type of ***automatic abstraction***. Once M has been abstracted, traditional automatic verification techniques can be used to verify that P holds. Abstractions can be classified into there different categories. ***Exact abstractions*** are those in which if the property holds on the abstracted model, if and only if it also holds in M. Clearly, this is the best type of abstraction to have, since no accuracy in verification is sacrificed. A ***conservative abstraction*** is one in which if the property passes when verification is performed on the abstracted system, then the property holds for M. However, if the property does not hold in the abstracted system, then no claims can be made about the correctness of M. Thus, it is possible for verification on the abstracted system to return ***false negatives***. False negatives are bugs reported by the verification tool that are not real bugs in M. An ***aggressive abstraction*** is one in which if the property fails when verification is performed in the abstracted system, then the property also does not hold for M (but not vise-versa). In this case, it is possible to produce ***false positives***. Here, verification incorrectly reports that the design has passed a property. If exact abstractions of M cannot be found, conservative abstractions may be desirable. This is particularly true for safety critical situations where it is important to insure that M has no bugs with respect to P. Aggressive abstractions are useful if one simply wants to find real bugs in a design, and thus is willing to trade coverage for greater efficiency in verification. They may be necessary because formal verification may blow up, in which case an aggressive abstraction may give better coverage than simulation.

A particular type of automatic abstraction technique, called ***finite instantiations***, has been used successfully for abstracting certain classes of ICS Model.

Definition 2.6.4.1 Let a model M not have any uninterpreted functions or uninterpreted predicates. The k-

ary finite instantiations of M, denoted M_k is formed by replacing all integer variables with variables that can take a range of values from $0, \ldots, k-1$.

Chapter 3 will discuss finite Instantiations of ICS Models in more detail. Table 1, however, lists different automatic abstraction techniques for classes of ICS Models that have been published in the literature *[HB95, HIKB96, HDB97]*. The table columns include the name of the class, the datapath functions allowed in the class, the types of predicates allowed on data, the types of properties that can be verified and the type of abstraction technique that can be used.

Table 1: Abstraction Techniques for ICS Models

Name	Datapath Integer Functions	Datapath Predicates	Property to be verified	Abstraction Technique
DIC	DM, CC	none	General LTL properties built from $x = y$ *[HDB97]*	FI (EA)
DSSC	DM, CC	$x = y$, $x = c_l, x < d_j$, $(x\ mod\ m_k) < r_k$ $(x\ mod\ m_k) = r_k$	PA	FI (EA)
DSSC	CC	$x = y, x < y$ $x = c_l, x < d_j$, $(x\ mod\ m_k) < r_k$ $(x\ mod\ m_k) = r_k$	PA	FI (EA)
DCC	DM, CC, UF	none	$x = term$	Sub-formula decomposition/exact abstraction
DCC	DM, CC, UF	none	if $b = 1$ then $x = y$	Linear Expansion/ aggressive abstraction

The following terms are used in the table above:

DM	data movement gates
CC	constant creators
FI	finite instantiations
UF	uninterpreted functions
EA	exact abstraction
PA	property automata
DIC	data insensitive controller
DSSC	data semi-sensitive controller
DCC	data computation controller

Chapter 3

Finite Instantiations: Improving the Bounds.

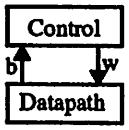
3.1 Introduction

Depending on the type of the system to be verified, verification can be performed by first replacing integer variables in the datapath with finite variables with ranges as small as only a few values. This type of automatic abstraction technique for ICS models is called ***finite instantiations***. Once a system is finitely instantiated, traditional state exploration algorithms can be employed to perform reachability analysis. A property of the system can be verified by either language containment *[HSIS94]* or model checking *[BCMD90, VIS96]*. There are three primary advantages of using finite instantiations. First, the infinite state space of the system can be abstracted (with respect to the property to be verified) to an equivalent finite state space. Second, since many finite instantiations produce exact abstractions, no accuracy is sacrificed when verification is performed on the instantiated model. Finally, unlike ICS symbolic simulation, finite instantiations allow traditional verification engines to be used to perform reachability analysis and language emptiness checking. Thus, previously well studied techniques and software, developed for verification of C/S models, can be leveraged when performing the verification of systems modeled with ICS using finite instantiations.

In this chapter, three new results are presented which give bounds on the minimum number of values needed in the datapath to accurately verify certain types of ICS Models. In Section 3.2, it is shown that for Data Semi-Sensitive Controllers (DSSC's), where only predicates of the form $x = y$ are used, the number of values needed in the integer datapath can be reduced significantly from what was earlier reported *[HB95]*. Section 3.3 introduces a new abstraction technique, called ***data shifting derivations***, that allows the bounds given in Section 3.2 to be reduced even further. Finally, in Section 3.4, a previously open problem is solved by showing that verification of DSSC's with predicate $x < y$ is decidable, and an exact data shifting derivation exists. This result is important because in *[HB95]* it was proved that ***no*** finite instantiations exist for this type of controller. Proofs and lower bounds are given for the approaches in each section.

3.2 Bound Reductions for DSSC's with $x = y$ Using Finite Instantiations

A ***Data Comparison Controller***, as shown in Figure 3.2.1, is a DSSC that can move integer data around in the datapath and perform equality comparisons on such data. The datapath can contain integer variables, integer latches, data movement gates and equality predicates. Communication from the datapath to the control is performed via integer equality predicates. Communication from the control to the datapath is performed via finite control signals fed to integer data movement gates. Previously, it was shown that for these types of controllers, n integer values were needed to verify language emptiness, where n is the number of integer variables in the datapath. In this section, we prove that this bound can be reduced further to k values, where k is the number of constant creators and integer latches in the datapath. The intuition for this reduction can be seen by making the following observations. First, the control is only sensitive to the relationship between integer variables in the datapath and not to the exact values these integer variables may take. For a given set of assignments to integer variables, the control can only distinguish between assignments where all integer values are distinct, assignments where all integer values are the same, or any other combination of equal/non equal predicate values applied to integer variables. Second, at any given state, there can be only a finite number of integer values in the datapath, since there are only a finite number of integer variables in the model. Finally, for data comparison controllers, the datapath can move data around but it cannot transform it (there are no interpreted or uninterpreted integer functions in the datapath), so the number of latches and constant creators dictate how may integer values can exist in the datapath.



A Data Comparison Controller is a DSSC with predicate $x = y$. The datapath contains only data movement gates, equality predicates and integer latches.

Figure 3.2.1: A Data Comparison Controllers.

A bound can be obtained by noting that control behavior is preserved as long as the number of values in the instantiated system is equal to, or greater than, the maximum number of distinct integer values that can be in the datapath at any given time. By proving that this number is at most k, the correctness of performing verification using k-ary finite instantiations follows.

Lemma 3.2.1 Let M be a data comparison controller, with k integer latches and constant creators. No more than k distinct values can exist in the datapath at any given time.

Proof The only gates in the datapath, other than latches or constant creators, are data movement gates. If an integer variable, x, is the output of a data movement gate, it must, by definition, be equal to one of its integer inputs. By using the operational semantics of ICS models, the value of x can be traced back to its source: either a constant creator or an integer latch in its transitive fanin. Thus, the value of all integer variables must be equal to a constant creator or an integer latch *(QED).*

Theorem 3.2.1 Let M be a data comparison controller with k integer latches and constant creators, and M_k its k-ary instantiation. Then $L(M) = \emptyset$ if and only if $L(M_k) = \emptyset$.

Proof The proof follows directly from lemma 4.1 in *[HB95]*. The bound in the proof for lemma 4.1 is equal to the number of distinct values in the datapath at any given time. By lemma 3.1, this values is equal to k *(QED)*.

Similar bound reductions can be made for lemmas 4.4, 4.5, 4.6 and 4.7 in *[HB95]*.

3.3 Bounds for DSSC's with $x = y$ using data shifting derivations

The finite instantiation techniques used in Section 3.2 and reported in *[HB95]*are based on the fact that there can only be a distinct number of integer values in the datapath at any given time. In this section, a new automatic abstraction technique is introduced, called data shifting derivations, that still requires k values for non-state integer variables as in Section 3.2, but only r of these values for integer latch variables, where r is the number of latches in the datapath. This optimization can be performed by noting that at each state, c new values can be introduced into the system (where c is the number of constant creators). Since it is possible that all of these new values are distinct from what is stored in the latches, $k = r + c$ values are needed in the datapath as discussed in 3.2. However, when the system performs a transition from a state s_i to a state s_{i+1}, only r values will be assigned to the next state variables in state s_i. The range of the integer latches can therefore be reduced to take values between 0 and $r - 1$ by "packing" the data given at the output of the next state functions. In order to facilitate this data shifting, the transition relation of the datapath has to be modified. The advantage of this approach is that it can reduce the number of reachable states in the abstracted model, since each datapath state variables has a range of r instead of k.

Data shifting derivations are performed by inserting a data shifting function β_{eq} between the next state functions in the datapath and their corresponding datapath latches. The integer latches variables can then be given a range of r. All other datapath variables are given a range of k.

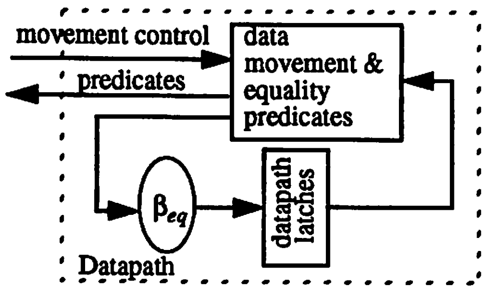
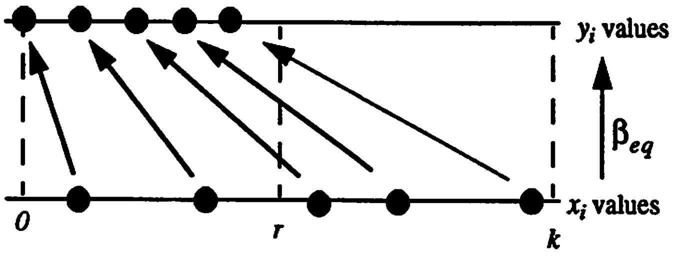


Figure 3.3.1: Data shifting derivations

The reductions that data shifting derivations provide can be helpful when performing explicit state exploration. This is because the complexity of this method is a function of the number of states in the system. However, the advantage of this approach for BDD-based reachability techniques is less clear. Since the transition relation of the datapath has to be modified in the derived model, the size and complexity of the BDDs needed to represent this relation may grow. On the other hand, the reduction in range of the datapath state variables may reduce the size and complexity of the BDDs for the reachable sets. Experiments need to be performed to determine if and when these abstractions are useful.

The data shifting derivation technique is first created by constructing a k-ary finite instantiation of M, M_k, as described in the Section 3.2. Next, as shown in Figure 3.3.1, a data shifting function, β_{eq}, is inserted between the datapath next state functions in M_k and their corresponding latches. Finally, the domains of the datapath latches are reduced to r. The formal definition of this procedure is given below.

Definition 3.3.1 β_{eq} is any multi-variable output function $\beta_{eq}:k^r \to r^r$ that is defined such that the following two conditions hold. First, the values that β_{eq} assigns to the integer latch variables, $Y = \{y_0, \ldots, y_{r-1}\}$, must always be less than r. Thus, given a set of input variables $X = \{x_0, \ldots, x_{r-1}\}$, where $\forall i, x_i < k$, and $Y = \beta_{eq}(X)$ then $\forall i, y_i < r$. Second, equality must be preserved between all input variables and output variables: $\forall i, j, y_i = y_j$ if and only if $x_i = x_j$. Figure 3.3.2 presents a graphical representation of a β_{eq} shifting data.



β_{eq} shifts data values assigned to the variables x_i, $i < r$ (where the x_i variables have been assigned at values between 0 and k) and places the results in the y_i variable's (where each variable is assigned values between 0 and r). β_{eq} preserves equality between the set of x_i variables and corresponding y_i variables. Note that order between values does not have to be preserved, i.e. $x_i < x_j$ does not imply that $y_i < y_j$.

Figure 3.3.2: a β_{eq} function.

Formally, a data shifting derivation is defined as follows:

Definition 3.3.2 A data shifting derivation, $M_{\beta_{eq}}$, can be constructed from M, a DSSC with predicate only of the form $x = y$, as follows:

(1) Create M_k.

(2) Insert a data shifting function, β_{eq}, between the next state functions in the datapath and their corresponding latches.

(3) Set the range of the datapath latch variables to take on any value from 0 to $r-1$.

(4) Use the exact same fairness constraints as given in M, since they are placed only on the control variables.

(5) The initial states of $M_{\beta_{eq}}$ are computed by taking each initial state in M_k and applying β_{eq} to the datapath variables assignments such that each has a value less than r. The control component of the initial states remain the same.

Theorem 3.3.1 proves that this data shifting derivation provides an exact abstraction when checking for language emptiness.

Theorem 3.3.1 Let M be a data comparison controller with r integer latches, c constant creators and $M_{\beta_{eq}}$ its data shifting derivation. Then, $L\left(M_{\beta_{eq}}\right) = \varnothing$ if and only if $L(M) = \varnothing$.

Proof Let's assume that $L(M) \neq \varnothing$, such that there exists a fair run $e = s_1, p_1, a_1, s_2, p_2, a_2, \ldots$ in M where each s_i, p_i, a_i triple is an assignment to the latch variables, integer predicate variables and integer non-state variables, respectively. It will be proved that there exists a fair run $\tilde{e} = \tilde{s}_1, \tilde{a}_1, \tilde{s}_2, \tilde{a}_2, \ldots$ in $M_{\beta_{eq}}$ where the following two conditions hold for all i.

(1) The assignment of values to the control (finite) variables are the same for both s_i, p_i, a_i and $\tilde{s}_i, \tilde{a}_i$. Note that there is no $\tilde{p}_i$ in $M_{\beta_{eq}}$, since it has no explicit integer predicate variables. Since the fairness constraints of M and $M_{\beta_{eq}}$ are the same, by showing that the assignments to the control variables are the same for both e and $\tilde{e}$, then $\tilde{e}$ will be proven fair.

(2) The assignment of the datapath variables in $\tilde{e}$ is made such that equality between assignments is maintained. Let n be the number of datapath variables and $\alpha_{0,i}, \ldots, \alpha_{n-1,i}, \tilde{\alpha}_{0,i}, \ldots, \tilde{\alpha}_{n-1,i}$ be assignments to datapath variables in s_i, a_i and $\tilde{s}_i, \tilde{a}_i$, respectively. In addition, let $\tilde{\alpha}_{n,i}, \ldots, \tilde{\alpha}_{n+r,i}$ be the assignment to the new variables introduced by β_{eq}. Then for all $k, j < n$, if $\tilde{\alpha}_{k,i} = \tilde{\alpha}_{j,i}$ then $\alpha_{k,i} = \alpha_{j,i}\big|_{p_i}$.

The existence of $\tilde{e}$ can be proved as follows:

For $i = 1$, we have $\tilde{s}_1 = \beta_{eq}(s_1)$, which by the definition of β_{eq} implies the above two conditions hold for s_1 and $\tilde{s}_1$.

For any i, given a state $\tilde{s}_i$ that satisfies the above two properties, there exist an $\tilde{a}_i$ such that all variables (let's exclude, for now, those created by the introduction of β_{eq}) also satisfy the above two properties. This can be shown by induction on the gate graph associated with $M_{\beta_{eq}}$.

Assume that there are m gates and that the hypothesis holds for variables of gates g_k for $k < l$. Note that gates associated with β_{eq} will come last in any topological sort of the gate graph of $M_{\beta_{eq}}$. Thus, assume also, that g_l is not one of those gates. We have the following cases.

(1) g_l is a finite table in the control. Using the induction hypothesis and property 1, its output is the same value that was given to it in a_i.

(2) g_l is an equality predicate. By property 2 and the induction hypothesis, its output will have the same value as in a_i.

(3) g_l is a datapath input. First, let's define a predicate, U_e, associated with the run e, that gives the intersection of all the predicates in e, i.e. $U_e = \prod_{\forall i} p_i \in e$. By the induction hypothesis and property 1, if a_i assigns g_l the value $\alpha_{l,i}$ and there is a datapath latch value or some other previously processed gate g_k, that was given the value $\alpha_{k,i}$, where $\alpha_{l,i} = \alpha_{k,i}|_{U_e}$, then give to g_l the value $\tilde{\alpha}_{k,i}$. Note that only k such values are needed.

(4) g_l is a datapath data movement gate. Assign it a value equivalent to one of its datapath inputs. By the induction hypothesis and property 1, this value corresponds to the one given to g_l in a_i.

If the two properties hold for state $\tilde{s}_i$, then they must also hold for the state $\tilde{s}_{i+1}$. This is because:

(1) the values assigned to the control next state functions at $\tilde{s}_i$ are the same as the values assigned to the con-

trol present state values at $\tilde{s}_{i+1}$. Thus, property 1 will be preserved.

(2) The values assigned to the datapath next state functions in $M_{\beta_{eq}}$ (the output of β_{eq}) are assigned such that property 2 is preserved. This implies that if $\alpha_{u,i}$, $\alpha_{v,i}$ are assignments to the next state functions in M, and $\tilde{\alpha}_{w,i}$, $\tilde{\alpha}_{x,i}$ are assignments to the corresponding next state functions in $M_{\beta_{eq}}$, then $\alpha_{u,i} = \alpha_{v,i}\big|_{p_i}$ if and only if $\tilde{\alpha}_{w,i} = \tilde{\alpha}_{x,i}$. Note that only r values are needed, since there are only r state variables.

The reverse direction of the proof is similar and not shown here.

(QED)

3.4 Bounds for DSSC's with $x<y$ using data shifting derivations.

In this section, it is proven that checking for language emptiness of DSSC's that use predicates of the form $x = y$ and $x<y$ is decidable and that an exact data shifting derivations exist. In *[HB95]*, it was shown that there exists such a DSSC controller, M, with a non-empty language, where all its finite instantiations had an empty language. M only accepts runs that consist of infinite sequences of increasing integers. Since the finite instantiation associated with this machine can only accept runs consisting of a finite sequence of increasing values, M_k cannot preserve all control behavior of M and hence no exact finite instantiations exists. When dealing with DSSC's with $x = y$ and $x<y$ predicates, note that at each state, c new values can be introduced into the system (where c is the number of constant creators). Each new value, can be 1) equal to one of the values stored in the latches, 2) between two values stored in the latches, 3) less than all values stored in the latches, or 4) greater than all values stored in the latches.

Intuitively, our approach is similar to that presented in Section 3.3. Enough values are allowed in the datapath such that each of the above cases can occur. Data is then shifted at the next state functions so that these cases can occur again in the next state. Since we are using the predicate $x<y$, a new function, β_{lt}, is introduced that guarantees that there are (1) always at least c values in between any two distinct assignments to latch variables, (2) there are at least c values greater than any assignment and (3) there are at least c less than any assignment. Thus, the total number of values needed in the datapath are $t = c(r+1) + r$ (where r is the number of datapath latches). Unlike the Section 3.3, all datapath variables are instantiated with the same number of values (t). The advantage of this approach is, of course, that it allows an exact abstraction to be performed on DSSC's with predicates of the form $x<y$ where finite instantiations cannot be used.

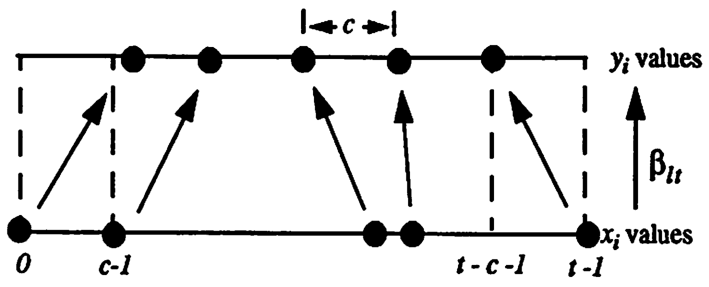
Definition 3.3.1 β_{lt} is any multi-variable output function $\beta_{lt}: t^r \rightarrow t^r$ that is defined such that the following four conditions hold.

(1) The values that β_{lt} assigns to the integer latch variables, $Y = \{y_0, \ldots, y_{r-1}\}$, must always be greater than $c-1$ and less than $t-c$. Thus, given a set of input variables $X = \{x_0, \ldots, x_{r-1}\}$, where $\forall i, x_i < t$, and $Y = \beta_{lt}(X)$ then $\forall i, c-1 < y_i < t-c$.

(2) Equality must be preserved between all input and output variables: $\forall i, j$, $y_i = y_j$ if and only if $x_i = x_j$.

(3) Order must be preserved between all input and output variables: $\forall i, j$, $y_i < y_j$ if and only if $x_i < x_j$.

(4) There should always be at least c values between any two distinct assignments to output variables: $\forall i, j$, $y_i < y_j$ if and only if $y_j - y_i > c$. Figure 3.4.1 contains an example of β_{lt} shifting values.



β_{lt} shifts data values assigned to the variables x_i, $i < r$ (where the x_i variables have been assigned values between 0 and $t-1$) and places the results in the y_i variables (where there is a distance of at least c between all distinct assignments). β_{lt} preserves equality and order between the set of x_i variables and corresponding y_i variables

Figure 3.4.1: the β_{lt} function.

Formally, data shifting derivations for DSSC's with predicates $x < y$ and $x = y$ are defined as follows:

Definition 3.4.2 A data shifting derivation, $M_{\beta_{lt}}$, can be constructed from M, a DSSC with predicates of the type $x < y$ and $x = y$, as follows:

(1) Create M_{lt}.

(2) Insert a data shifting function β_{lt} between the next state functions in the datapath and their corresponding latches.

(3) Use the same fairness constraints as in M, since they are placed only on the control variables.

(4) The initial states of $M_{\beta_{lt}}$ are computed by taking each initial state in M and applying

β_{lt} to it.

(4) The initial states of $M_{\beta_{lt}}$ are computed by taking each initial state in M_{lt} and applying β_{lt} to the datapath variables. The control component of the initial states remains the same.

Theorem 3.4.1 proves that language emptiness checking on DSSC with predicates $x < y$ and $x = y$ is decidable and that data shifting derivations will always provide an exact abstraction when performing this check.

Theorem 3.4.1 Let M be a data comparison controller with predicates $x < y$, $x = y$, r integer latches and c constant creators. Let $t = c(r + 1) + r$ and $M_{\beta_{lt}}$ its data shifting derivation. Then, $L(M_{\beta_{lt}}) = \varnothing$ if and only if $L(M) = \varnothing$.

Proof Let's assume that $L(M) \neq \varnothing$ and there exists a symbolic fair run $e = s_1, p_1, a_1, s_2, p_2, a_2, \ldots$ in M where each s_i, p_i, a_i triple is an assignment to the latch variables, predicate variables and non-state variables, respectively. It will be proved that there exists a fair run $\tilde{e} = \tilde{s}_1, \tilde{a}_1, \tilde{s}_2, \tilde{a}_2, \ldots$ in $M_{\beta_{lt}}$ where the following two conditions hold for all i.

(1) The assignment of values to the control variables are the same for both s_i, p_i, a_i and $\tilde{s}_i, \tilde{a}_i$. Note that there is no $\tilde{p}_i$ in $M_{\beta_{lt}}$, since it has no explicit predicate variables. Since the fairness constraints of M and $M_{\beta_{lt}}$ are the same by showing that the control variables in e and $\tilde{e}$ are the same, the run $\tilde{e}$ will be proven fair.

(2) The assignment of the datapath variables in $\tilde{e}$ is made in such a way that equality and order between assignments is maintained. Let n be the number of datapath variables and $\alpha_{0,i}, \ldots, \alpha_{n-1,i}$, $\tilde{\alpha}_{0,i}, \ldots, \tilde{\alpha}_{n-1,i}$ be datapath assignments to s_i, a_i and $\tilde{s}_i, \tilde{a}_i$, respectively. Let $\tilde{\alpha}_{n,i}, \ldots, \tilde{\alpha}_{n+r,i}$ be the assignment to the new variables introduced by β_{lt}. Then for all $k, j < n$, if $\alpha_{k,i} = \alpha_{j,i}\big|_{p_i}$ is valid then $\tilde{\alpha}_{k,i} = \tilde{\alpha}_{j,i}$. Also if $\alpha_{k,i} < \alpha_{j,i}\big|_{p_i}$ is valid then $\tilde{\alpha}_{k,i} < \tilde{\alpha}_{j,i}$.

To prove the existence of $\tilde{e}$, observe that the predicates in e may only imply a partial order on the symbolic values assigned to datapath variables. The run $\tilde{e}$, however, can only produce a total order on assignments to the datapath variables, since only concrete values in $M_{\beta_{lt}}$ are used to represent datapath behavior. Thus, it must be shown that assuming a total order on symbolic values in the datapath will not change the assignment of any values to variables in the run e. To show this, define a predicate, U_e, that gives the intersection of all the predicates in e, i.e. $U_e = \prod_{\forall i} p_i \in e$. Next, define another predicate, $\hat{U}_e$, that is constructed as follows:

(1) Let $\hat{U}_e = U_e$.

(2) For each α_1, α_2 that has been assigned to integer variables in e, if $\hat{U}_e \rightarrow ((\alpha_1 < \alpha_2) \vee (\alpha_1 = \alpha_2) \vee (\alpha_1 > \alpha_2))$ is not valid, then arbitrarily choose one of these, say $\alpha_1 = \alpha_2$, and let $\hat{U}_e = \hat{U}_e \wedge (\alpha_1 = \alpha_2)$.

Note that $\hat{U}_e$ represents a total order on all the ICS terms assigned to variables in the run e. Let's now assume that for some state s_i, p_i in e that $\hat{p}_i = \hat{U}_e$. Also, let $\hat{s}_j, \hat{a}_j$ be the assignment of state and non state variables for all successor states j $(i \le j)$. For all assignments $\hat{s}_j, \hat{a}_j$ it must be true that $\hat{s}_j = s_j$ and $\hat{a}_j = a_j$. This can be argued as follows. Assume that this is not true. This implies that at some state $\hat{s}_j$, a predicate gate $o = g(i)$ is executed such that $p_j \rightarrow o = b$ but $\hat{p}_j \rightarrow o = \bar{b}$ ($b = 0$ or $b = 1$). Without loss of generality, assume that this gate executes the predicate $\alpha_1 < \alpha_2$. Therefore, $p_j \rightarrow \alpha_1 < \alpha_2$ is valid. However, if this were true, it would imply that $U_e \rightarrow \alpha_1 < \alpha_2$ is valid. Since $U_e \rightarrow p$ is valid implies that $\hat{U}_e \rightarrow p$ is valid for all predicates p, we have a contradiction.

Let's now define the run $\hat{e} = \hat{s}_1, \hat{a}_1, \hat{p}_1, \hat{s}_2, \hat{a}_2, \hat{p}_2, \ldots$, such that for all i, $\hat{s}_i = s_i$, $\hat{a}_i = a_i$ and $\hat{p}_i = \hat{U}_e$. Observe that $\hat{e}$ preserves both properties 1 and 2 with respect to e.

The existence of $\tilde{e}$ can now be shown by performing induction on the length of the run $\hat{e}$ and showing that $\tilde{e}$ preserves both properties 1 and 2.

For $i = 1$, we have $\tilde{s}_1 = \beta_{lt}(\hat{s}_1)$, which by the definition of β_{lt} implies the above two conditions hold of s_1 and $\hat{s}_1$.

For any i, given a state $\tilde{s}_i$ that satisfies the above two properties, there exist an $\tilde{a}_i$ such that all variables (let's exclude, for now, those created by the introduction of β_{lt}) also satisfy the above two properties. This can be shown by induction on the gate graph associated with $M_{\beta_{lt}}$.

Assume that they are m gates and that the hypothesis holds for variables of gates g_k for $k < l$. Note that gates associated with β_{lt} will come last in any topological sort of the gate graph. Thus, assume also, that g_l is not one of those gates. We have the following cases.

(1)g_l is a finite table in the control. Using the induction hypothesis and property 1, it is assigned the same value that was given to it in $\hat{a}_i$.

(2)g_l is an equality predicate. By the induction hypothesis and property 2, its output will have the same value as in $\hat{a}_i$.

(3)g_l is a less than predicate. B y the induction hypothesis and property 2, its output will have the same value as in $\hat{a}_i$.

(4)g_l is a datapath constant creator. Assume that z_l is the number of datapath constant creator assignments $\alpha_{d,i}$ given by $\hat{a}_i$ such that $\alpha_{d,i} < \alpha_{l,i}\big|_{\hat{U}_e}$ and z_u is the number of datapath constant creator assignments $\alpha_{d,i}$ such that $\alpha_{l,i} < \alpha_{d,i}\big|_{\hat{U}_e}$. Four cases for g_l may occur.

(4a) If $\hat{a}_i$ assigns g_l the value $\alpha_{l,i}$, and there is a datapath latch value or some other previously processed gate g_k, that was given the value $\alpha_{k,i}$, where $\alpha_{l,i} = \alpha_{k,i}\big|_{\hat{U}_e}$, then give to g_l the value $\tilde{\alpha}_{k,i}$.

(4b) There are two datapath latch values, $\alpha_{j,i}$ and $\alpha_{k,i}$, assigned in $\hat{s}_i$ such that $\alpha_{j,i}$ is the greatest value where $\alpha_{j,i} < \alpha_{l,i}\big|_{\hat{U}_e}$ is valid and $\alpha_{k,i}$ is the least value where $\alpha_{l,i} < \alpha_{k,i}\big|_{\hat{U}_e}$ is valid. Assign $\tilde{\alpha}_{l,i}$ any value where $\tilde{\alpha}_{j,i} + z_l < \tilde{\alpha}_{l,i} < \tilde{\alpha}_{k,i} - z_u$ holds. Note that only c such values are needed between $\tilde{\alpha}_{j,i}$ and $\tilde{\alpha}_{k,i}$.

(4c) There is no datapath latch value, $\alpha_{j,i}$,where $\alpha_{j,i} < \alpha_{l,i}\big|_{\hat{U}_e}$, but there exist at least one datapath latch value $\alpha_{k,i}$ assigned in $\hat{s}_i$ such that $\alpha_{l,i} < \alpha_{k,i}\big|_{\hat{U}_e}$ is valid. Let $\alpha_{k,i}$ be the smallest such value where this is true. Assign $\tilde{\alpha}_{l,i}$ a value such that $z_l - 1 < \tilde{\alpha}_{l,i} < c - z_u$. Note that only c such cases may occur.

(4d) There is no datapath latch value $\alpha_{j,i}$ where $\alpha_{l,i} < \alpha_{j,i}\big|_{\hat{v}_e}$ is valid, but there exist a datapath latch value $\alpha_{k,i}$ assigned in $\hat{s}_i$ such that $\alpha_{k,i} < \alpha_{l,i}\big|_{\hat{v}_e}$ is valid. Let $\alpha_{k,i}$ be the largest such value where this is true. Give $\tilde{\alpha}_{l,i}$ a value such that $t - c + z_l - 1 < \tilde{\alpha}_{l,i} < t - z_u$. Again note that only c such cases may occur.

It is thus evident, that only $t = c(r+1) + r$ values are needed.

(5) g_l is a datapath data movement gate. Assign g_l the value equivalent to one of its datapath inputs. By the induction hypothesis and property 1, this value corresponds to the one given to g_l in $\hat{a}_i$.

If the two properties hold for the state $\tilde{s}_i$, then they must also hold for the state $\tilde{s}_{i+1}$. This is because:

(1) the values assigned to the control next state functions at $\tilde{s}_i$ are the same as the values assigned to the control present state values at $\tilde{s}_{i+1}$. Thus, property 1 will be preserved.

(2) The values assigned to the datapath next state functions in $M_{\beta_{lt}}$ (the output of β_{lt}) are assigned such that property 2 is preserved. This implies that if $\alpha_{u,i}, \alpha_{v,i}$ are assignments to the next state functions in M, and $\tilde{\alpha}_{w,i}, \tilde{\alpha}_{x,i}$ are assignments to the corresponding next state functions in $M_{\beta_{lt}}$, then if $\alpha_{u,i} = \alpha_{v,i}\big|_{\hat{p}_i}$ then $\tilde{\alpha}_{w,i} = \tilde{\alpha}_{x,i}$ and if $\alpha_{u,i} < \alpha_{v,i}\big|_{\hat{p}_i}$ then $\tilde{\alpha}_{w,i} < \tilde{\alpha}_{x,i}$. Note that only t values are needed.

For completeness, the reverse will also be proved. Assume $L\left(M_{\beta_{lt}}\right) \neq \emptyset$, such that there exists a fair run $\tilde{e} = \tilde{s}_1, \tilde{a}_1, \tilde{s}_2, \tilde{a}_2, \ldots$ in $M_{\beta_{lt}}$ where each $\tilde{s}_i, \tilde{a}_i$ is an assignment to the latch variables and non-state variables, respectively. It will be proved that there exists a fair run $e = s_1, p_1, a_1, s_2, p_2, a_2, \ldots$ in M where the following two conditions hold for all i.

(1) The assignment of values to the control variables are the same for both s_i, p_i, a_i and $\tilde{s}_i, \tilde{a}_i$.

(2) The assignment of the datapath variables in s_i, p_i, a_i is made in such a way that equality and order between assignments is maintained. Let $\alpha_{0,i}, \ldots, \alpha_{n-1,i}$ and $\tilde{\alpha}_{0,i}, \ldots, \tilde{\alpha}_{n-1,i}$ be datapath assignments to s_i, a_i and $\tilde{s}_i, \tilde{a}_i$, respectively. Let $\tilde{\alpha}_{n,i}, \ldots, \tilde{\alpha}_{n+r,i}$ be the assignment to the new variables introduced by

β_{lt}. Then for all $k, j < n$, if $\tilde{\alpha}_{k,i} = \tilde{\alpha}_{j,i}$ then $\alpha_{k,i} = \alpha_{j,i}|_{p_i}$. Also if $\tilde{\alpha}_{k,i} < \tilde{\alpha}_{j,i}$ then

$\alpha_{k,i} < \alpha_{j,i}|_{p_i}$.

The existence of the run e can be proved as follows.

For $i = 1$, for all $k, j < n$, if $\tilde{\alpha}_{k,1} = \tilde{\alpha}_{j,1}$ then let $p_1 = p_1 \wedge (\alpha_{k,1} = \alpha_{j,1})$ (by definition of DSSCs, all assignments to latches are fresh constants). Also, if $\tilde{\alpha}_{k,1} < \tilde{\alpha}_{j,1}$, then let

$p_1 = p_1 \wedge (\alpha_{k,1} < \alpha_{j,1})$.

The resulting pair, s_i, p_i, is a valid initial state of M, since each initial state in $M_{\beta_{lt}}$ was constructed such that the above two constraints were valid.

For any i, given a state s_i, p_i satisfying the above two properties, there exist an a_i such that all variables also satisfy the above two properties. This can be shown by induction on the gate graph associated with M. Assume that there are m gates and that the hypothesis holds for variables of gates g_k for $k < l$. Also, let p_i' be the insertion of p_i and all the predicates created thus far during the symbolic execution of this gate graph. If g_l is

(1) a finite table in the control, then it is assigned the same value that was given to it in $\tilde{a}_i$, using the induction hypothesis and property 1.

(2) an equality predicate, then its output will have the same value as in $\tilde{a}_i$, by the induction hypothesis and property 2.

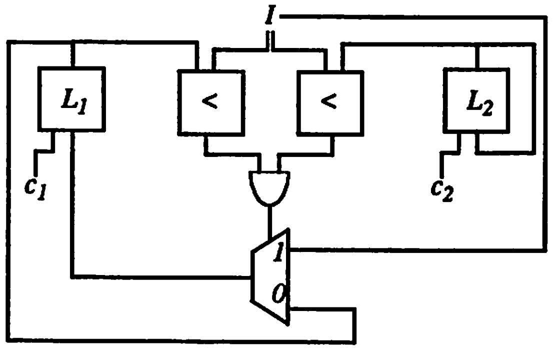
(3) a less than predicate, then its output will have the same value as in $\tilde{a}_i$ by the induction hypothesis and property.

(4)a datapath constant creator, then assign it a new fresh constant, c_j. If $\tilde{a}_i$ assigned it to a value equal to one of the variables processed so far (let's say a ICS term μ corresponds to that value), then let $p_i' = p_i' \wedge (c_j = \mu)$. If the value is distinct from all other terms processed thus far, then for each such term μ, let $p_i' = p_i' \wedge (c_j \neq \mu)$. If the value is less than all terms processed so far, then for all such terms, μ, assign $p_i' = p_i' \wedge (c_j < \mu)$. Similarly, if the value is greater than all term processed thus far, assign $p_i' = p_i' \wedge (c_j > \mu)$. If the value is distinct, but between two terms, μ_1 and μ_2, then assign

$p_i' = p_i' \wedge (\mu_1 < c_j) \wedge (c_j < \mu_2)$.

(5) a datapath data movement gate, then assign it a value equivalent to one of its datapath inputs. This value will be equivalent to the one corresponding assigned i a_i by the induction hypothesis and property 1.

If the two properties hold for the state s_i, p_i, then they must also hold for the state s_{i+1}, p_{i+1} (where $p_{i+1} = p_i'$ at end of the symbolic execution of the gate graph). This is because the values assigned to the control and datapath next state functions at s_i, p_i are the same as the values assigned to the control and datapath present state values at s_{i+1}, p_{i+1}. (QED)



This circuit takes a fresh constant given by input I and assigns it to latch L_1 if its value is between the terms stored in L_1 and L_2. Otherwise, the current value of L_1 will not change. This process is repeated at every state. A run consisting of an infinite sequence of increasing values that are greater than c_1 and less than c_2 is accepted by this circuit. This run, however, has no concrete interpretation, since between any two integers there can only be a finite number of values. Although this run will not be accepted by any finite instantiation of this circuit, it will be accepted by its data shifting derivation.

Figure 3.4.2: An ICS Circuit accepting runs with no concrete interpretation.

It is important to note that some ICS models accept runs that do not have *concrete interpretations [HIKB96]*. A concrete interpretation of the behavior of an ICS Model can be performed by first replacing all uninterpreted functions (if any) by interpreted functions, and then taking each fair run accepted by the system and replacing all ICS terms with their appropriate integer values. For example, consider the circuit in Figure 3.4.2. This ICS circuit takes two symbolic integers c_1 and c_2 and can accept an infinitely increasing sequence of symbolic integers that are between these two values. Such runs of this system do not have a concrete interpretations, since between any two concrete integers, there can only be a finite number of values in between. Its data shifting derivation preserves this behavior.

Chapter 4

Reachability Analysis of ICS Models

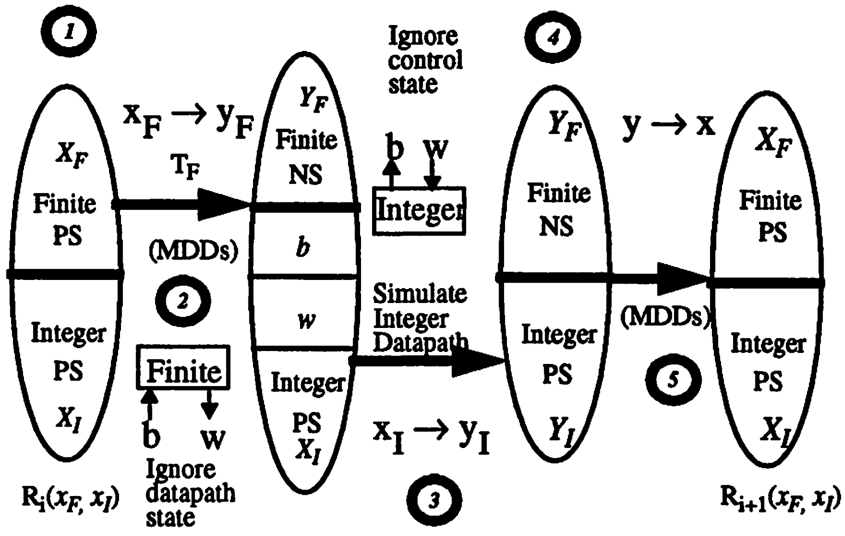
4.1 Introduction

For those cases where equivalent finite instantiations cannot be found, ICS reachability analysis can be used to generate the ICS state space. A traditional state exploration based verification tool can then be used to verify properties on the set of reachable states generated. Since some ICS models may have an infinite number of reachable states, ICS reachability analysis may not always terminate. In such cases, however, reachability can be performed for a finite number of steps (let's say n). Partial verification can then be used to show that no error trace up to length n exists. Since, in practice, most error traces are short, this may be useful for finding bugs. In addition, partial verification can provide a level of confidence in a design (higher coverage) that may not be available by regular simulation.

ICS reachability can also be used for performing processor verification. Here, one would like to verify that a pipelined microprocessor implements its instruction set architecture. Burch and Dill *[BD94]* presented a unique approach for solving this problem, and ICS reachability can be used to approximate their approach.

An overview of the ICS reachability algorithm can be described as follows. The set of all reachable states generated by ICS symbolic simulation can be represented using Multi-valued Decision Diagrams (MDDs) *[Kam90]*. To represent values (ICS terms) assigned to integer latch variables, they are encoded using a finite domain, i.e. a set of bits. This is possible, because, although the set of all ICS terms is infinite, the set of all ICS terms that have been assigned to latches after n steps of reachability is always finite. Transitions between ICS states are computed using a combination of implicit techniques for the control, and explicit techniques for the integer datapath and memory. Assume that b is the set of predicates going from the datapath to the control, and w is the set of control signals going to the datapath (both b and w are bit vectors). For the control portion of the design, an MDD is used to represent its transition relation, and the standard MDD image computation is used to get an approximation of the possible next control states. It is an over approximation since it is assumed that b and w are free inputs and outputs, respectively, of the control part. For each next state computed using the image computation, the corresponding (b, w) pair that made the transition possible is recorded. For the integer datapath, the next state is computed through a special form of symbolic simulation. Here, the integer datapath is simulated using each (b, w) pair and current state assignments (ICS terms) to integer latches, predicates, and memory. If the values for b and w are not consistent with the datapath, then the pair is invalid and thrown out. Finally, the next states in the datapath are combined with the appropriate next states in the control to form a composite set of next states of the machine. Note that by throwing out invalid (b, w) pairs, we have reduced the over approximation to an exact computation. This composite set of next states is again represented using an MDD. Figure 4.1.1 provides a graphical description of this process.

This chapter focuses on an analysis of the ICS reachability algorithm, the implementation of our first generation ICS reachability tool, and experimental results generated by the tool. The outline for the rest of this chapter is as follows. A detailed explanation of the ICS reachability algorithm is first given in Section 4.2. In Section 4.3, experimental results, produced by the tool, are presented. In particular, results are given for computing the set of reachable states on a simple design. A major bottleneck associated with this algorithm is also discussed. Finally, in Section 4.4, it is demonstrated how ICS reachability can be used for solving the problem of pipelined processor verification, by approximating the approach described in *[BD94]*. Experimental results are given for two simple pipeline microprocessors, and comparisons to other approaches are made.



ICS Reachability is described as follows: (1) the reachable states at step i are represented using an MDD. Assume that the variables x_F, y_F, x_I, and y_I are the present and next state variables of the finite control and integer datapath, respectively. Here, ICS terms assigned to integer latches, memories, and predicates are encoded into MDD variables. (2) The communication variables, (b, w), are assumed free and the transition relation, T_F, of the control (also represented as an MDD) is used to get the next control states, as well as their corresponding (b, w) pairs. (3) While ignoring the control, each integer state and (b, w) pair is simulated through the datapath. Invalid (b, w) pairs detected during simulation are thrown away. (4) New ICS terms created during simulation are re-encoded, and an MDD representing the composite control and datapath next states is created. (5) The next state MDD variables are replaced with present state MDD variables to get the reachable states at step $i + 1$.

Figure 4.1.1: ICS Reachability, Step by Step.

4.2 Reachability Analysis Algorithm

The algorithm presented in this section is based on the algorithm of *[HB95]*. Recall that an ICS state contains three components: assignments to latches, predicates and memories. In order to represent the set of reachable states as an MDD, three tables are used to encode ICS terms assigned to the datapath state variables. H_I is a table of all ICS terms which have been assigned to latches, and their corresponding MDD encoded values. An example is given in figure 4.2.2.

Expand as necessary

ICS term	MDD Encoding
f(a)	0
a + b	1
f(f(b))	2
a	3

The term table, H_I, stores each ICS term that has been previously assigned to an integer latch during symbolic simulation. In general, the number of all possible ICS terms is infinite. However, since the symbolic simulation algorithm can only can execute a finite number of steps, this table is always finite. The size of the table is expanded as more terms are assigned to latches.

Figure 4.2.1 The Term Table H_I

H_P is a table of predicates that have been enumerated thus far and their MDD encoding. Finally, H_M is a table of memories encountered in the states enumerated so far. Here, each memory is a table of pairs of ICS terms. Memories and predicates are encoded in a similar fashion to integer latch assignments in H_I, where each unique memory or predicate is given a unique finite value. Note that since these three tables are continually expanding, the range of the MDD variables representing these encodings also has to expand. The MDD representing the reachable state set consists of one MDD variable for each finite latch, one MDD variable for each integer latch that ranges over all encodings in H_I, one MDD variable p representing the state predicates, ranging over the encodings in H_P, and one variable m representing the state memory and ranging over all encodings in H_M.

In the following, assume that the variables x_F, y_F, x_I, and y_I are the present and next state variables of the control and datapath, respectively. The transition relation of the finite part, denoted $T_F(x_F, y_F, b, w)$, is computed in a fashion similar to C/S models: take the intersection of all finite tables in the control. The transition relation of the integer datapath, denoted $T_I(x_I, y_I, b, w)$, must be computed incrementally by the algorithm since it may consist of an infinite number of transitions. If $R_i(x_F, x_I)$ is the set of states reached after i steps, then

$$R_{i+1}(y_F, y_I) = \exists b \exists w \exists x_I \exists x_F (T_F(x_F, y_F, b, w) \wedge R_i(x_F, x_I) \wedge T_I(x_I, y_I, b, w))$$

By rearranging the terms it can equivalently be expressed as

$$R_{i+1}(y_F, y_I) = \exists b \exists w \exists x_I (\exists x_F (T_F(x_F, y_F, b, w) \wedge R_i(x_F, x_I)) \wedge T_I(x_I, y_I, b, w))$$

The symbolic simulation algorithm proceeds in a fashion similar to the operational semantics described in Chapter 2. First, the function $U_i(y_F, x_I, b, w) = \exists x_F (T_F(x_F, y_F, b, w) \wedge R_i(x_F, x_I))$ is computed. This gives all possible next control states, assuming that communication with the integer datapath is free. Each minterm $(x_I, b, w) \in \exists y_F U_i(y_F, x_I, b, w)$ is then simulated through the integer datapath. Inconsistent (x_I, b, w)'s are ignored, and the remaining minterms are used to compute the integer portion of the transition relation $T^I_{i+1}(x_I, y_I, b, w)$, where T^I_{i+1} represents the set of transitions of the datapath reachable in $i+1$ steps. $R_{i+1}(y_F, y_I)$ is then computed using the equation above. The algorithm is given below.

1. Let $U_i(y_F, x_I, b, w) = \exists x_F (T_F(x_F, y_F, b, w) \wedge R_i(x_F, x_I))$.

2. For each $(x_I, b, w) \in \exists y_F U_i(y_F, x_I, b, w)$,

2a. Propagate values through the integer datapath. If an integer predicate gate is encountered, then assign it the value given to it by b*, if possible. If not, the tuple* (x_I, b, w) *is invalid. Stop value propagation.*

2b. If a new ICS term has been assigned to a next state integer variable, add it to H_I *along with a unique encoding. Extend the range of all integer latch MDD variables.*

2c. Add the new integer predicates to the predicate table. If the new predicate table is not in H_P *then add it to* H_P *along with a unique encoding. Extend the range of* p.

2d. If the new memory has not been encountered before, add it to H_M *along with a unique encoding. Extend the range of* m.

3. Let $T^I_{i+1}(x_I, y_I, b, w) = \sum\{b \wedge w \wedge (x_I, y_I)\}$*. Here,* y_I *represents the encoded next state assignments to latch, predicate and memory assignments generated from step 2 (due to propagation of* (x_I, b, w)*).*

4. Let $R_{i+1}(y_F, y_I) = \exists b \exists w \exists x_I \Big(U_i(y_F, x_I, b, w) \wedge T^I_{i+1}(x_I, y_I, b, w) \Big)$.

The details of step 2 above are as follows.

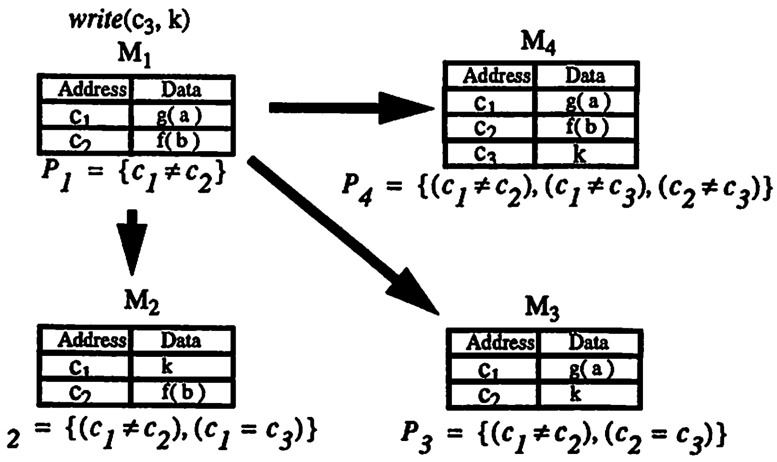
1. In 2a, if the gate executed represents a predicate Q, then, if $Q_1 \wedge \ldots \wedge Q_k \rightarrow (Q = b_Q)$ is valid,

allow propagation to continue. If, however, $Q_1 \wedge \ldots \wedge Q_k \rightarrow (Q = \overline{b_Q})$ is valid, then stop propagation. If neither are valid, then let $Q = b_Q$, and add it to the predicate table. Note that when propagation is started, the predicate table is taken from x_1.

2. If the gate executed corresponds to a $read(x)$, find an address y in the memory table such that $Q_1 \wedge \ldots \wedge Q_k \rightarrow (x = y)$ is valid. If such a case exists, return the data value corresponding to y. Otherwise, for each address y, where $Q_1 \wedge \ldots \wedge Q_k \rightarrow (x \neq y)$ is not valid, add a new predicate $x = y$ to the predicate table and return the data value corresponding to x. The case where all such predicates are false must also be considered. In this case, add y to the memory table, along with a new fresh constant as the data value. Return this fresh constant. This case corresponds to reading a location that has never been written too. Note that when propagation is started, the memory table is taken from x_1.

3. If the gate executed corresponds to a $write(x, d)$, find an address y in the memory table such that $Q_1 \wedge \ldots \wedge Q_k \rightarrow (x = y)$ is valid. If such a case exists, write d to the memory location associated with y. Otherwise, for each address y, where $Q_1 \wedge \ldots \wedge Q_k \rightarrow (x \neq y)$ is not valid, add a new predicate $x = y$ to the predicate table and write d to the memory location associated with x. For the case where all such predicates are false, add location y to the memory table, with data d. An example is given in Figure 4.2.2.

4. Our implementation disallows arithmetic functions and predicates. Thus, congruence closure (*[NO80]*) is sufficient for validity checking of ICS terms.



In the diagram to the left, a $write(c_3, k)$ is performed to memory M_1. Since c_3 is not equal to the addresses c_1 and c_2 currently stored in M_1, all cases have to be explored: $c_3 = c_1$, $c_3 = c_2$ and $c_1 \neq c_2 \neq c_3$. This creates three new paths of propagation, along with the associated memory tables M_2, M_3, and M_4.

Figure 4.2.2: Performing a write during symbolic simulation

4.3 Experimental Reachability Analysis Results

The algorithm presented in Section 4.2 has been implemented in the VIS environment (*[VIS95]*), UC Berkeley's and the University of Colorado's second generation, BDD-based verification system. In Section 4.3.1, a bottleneck associated with this algorithm is presented. Next, in Section 4.3.2, reachability results are given for a simple three stage microprocessor, referred to as CMU that was first published in *[BCMD90]*.

4.3.1 A Major Bottleneck in Computing Reachable States of Microprocessors

There is a significant bottleneck that occurs when symbolically executing memory gates during ICS reachability analysis. In particular, when writing to or reading from a particular memory location that is not equal to any other location in memory, all possible locations have to be explored. This can lead to a case explosion. This is particularly the case in microprocessors, where each instruction generates two reads and one write. For example, suppose there are two locations a_1 and a_2 to be read, and there is one location a_3 to be written. Assume a_1, a_2, and a_3 are all different constants. The ICS reachability algorithm must consider five possible cases: $a_1 \neq a_2 \neq a_3$, $a_1 = a_2$, $a_1 = a_3$, $a_2 = a_3$, and $a_1 = a_2 = a_3$. Let's now consider the situation where there are n addresses. Here, the number of cases that have to be considered is equal to the number of sets of subsets of elements of n (denoted $p(n)$). In table 4.3.1.1, $p(n)$ is calculated for small values of n. Note that this is a lower bound on the number of cases, since there may be more memory configurations for each trace. This case explosion is the primary bottleneck in the algorithm.

Table 4.3.1.1 Memory Configuration Explosion

n	1	2	3	4	5	6	7	8	9
p(n)	1	2	5	15	52	213	877	4175	21707

4.3.2 Reachability Analysis Results

In performing reachability analysis, experiments were performed on the CMU pipeline. CMU is a three stage pipeline. In the first stage, instructions are fetched from memory and then decoded. In the second stage, instructions are executed and in the third stage instructions are written back to memory. The ALU is abstracted away and replaced with a single uninterpreted function. Note that CMU has only memory and no register file. Although simple, most instructions in CMU can cause a read of two locations and a write to a third location. Therefore, after *n* steps of reachability, at most *3n* memory locations are stored (remember a read can cause the creation of a memory location, if that location has not previously been written). The results of the experiment are shown in table 4.3.2.1. The columns include the number of reachable states, the maximum number of memory locations created, the number of present state BDD variables, the number of ICS terms, and the simulation time (in CPU seconds). The MDD package used in the ICS reachability tool, encodes MDD variables into a set of BDD variables. Thus, the increase in present state BDD variables reflects the increase in MDD variable ranges. Note that only 3 cycles were executed. The experiment was performed on a DECStation Alpha 21064, 182MHz, with 1 gigabyte of main memory.

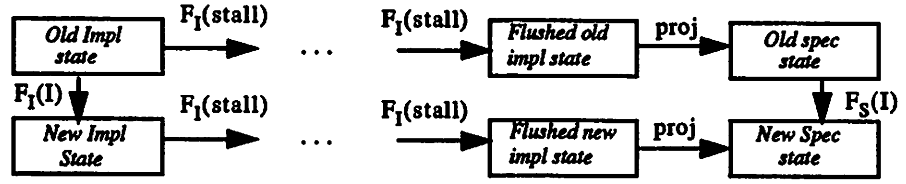
Table 4.3.2.1: Reachability Analysis Results on CMU

steps	reachable states	Max Mem Ops	PS Vars	ICS Terms	Time
1	12	3	32	16	0.1
2	378	6	45	32	1.6
3	33525	9	73	148	170.8

4.4 Pipelined Microprocessor Verification

The goal here is to verify if a pipelined implementation of a microprocessor implements its unpipelined specification. The unpipelined version, called the ***architectural model***, represents the instruction set architecture, which consists of the ***programmer visible state*** and instructions. Programmer visible states includes logical registers, the program counter and memory. In general this model describes how the programmer should expect the machine to behave for each instruction. Thus, intuitively, what one would like to verify is that for any sequence of instructions given to both machines, when the pipeline completes, its programmer visible state will equal the programmer visible state given in the architectural model.

Traditionally, one of the difficulties with performing pipeline verification is mapping pipeline states to architectural states. The problem here is one of ***pipeline latency***. Although an instruction given to the architectural model takes only one cycle to complete, it may take several cycles for the pipeline to execute that same instruction. Thus, unless all instructions take exactly the same number of cycles to complete, coming up with a state mapping function may be a very difficult problem. Note that in complex superscalar machines, this mapping task is further complicated by execution of instructions in parallel or out of order. Burch and Dill *[BD94]* came up with a novel approach to this problem. Here, the pipeline itself is used as a temporal abstraction function between it and the architectural model. Intuitively, this is done by flushing the pipeline after an instruction has been executed. Once the pipeline has been flushed, then all the instructions in the pipeline have been executed. The technique in *[BD94]* can be described as follows. For each arbitrary pipeline state (denoted *old implementation state*), the pipeline is flushed. The flushed pipeline state is then projected onto an equivalent state of the architectural model (denoted *old spec state*). It is necessary to flush the pipeline because there may be partially executed instructions still sitting there. It is very difficult to find out what the equivalent state in the architectural model is while these instruction are still in the pipeline. Next, an instruction is given to both the pipeline and the architectural model, starting at the old implementation state and the old specification state, respectively. The pipeline is then flushed and the resulting state is then projected onto the state space of the architectural model. If the final two states are the same, then the pipeline is considered correct. This correctness criteria reduces to a combinational check along two paths i.e. checking the transitions along the two paths are equal given a subset of states. A diagram of this technique is shown in figure 4.4.1.



The corrections criteria of [BD94] checks whether the transitions functions along both paths in the above diagram are equivalent. Along one path, starting for the that state labeled Old Impl state, the pipelined is flushed, projected on the spec state space, and an instruction is run. Along the second path, an instruction is run and then flushed and projected.

Figure 4.4.1 The Burch and Dill Approach.

We have approximated *[BD94]* by checking whether the sets of states reachable along both paths in figure 4.3.3.1 are equivalent. Note that this is a necessary but not sufficient condition, since although the sets of reachable states along both paths may be equal, they may be reachable under different instructions. The arbitrary pipeline state was generated by giving all latches and memory fresh constants, and allows the control state variables to non-deterministically be given arbitrary values. In practice, for many microprocessors, the initial set of states has to be restricted. Otherwise, the two functions would not be equal. An instruction was then given to both the architectural model and pipeline.

Experiments were run on two microprocessors: CMU, as described in Section 4.3.1 and DLX. DLX has a five stage pipeline. These stages are fetch, decode, execute, memory access and write back. Unlike CMU, DLX has both a register file and a main memory. In table 4.3.3.1, the results are given for CMU. The table includes columns for the number of reachable states in the pipeline, the maximum number of memory operations, the number of present state BDD variables, the number of ICS terms stored and the total CPU time. The experiment was run on a DECStation Alpha 21064, 182 MHz, with 1 gigabyte of main memory.

The algorithm was unable to complete for DLX, due to the memory bottleneck described in Section 4.3.1.

Table 4.4.1 ICS Approximation of *[BD94]* on CMU

steps	reachable states	Max Mem Ops	PS Vars	ICS Terms	Time
CMU	78	5	48	53	0.8

4.4.1 Comparison to other approaches

Much of the previous work in pipeline verification can be classified as either theorem proving or automatic methods. Since the technique described in this chapter is automatic, the comparison here will be restricted to other techniques which are similar to ours.

Multi-Way Decision Graphs (MDGs, [Cor93]). An MDG is a data structure that extends BDDs by allowing uninterpreted functions to be represented in the graph. This technique, however, does not seem to have the same capabilities of ICS, since MDGs cannot represent memories or predicates. Note that the current bottleneck is not the size of the reachable state set, but rather the case explosion that occurs when executing memory gates.

Comparison with [BD94]. The advantage of *[BD94]* is that it is more efficient than the algorithm for ICS reachability presented in this chapter. In addition to verifying DLX, they have successfully applied their tool for verifying other architectures, including portions of Stanford's Flash multiprocessor *[JDB95]*. Intuitively, their paradigm is faster and more efficient because it only requires reasoning over two transition functions. Generating and representing the set of reachable states does not have to be performed. Our technique is more general in the sense that we compute the set of reachable states and can check more general properties. But we suffer in efficiency.

The primary drawback with *[BD94]* is the state invariant problem. Here, the user has to give the set of reachable pipeline states (denoted ***old implementation state*** in figure 4.3.3.1) as an input to the algorithm. In theory, finding ***state invariants*** is as hard as the reachability problem. An example of a need for a state invariant in pipeline microprocessors occurs because of the ***load interlock*** problem *[HP90]*. In DLX, for example, ALU instructions are executed in stage 3 and memory operations are executed in stage 4 of the pipeline. Suppose there is a sequence of two instructions I_1 and I_2, where I_1 performs a load from memory to a register R, and I_2 takes the value in R and uses it to perform an ALU operation. Here, there is a data dependency between I_1 and I_2 and thus executing both instructions at the same time will cause an incorrect result. DLX does not permit this situation from occurring, and hence, such a state is unreachable. In a complex microprocessor, there maybe many such constraints.

Acknowledgment

The work in this chapter is based on the paper *[IHB96]*. In addition, the analysis in Section 4.3.1 is based on *[HojUn96]*.

Chapter 5

Conclusions and Future Work

This thesis has made several new contributions to performing verification in the ICS paradigm. Some of these contributions are outlined below.

1. A new definition of symbolic languages and automata of ICS Models was proposed. This definition was given because it was discovered that there was an inherent problem in the original definition presented in *[HB95]*. This problem centered around the fact that the operational semantics of ICS models allows behavior to occur in the datapath that may not have a concrete interpretation. In *[HB95]*, operations on the languages of symbolic automata, such as intersection and containment, was based on the ability to determine existence of such interpretations. Since we currently do not have a technique for determining which runs in a model have concrete interpretations, we felt that a re-definition was necessary. The new definition of symbolic ICS automata and languages was given chapter 2. It was then proved that the language of the composition of two ICS models is equal to the intersection of the languages of the two individual machines. This theorem is crucial for verifying ICS models in practice. In addition, Boolean operations on languages, such as union and intersection, were also redefined.

2. Three new results were presented which give bounds on the minimum number of finite values needed in the datapath to accurately verify certain types of ICS Models. We showed that for Data Semi-Sensitive Controllers, where only predicates of the form $x = y$ are used, the number of values needed in the integer datapath for verification can be reduced to the number of integer latches and constant creators in the system. Note that in *[HB95]* it was shown that the number of values needed was equal to the number of integer variables in the system. In practice, this new bound can reduce computational time for verification significantly. Next, a new abstraction technique, called data shifting derivations, was proposed that allows the number of bits needed in the datapath to be reduced even further, when compared to finite instantiations. Data shifting derivations is different than finite instantiations because it modifies the functionality of the transition relations in addition to reducing the range of the integer variables. For DSSCs with predicates of the form $x = y$, it was shown that with data shifting derivations, the range of the integer state variables can be reduced to the number of integer latches in the circuit. Finally, a previously open problem in the verification of ICS Models is solved by showing that verification of DSSC's with predicates $x < y$ and $x = y$ is decidable and an exact data shifting derivation exists. Previously, it was proved that no finite instantiations exist for this type of controller.

3. Finally, we presented experimental results, produced by the first generation ICS reachability tool. In particular, results were given for computing the set of reachable states on the three stage pipelined microprocessor called CMU. Our reachability tool was able to perform symbolic simulation for three cycles before

our tool ran out of memory. While performing experimental results, however, a major bottleneck associated with our algorithm was found. This bottleneck was due to an explosion in the number of memory locations and was the primary cause of our tool running out of memory. Finally, we demonstrated that ICS reachability can be employed in solving the specific problem of pipelined processor verification. Here, one would like to verify that a pipelined microprocessor implements its instruction set architecture. In *[BD94]*, Burch and Dill presented a unique approach for solving this problem. We showed that ICS reachability can be used to approximate this approach. Experimental results were given for two simple pipelined microprocessors: CMU and DLX. For the CMU microprocessor, we were able to use our technique to verify its correctness. For DLX, however, our tool did not complete, due also to an explosion in the number of memory locations during its symbolic execution.

There are many possible directions that can be taken with this research. Below we outline a few research directions that may show promise.

1. One of the most important directions this research should take is in improving the efficiency of symbolic simulation and dealing with the memory explosion problem outlined in Chapter 4. Recently, in *[HIB97]*, we have developed new techniques for collapsing together ICS states that are equivalent with respect to certain properties being verified. We feel that these new techniques may be very helpful in dealing with the memory explosion problem and useful in improving the overall speed of our tool.

2. Extend the class of ICS circuits for which automatic abstraction techniques can be performed, such as finite instantiations and data-shifting derivations. In particular, the problem of proving the property "when b becomes true, $x = y$ "is still open for circuits where the datapath contains data movement gates and uninterpreted functions, but no predicates.

References

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

[BCMD90] Jerry R. Burch, E. M. Clarke, K. L. McMillan, David L. Dill, *"Sequential Circuit Verification Using Symbolic Model Checking"*, In 27th ACM/IEEE Design Automation Conference, 1990.

[BD94] Jerry R. Burch, David L. Dill, *"Automatic Verification of Pipelined Microprocessor Control"*, Computer Aided Verification, Stanford, CA, June 1994.

[Cor93] F. Corella, *"Automatic High-Level Verification Against Clocked Algorithmic Specifications"*, Proceedings of the IFIP WG10.2 Conference on Computer Hardware Description Languages and their Applications, Ottawa, Canada, Apr. 1993. Elsevier Science Publishers B.V.

[Cou89] Olivier Coudert, Christian Berthet, and Jean-Christophe Madre, *"Verification of Sequential Machines Based on Symbolic Execution"*, Proceedings of the Workshop on Automatic Verification Methods for Finite States Systems, Grenoble, France, 1989.

[HP90] John L. Hennessy, David A. Patterson, *"Computer Architecture A Quantitative Approach"*, Morgan Kaufmann Publishers, 1990.

[HB95] Ramin Hojati, Robert K. Brayton, *"Automatic Datapath Abstraction of Hardware Systems"*, Conference on Computer-Aided Verification, 1995.

[HDB97] Ramin Hojati, David L. Dill, Robert K. Brayton, *"Verifying Linear Temporal Properties of Data Insensitive Controllers"*, to appear in Conference on Hardware Description Languages (CHDL) 1997.

[HIB97] Ramin Hojati, Adrian J. Isles, Robert K. Brayton, *"State Reduction Techniques of Hardware Systems Modeled Using Uninterpreted Functions and Finite Instantiations"*, submitted to the International Conference on Computer-Aided Design, 1997.

[HIKB96] Ramin Hojati, Adrian Isles, Desmond Kirkpatrick, Robert K. Brayton, *"Verification Using Uninterpreted Functions and Finite Instantiations"*, FMCAD, 1996.

[HMB94] Ramin Hojati, Vigyan Singhal, Robert K. Brayton, *"Edge-Strett/Edge-Rabin Automata Environment for Formal Verification Using Langauge Containment"*, Memorandum No. UCB/ERL M94/12, UC Berkeley, 1994.

[Hoj96] Ramin Hojati, *"A BDD-Based Environment for Formal Verification of Hardware Systems"*, Ph. D. thesis, University of California at Berkeley, 1996.

[HohUn96] Ramin Hojati, *"On calculating p(n) for small values"*, unpublished document, 1996

[HSIS94] A. Aziz, F. Balarin, S. T. Cheng, R. Hojati, T. Kam, S. C. Krishnan, R. K. Ranjan, T. R. Shiple, V. Singhal, S. Tasiran, H.-Y. Wang, R. K. Brayton and A. L. Sangiovanni-Vincentelli, *"HSIS: A BDD-Based Environment for Formal Verification"*, Design Automation Conference, 1994.

[IHB96] Adrian J. Isles, Ramin Hojati, and Robert K. Brayton, *"Reachability Analysis of ICS Models"*, SRC Techcon, September 1996.

[JDB95] Robert B. Jones, David L. Dill, Jerry R. Burch, *"Efficient Validity Checking for Processor Verification"*, Proceedings of IEEE International Conference on Computer Aided Design (ICCAD95), November 1995.

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

[Kam92] Timothy Kam and Robert K. Brayton, *"Multi-valued Decision Diagrams"*, Electronics Research Laboratory, University of California, Berkeley, Memorandum No. UCB/ERL, M90/125, 1990.

[Kur87] R. P. Kurshan, *"Reducibility in Analysis of Coordination"*, In LNCIS, volume 103, pages 19-39, Springer-Verlag, 1987.

[NO80] Greg Nelson, Derek C. Oppen, *"Fast Decision Procedures Based on Congruence Closure"*, Journal of the ACM, 27(2):356-364, April 1980.

[Rho95] Richard C. Ho, Han Yang, Mark A. Horowitz, David L. Dill, *"Architecture Validation for Processors"*, Proceedings of the 22nd Annual International Symposium on Computer Architecture, June 1995.

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

[Sho82] R. E. Shostak, *"Deciding Combinations of Theories"*, proceedings of sixth conference on Automated Deduction (Loveland, Ed.), Lecture Notes in Computer Science, 138, Springer-Verlag, Berlin, June 1982, 209-223.

[Tou90] Herve' J. Touati, Hamid Savoj, Bill Lin, Robert K. Brayton, and Alberto Sangiovanni-Vincentelli, *"Implicit State Enumeration of Finite State Machines using BDDs"*, Proceedings of the International Conference on Computer Aided Design, pages 130 - 133, 1990.

[VIS96] The VIS Group, *"VIS: A system for Verification and Synthesis"*, In the Proceedings of the 8th International Conference on Computer Aided Verifictaion,p428-432, Springer Lecture Notes in Computer Science, #1102, Edited by R. Alur and T. Henzinger, New Brunswick, NJ, July 1996.

[VW86] Moshe Y. Vardi and Pierre Wolper. *"An Automata-theoretic Approach to Automatic Program Verification"*. In Proc. IEEE Symp. on Logic in Computer Science, pages 332-344, Boston, July 1986.