Automated Verification with Abstract State Machines Using Multiway Decision Graphs

E. Cerny, F. Corella, M. Langevin, X. Song, S. Tahar, and Z. Zhou

1. Introduction

1.1 Motivations

Formal verification methods can be classified into two main categories: interactive verification using a theorem prover and automated finite state machine (FSM) verification based on state enumeration [Gupt92].

The most general approach to verification is to state the correctness condition for a system as a theorem in a mathematical logic and to generate a proof of this theorem that is verified using a general-purpose theorem-prover. Theorem provers use powerful formalisms such as higher-order logic [GoMe93] that allow the verification problem to be stated at many levels of abstraction. This approach has attained significant success in verifying microprocessor designs, for example [Hunt85, Joyc90, SrMi95, TaKu95]. However, theorem-proving-based verification has a drawback, viz. the user is responsible for coming up with the proof of correctness and for feeding it to the theorem prover, which can be quite difficult and time consuming.

At the other extreme of the spectrum lies state space exploration of finite state machines. State enumeration techniques permit automatic behavioral comparison and model checking [TSLB90, BCLM94]. They are effective for detecting design errors in finite-state systems. The major problem with these methods is that the size of the state space may grow very rapidly with the size of the model. This is known as the *state explosion problem.*

Many strategies have been proposed to alleviate the state explosion problem [BoFi89a, BrBS91, BCLM94, CHJP90, CPVM91, CoMa90, TSLB90]. They exploit Bryant's Reduced Ordered Binary Decision Diagrams (ROBDDs) [Brya86] to encode sets of states and to perform an implicit enumeration of the state space, making it possible to verify FSMs with a large number of states. For some specific circuits with datapath, these methods achieve linear complexity with respect to the data width. However, these methods are not adequate in general for verifying circuits with large and complex datapaths, still leading to the state explosion problem. Even the ROBDD encoding cannot resolve the problem because of the binary representation of the circuit. More specifically, every individual bit of every data signal must be represented by a separate Boolean variable, while the size of an ROBDD grows, sometimes exponentially, with the number of variables. This means that ROBDD-based verification methods often take too much time, or run

out of memory, when applied to circuits having a complex data path. Furthermore, these methods do not permit an abstract representation of the circuit, in contrast to the approaches based on theorem proving.

To overcome some of the above drawbacks, we present here a new verification approach based on *abstract descriptions of state machines* (ASM) which are encoded by a new class of decision graphs, called *Multiway Decision Graphs* (MDGs) [CZSL94], of which ROBDDs are a special case. The essential contribution of MDGs is that they make it possible to integrate two verification techniques that have been very successful: *implicit state enumeration* on one hand, and *the use of abstract sorts and uninterpreted function symbols* on the other. MDGs are decision graphs that can represent relations as well as sets of states. They allow sharing of isomorphic subgraphs which decreases the size of the graphs. MDGs incorporate variables of abstract types to denote data values and uninterpreted function symbols to denote data operations. This means that sequential circuits can be verified with a runtime that is independent of the width of the datapath. In MDG-based verification, abstract descriptions of state machines (ASM) are used to model the systems. Note that the ASMs are not a new kind of a state machine, but rather a new way of describing state machines at a higher level of abstraction. While the state machines that we want to verify are ordinary finite state machines (FSM), the abstract descriptions admit non-finite state machines as models in addition to their intended finite interpretations. The motivation for such abstract descriptions is eminently practical: it is possible to verify a circuit at the register transfer (RT) level without getting bogged down in the details of a gate-level implementation. Thus, we can raise the level of abstraction of automated verification methods to approach those of interactive methods, without sacrificing automation.

1.2 Limitations of the approach

Our approach, on the other hand, has its own significant limitations. First, the fact that function symbols denoting data operations are uninterpreted means that correctness must not depend on their intended denotation. That is, the implementation and the specification must be stated in terms of the same uninterpreted function symbols, and the correctness statement to be verified must hold for any allowable interpretation of those function symbols. For example, a circuit that computes the GCD of two numbers by repeated subtraction cannot be compared against a specification where the GCD is computed by repeated division, since the implementation and the specification use different function symbols in this case, and correctness depends on the arithmetic meaning of those symbols.

This limitation can be alleviated by the use of *term rewriting* or other automated deduction techniques. A conditional term rewriting algorithm for MDGs is provided with the MDG package, but will be described elsewhere. Rewrite rules can be viewed as axioms that limit the range of allowable

interpretations of the function symbols that denote data operations. (Some authors say that the function symbols become *partially interpreted.*) The use of rewrite rules extends the class of verification problems that can be solved but reduces the degree of automation, since the user has to provide a problem-specific set of rules. The possibility of combining rewriting and other automated deduction techniques with state exploration opens up exciting possibilities for further research.

A second limitation is the fact that the computation of the set of reachable states does not always terminate. This is discussed in Section 4.2.3. A third limitation is the fact that we have not implemented algorithms for the verification of liveness properties. We expect to be able to do this in the future.

1.3 Related Work

Interactive verification by theorem proving does not require a Boolean representation of the circuit: it is usually carried out at a higher level of abstraction. Indeed, part of the inspiration for our work comes from prior work on interactive verification, and in particular from the fact that Joyce verified the Tamarack-3 microprocessor at such a high level of abstraction that he did not even mention the width of the datapath [Joyc90]. This is in striking contrast with ROBDD-based methods, where an increase in the width of the datapath often makes verification impossible.

In the automated verification community, the difficulties faced by Boolean methods when verifying circuits with a substantial datapath are well known, and have been tackled by many researchers.

Clarke, Grumberg, and Long [ClGL92, Long93] have shown examples where verifications problems involving circuits with wide datapaths can be reduced by a *data abstraction* technique to simpler problems involving *parameterized circuit descriptions* where the datapath is only a few bits wide. These simpler problems can then be solved by ROBDD-based model checking. However, the fact that correctness of the simpler circuit implies correctness of the original circuit is not always obvious, and is not verified mechanically. Also, the data abstraction function has to be provided by the user; this may require considerable ingenuity, and has to be done anew for each verification problem. Similarly, Kurshan [Kurs89] has proposed the use of *homomorphic reductions* to simplify verification problems stated as tests of ω-language containment. Again, each problem requires its own homomorphism, which has to be provided by the user.

Wolper [Wolp86] has shown that *data independent systems* can be verified by reducing the domain of data values to a very small set. But data independent systems are essentially systems that transfer data without observing it or performing any computation on it, and thus the method is not widely applicable.

In some cases it is possible to restate a verification problem concerning a circuit that consists of a datapath and a controller in terms of the controller only [VLAD92, Fuji92]. In this case the CPU time needed for verification is of course independent of the width of the datapath, which is not the case for the data abstraction method of [ClGL92]. However this is practically feasible only when the interface between the datapath and the control circuitry is easy to specify, and the equivalence of the original problem to the restated one is not verified mechanically.

In contrast to these *problem reduction* techniques, new representation tools have been developed which expand the range of circuits that can be verified directly, without recourse to ingenious problem transformations. Recently, a number of ROBDD extensions such as BMDs [BrCh95], HDDs [ClFZ95] and K*BMDs [DrBR95] have been developed to represent functions that map Boolean variables to integer values. They are mainly useful for verifying arithmetic circuits.

Our approach has its roots in the work of Langevin and Cerny [LaCe91, LaCe91a, LaCe94] and Corella [Core93, Core94], who have independently developed similar techniques based on the use of variables of abstract type to denote data values and uninterpreted function symbols to denote data operations. These approaches are well-suited for verifying simple microprocessors, as well as circuits produced by high-level synthesis, since in both cases data operations are viewed as black boxes. However, explicit control state enumeration was used, and this is not adequate for circuits containing complex controllers.

The immediate precursors of MDGs are Langevin and Cerny's EOBDDs [LaCe94]. EOBDDs were used to represent the transition and output relation of a sequential circuit. However, they were not used to represent *sets of states.* With MDGs we go one step further: we are able to represent sets of abstract states, just like ROBDDs can be used to represent sets of states in the Boolean domain. We are thus able to lift the technique of *implicit state enumeration* from the Boolean domain (where ROBDDs are used) to the domain of abstract types (where MDGs are used); we call the lifted technique *abstract implicit enumeration.*

MDGs are similar in name and structure to the *Multivalued Decision Diagrams* (MDDs) of [SKMB90], but the similarity is superficial. MDDs and MDGs have in common that any number of edges can issue from a given node. In MDDs, however, those edges are labeled by constants that denote pairwise distinct values comprising the entire range of values for the node. In MDGs the labels of the edges can be first-order terms, need not be mutually exclusive, and need not denote all the values in a given range. This makes it possible to use variables of abstract type and uninterpreted function symbols in MDGs, which is not possible in MDDs.

More recently, a number of automatic verification methods emerged which are also based on the use of abstract sorts and uninterpreted function sym-

bols. Burch and Dill [BuDi94, JoDi95] used a validity checking algorithm for instruction-set processor verification. A logic expression representing the correctness statement is generated using symbolic simulation. The algorithm is then used to check its validity. The authors verified a subset of the RISC pipeline processor DLX [BuDi94] and a protocol processor (PP) [JoDi95], using problem-specific heuristics.

Galter [Galt94] also presented a similar symbolic approach for the verification of processors. Two IF-expressions (If-Then-Else) which represent the functions of the specification and the implementation are derived using symbolic execution. They are then compared for syntactic equivalence. As IF-expressions may grow exponentially, a technique called IF-algebra was developed to simplify the expressions. The benchmark Tamarack-3 microprocessor was verified using the method.

Barringer [Barr95] proposed a verification methodology which can be characterized as symbolic simulation plus theorem proving. The symbolic simulation is performed on the implementation and the specification for a finite number (system-dependent) of steps, generating a pair of logical expressions which represent the circuit behaviors. These two expressions are further analyzed automatically and decomposed into sets of smaller expressions called *equivalent verification conditions* which are then checked by the theorem prover PVS.

Cyrluk and Narendran [CRSS94] defined a first-order temporal logic – *Ground Temporal Logic* (GTL) which also uses uninterpreted function symbols. Using a decidable fragment of GTL, they are able to automate part of the verification at a higher level of abstraction in the PVS theorem-proving system.

All the above methods are in fact validity checking procedures of logic formulas. Therefore, they are not applicable to state exploration-based verification such as model checking or behavioral equivalence checking. In contrast, MDGs are capable of both validity checking and verification based on state-space exploration.

1.4 Outline

We describe the theoretical foundations of our approach in Section 2. In particular, we define the formal logic used and the structure of MDGs, and briefly describe the basic MDG manipulation algorithms. Furthermore, we formulate the abstract description of a state machine and show how abstract state enumeration proceeds using MDGs. In Section 3, we describe hardware modeling using our approach, i.e., how to describe circuit components using MDGs. In Section 4, we present the application of our method to hardware verification. In particular, several techniques for combinational and sequential circuits are discussed. In Section 5, we report experimental results on a number of the IFIP benchmarks. In Section 6, we present a case study of formal verification of the Fairisle 4×4 ATM (Asynchronous Transfer Mode) switch fabric

using MDGs, including experimental results. In Section 7, we summarize the contributions of the paper and point out the direction of further work.

2. Foundations of the Methodology

While Boolean logic is sufficient to represent circuits at the bit level, to represent and reason about circuits using abstract types and uninterpreted function symbols we need a first-order logic. We use a many-sorted first-order logic with a distinction between abstract and concrete sorts that mirrors the hardware distinction between data path and control. Multiway Decision Graphs are canonical representations of a certain class of quantifier-free formulas of the logic, which we call *Directed Formulas* (DFs). DFs can represent the transition and output relations of a state machine, as well as the set of possible initial states and the sets of states that arise during reachability analysis. We refer to state machines whose transition relation, output relation, and the set of initial states are given by DFs, or equivalently by MDGs, as *Abstract State Machines* (ASMs).

2.1 Logic

2.1.1 Syntax. As in ordinary many-sorted first-order logic, the vocabulary consists of *sorts*, *constants*, *variables*, and *function symbols* (or *operators*). Constants and variables have sorts. An n-ary function symbol ($n > 0$) has a type $\alpha_1 \times \ldots \times \alpha_n \rightarrow \alpha_{n+1}$, where $\alpha_1 \ldots \alpha_{n+1}$ are sorts. We deviate from standard many-sorted first-order logic by introducing a distinction between *concrete* (or *enumerated*) sorts, and *abstract sorts*; the difference is that concrete sorts have *enumerations*, while abstract sorts do not. The enumeration of a concrete sort α is a set of distinct constants of sort α. We refer to constants occurring in enumerations as *individual constants*, and to other constants as *generic constants*. An individual constant can appear in the enumeration of more than one sort α, and is said to be of sort α for each of them. Variables and generic constants, on the other hand, have unique sorts.

The distinction between abstract and concrete sorts leads to a distinction between three kinds of function symbols. Let f be a function symbol of type $\alpha_1 \times \ldots \times \alpha_n \rightarrow \alpha_{n+1}$. If α_{n+1} is an abstract sort then f is an *abstract function symbol*. If all the $\alpha_1 \ldots \alpha_{n+1}$ are concrete, f is a *concrete* function symbol. If α_{n+1} is concrete while at least one of $\alpha_1 \ldots \alpha_n$ is abstract, then we refer to f as a *cross-operator*. While abstract function symbols are used to denote data operations, cross-operators are used to denote feedback from the data path to the control circuitry. Both abstract function symbols and cross-operators are *uninterpreted*, i.e. their intended interpretation is not specified. However, information about them can be provided by axioms such as conditional equations which can be used as conditional rewrite rules. Such axioms limit the range of allowable interpretations.

The terms and their types (sorts) are defined inductively as follows: a constant or variable of sort α is a term of type α; and if f is a function symbol of type $\alpha_1 \times \ldots \times \alpha_n \rightarrow \alpha_{n+1}$, $n \geq 1$, and $A_1 \ldots A_n$ are terms of types $\alpha_1 \ldots \alpha_n$, then $f(A_1, \ldots, A_n)$ is a term of type α_{n+1}. A term consisting of a single occurrence of an individual constant has multiple types (the sorts of the constant) but every other term has a unique type. The *top symbol* of a term is defined as follows: the top symbol of $f(A_1, \ldots, A_n)$ is f, and the top symbol of a term consisting of a single occurrence of a variable or a constant is that variable or constant.

We say that a term, variable or constant is concrete (resp. abstract) to indicate that it is of concrete (resp. abstract) sort. A term is *concretely reduced* if and only if it contains no concrete terms other than individual constants. Thus a concretely reduced term can contain abstract function symbols, abstract variables, abstract generic constants and individual constants, but it can contain no cross-operators, concrete function symbols, concrete generic constants, or concrete variables; and a concretely reduced term that is itself concrete must be an individual constant. A term of the form "$f(A_1, \ldots, A_n)$" where f is a cross-operator and $A_1 \ldots A_n$ are concretely-reduced terms is called a *cross-term*. For example, if f is an abstract function symbol, c is an individual constant, x is a variable of concrete sort, and y is a variable of abstract sort, then $f(c, y)$ is a concretely-reduced term (assuming that it is well-typed) while $f(x, y)$ is not. And if g is a cross-operator, then $g(c, y)$ is a cross-term (again, assuming that it is well typed) but $g(x, y)$ is not.

A (well-typed) *equation* is an expression "$A_1 = A_2$" where the left-hand side (LHS) A_1 and the right-hand side (RHS) A_2 are terms of same type α. The *atomic formulas* are the equations, plus T (truth) and F (falsity). The *formulas* are defined inductively as follows: an atomic formula is a formula; if P and Q are formulas, then $\neg P$, $P \wedge Q$ and $P \vee Q$ are formulas; if P is a formula and x is a variable, then $(\exists x)P$ is a formula (with x bound in P). We use the abbreviation $P \Leftrightarrow Q$ for $(P \Rightarrow Q) \wedge (Q \Rightarrow P)$.

2.1.2 Semantics. An *interpretation* is a mapping ψ that assigns a denotation to each sort, constant and function symbol, and satisfies the following conditions:

1. The denotation $\psi(\alpha)$ of an abstract sort α is a non-empty set.
2. If α is a concrete sort with enumeration $\{a_1, \ldots, a_n\}$ then $\psi(\alpha) = \{\psi(a_1), \ldots, \psi(a_n)\}$ and $\psi(a_i) \neq \psi(a_j)$ for $1 \leq i < j \leq n$.
3. If c is a generic constant of sort α, then $\psi(c) \in \psi(\alpha)$. If f is a function symbol of type $\alpha_1 \times \ldots \times \alpha_n \rightarrow \alpha_{n+1}$, then $\psi(f)$ is a function from the cartesian product $\psi(\alpha_1) \times \ldots \times \psi(\alpha_n)$ into the set $\psi(\alpha_{n+1})$.

V being a set of variables, a *variable assignment* with domain V compatible with an interpretation ψ is a function ϕ that maps every variable $v \in V$ of sort α to an element $\phi(v)$ of $\psi(\alpha)$. We write Φ_V^{ψ} for the set of ψ-compatible assignments to the variables in V.

The denotation of a term under an interpretation ψ and a ψ-compatible variable assignment ϕ whose domain contains all the variables that occur in the term is defined by induction as follows: a constant c denotes $\psi(c)$; a variable x denotes $\phi(x)$; and if $A_1 \ldots A_n$ denote $\nu_1 \ldots \nu_n$, then $f(A_1, \ldots, A_n)$ denotes $(\phi(f))(\nu_1, \ldots, \nu_n)$. The truth of a formula P under an interpretation ψ and a ψ-compatible variable assignment ϕ whose domain contains the variables that occur free in P, written $\psi, \phi \models P$, is also defined by induction: $\psi, \phi \models A_1 = A_2$ iff A_1 and A_2 have same denotation; $\psi, \phi \models \neg P$ iff it is not the case that $\psi, \phi \models P$; $\psi, \phi \models P \wedge Q$ iff $\psi, \phi \models P$ and $\psi, \phi \models Q$; $\psi, \phi \models P \vee Q$ iff $\psi, \phi \models P$ or $\psi, \phi \models Q$; and $\psi, \phi \models (\exists x)P$ iff $\psi, \phi' \models P$ for some ϕ' that assigns an arbitrary value to x and otherwise coincides with ϕ.

We write $\psi \models P$ when $\psi, \phi \models P$ for every ψ-compatible assignment ϕ to the variables that occur free in P, and $\models P$ when $\psi \models P$ for all ψ. Two formulas P and Q are *logically equivalent* iff $\models P \Leftrightarrow Q$. A formula P *logically implies* a formula Q iff $\models P \Rightarrow Q$.

2.1.3 Directed Formulas. Given two disjoint sets of variables U and V, a directed formula of type $U \rightarrow V$ is a formula in disjunctive normal form (DNF) such that

1. Each disjunct is a conjunction of equations of the form
   - $A = a$, where A is a term of concrete sort α of the form "$f(B_1, \ldots, B_n)$" (f is thus a cross-operator) that contains no variables other than elements of U, and a is an individual constant in the enumeration of α, or
   - $u = a$, where $u \in U$ is a variable of concrete sort α and a is an individual constant in the enumeration of α, or
   - $v = a$, where $v \in V$ is a variable of concrete sort α and a is an individual constant in the enumeration of α, or
   - $v = A$, where $v \in V$ is a variable of abstract sort α and A is a term of type α containing no variables other than elements of U;
2. In each disjunct, the LHSs of the equations are pairwise distinct; and
3. Every abstract variable $v \in V$ appears as the LHS of an equation $v = A$ in each of the disjuncts. (Note that there need not be an equation $v = a$ for every concrete variable $v \in V$.)

Intuitively, in a DF of type $U \rightarrow V$, the U variables play the role of independent variables, the V variables play the role of dependent variables, and the disjuncts enumerate possible cases. In each disjunct, the equations of the form $u = a$ and $A = a$ specify a case in terms of the U variables, while the other equations specify the values of (some of the) V variables in that case. The cases need not be mutually exclusive, nor exhaustive. The condition that every abstract variable $v \in V$ must appear in every disjunct is less stringent than it seems. In practice, one can introduce an additional dependent variable u and add an equation $v = u$ to a disjunct where v is missing.

A DF is said to be *concretely reduced* iff every A in an equation $A = a$ is a cross-term, and every A in an equation $v = A$ is a concretely reduced term. It is easy to see that every DF is logically equivalent to a concretely reduced DF, given complete specifications of the concrete function symbols and concrete generic constants; the reduction can be accomplished by case splitting.

A concretely reduced DF contains no concrete function symbols and no concrete generic constants; and, in a concretely reduced DF of type $U \to V$, if A is the cross-term in the LHS of an equation $A = a$, or the concretely reduced term in the RHS of an equation $v = A$, then every variable that occurs in A is an abstract variable $u \in U$. We refer to such an occurrence of a variable as a *secondary occurrence* in the DF. A *primary occurrence* of a variable, on the other hand, is an occurrence as the LHS of an equation. From now on, by *DF* we shall mean *concretely reduced DF*.

For example, suppose that $U = \{u_1, u_2\}$ and $V = \{v_1, v_2\}$, where u_1 and v_1 are variables of a concrete sort *bool* with enumeration $\{0, 1\}$ while u_2 and v_2 are variables of an abstract sort *wordn*. Suppose that f is an abstract function symbol of type *wordn* $\to$ *wordn* and g is a cross-operator of type *wordn* $\to$ *bool*. Then the formula

$$\begin{array}{l} ((f(u_2) = 0) \wedge (v_2 = u_2)) \vee \\ ((f(u_2) = 1) \wedge (v_1 = u_1) \wedge (v_2 = g(u_2))) \end{array} \tag{2.1}$$

is a DF of type $U \to V$. In the case $f(u_2) = 0$ it assigns the symbolic value u_2 to v_2. In the case $f(u_2) = 1$ it assigns the symbolic values u_1 to v_1 and $g(u_2)$ to v_2. Note that, in the case $f(u_2) = 0$, the value of v_1 is left unspecified and thus is arbitrary.

The above DF (2.1) is *not* concretely reduced. This is because the right-hand side term u_1 in the second conjunct of the second disjunct is not concretely reduced. A concretely reduced DF logically equivalent to (2.1) can be obtained by further distinguishing the cases $u_1 = 0$ and $u_1 = 1$ in the case where $f(u_2) = 1$:

$$\begin{array}{l} ((f(u_2) = 0) \wedge (v_2 = u_2)) \vee \\ ((f(u_2) = 1) \wedge (u_1 = 0) \wedge (v_1 = 0) \wedge (v_2 = g(u_2))) \vee \\ ((f(u_2) = 1) \wedge (u_1 = 1) \wedge (v_1 = 1) \wedge (v_2 = g(u_2))) \end{array} \tag{2.2}$$

Note that, in the absence of abstract sorts, a DF contains only equations of the form $u = a$ or $v = a$, and the sets of variables U and V play symmetrical roles. If there is only one sort, and that sort is concrete with enumeration $\{0, 1\}$, then a DF is a simply a Boolean formula in DNF.

2.2 Multiway Decision Graphs

2.2.1 Structure. An ROBDD is usually viewed as the representation of a function, with the leaf nodes labeled by values (0 or 1). But it can also be

viewed as representing an assertion, with the leaf nodes labeled by propositions (truth or falsity). This latter view is the one that can be generalized to accommodate abstract types.

Let G be a finite directed acyclic graph with one root, whose internal nodes are labeled by terms and whose leaves are labeled by formulas of the logic. Then G can be viewed as representing a formula defined inductively as follows: (i) if G consists of a single leaf node labeled by a formula P, then G represents P; (ii) if G has a root node labeled A with edges labeled $B_1 \ldots B_n$ leading to subgraphs $G'_1 \ldots G'_n$, and if each G'_i represents a formula P_i, then G represents the formula $\bigvee_{1 \leq i \leq n}((A = B_i) \wedge P_i)$.

The concept of MDG is relative to two orderings, the *standard term ordering* and the *custom symbol ordering*. The standard term ordering is a total ordering of all the terms of the logic. The custom symbol ordering is a total ordering on a set of symbols C that includes the cross-operators, the concrete variables, and some, but not necessarily all, of the abstract variables. The custom symbol ordering need not be compatible with the standard term ordering. Variables that are elements of C are said to participate in the custom symbol ordering.

Let U and V be disjoint sets of variables, such that all the abstract variables in V participate in the custom symbol ordering. An *MDG of type* $U \rightarrow V$ is a directed acyclic graph (DAG) G with one root and ordered edges, such that:

1. Every leaf node is labeled by the formula T, except if G has a single node, which may be labeled T or F.
2. For every internal node N, either
   a) N is labeled by a cross-term A of type α with variables in U, and the edges that issue from N are labeled by individual constants in the enumeration of α, or
   b) N is labeled by a variable $u \in U$ of concrete sort α and the edges that issue from N are labeled by individual constants in the enumeration of α, or
   c) N is labeled by a variable $v \in V$ of concrete sort α and the edges that issue from N are labeled by individual constants in the enumeration of α, or
   d) N is labeled by a variable $v \in V$ of abstract sort α and the edges that issue from N are labeled by concretely reduced terms of sort α with variables in U.
3. Along every path, every abstract variable $v \in V$ appears as a node label, there are no duplicate node labels, the top symbols of the node labels appear in the custom symbol order, and nodes labeled by cross-terms with same cross-operator appear in the standard term order.
4. The edges issuing from a given node are arranged in the standard term order.

5. There are no distinct isomorphic subgraphs, and no redundant nodes, a node being *redundant* iff it is labeled by a concrete variable or cross-term of sort α whose edges are labeled by all the individual constants in the enumeration of α, and all lead to the same subgraph.
6. If a node N is labeled by an abstract variable x, and an abstract variable y participating in the custom symbol order occurs in a term A that labels one of the edges that issue form N, then y comes before x in the custom symbol order. Similarly, if N is labeled by a cross-term A with cross-operator f, and y is an abstract variable that occurs in A and participates in the custom symbol order, then y comes before f in the custom symbol order.

The *primary occurrences* and *secondary occurrences* of variables are defined in the same manner for MDGs as for DFs. Note that, given an MDG G, if U is the set of variables having secondary occurrences in G, and V the set of variables having primary occurrences, then G is of type $U \rightarrow V$.

When we say that an MDG is of type $U \rightarrow V$, it will always be understood that U and V are disjoint sets of variables, and that all the abstract variables in V participate in the custom symbol order.

An MDG is a graph representation of a formula as defined above. The formula represented by an MDG of type $U \rightarrow V$ is usually not in DNF. However, it can be put in DNF by distributing $\wedge$ over $\vee$. It is easy to see that the resulting formula is a concretely reduced DF of type $U \rightarrow V$, whose disjuncts correspond to the paths of the MDG. In this sense, we say that an MDG is a representation of a concretely reduced DF. As an example, the MDG shown in Figure 2.1 represents the DF (2.2).

Conversely, given a concretely reduced DF P of type $U \rightarrow V$, a standard term order, and a custom symbol order comprising all the variables in V and all the cross-operators in P, it is easy to construct an MDG representing a DF that coincides with P up the ordering of the disjuncts in each conjunct and the ordering of the conjuncts themselves.

The following theorem states that MDGs are a *canonical representation*:

Theorem 2.1. *For a given custom symbol order and a given standard term order, if G and G' are MDGs representing formulas P and P' respectively, and $\models P \Leftrightarrow P'$, then G and G' are isomorphic graphs.*

Although MDGs represent DFs, which are first-order formulas, this result is not surprising, because DFs are a *restricted class* of first-order formulas. The proof of the theorem can be found in [CZSL94]. The proof uses a notion of Herbrand model suitable for our logic. If G and G' are not isomorphic, a Herbrand model can be constructed that satisfies one of the formulas P or P', but not the other.

2.2.2 Basic algorithms. We have implemented the following basic MDG algorithms, which are the building blocks of the procedures for combinational

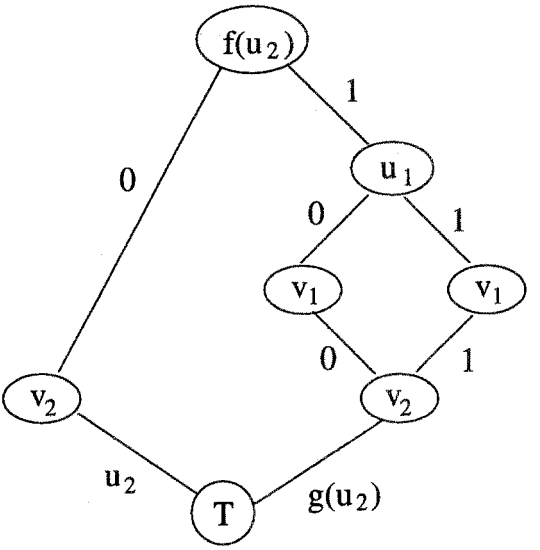

Fig. 2.1. MDG representing (2.2)

verification and reachability analysis. To simplify the description of the algorithms we shall identify an MDG with the formula that it represents.

Disjunction. Given two MDGs P_1 and P_2, there does not always exist an MDG R such that $\models R \Leftrightarrow (P_1 \vee P_2)$. For example, let x and y be distinct abstract variables, and a and b distinct abstract generic constants. Let P_1 be $x = a$ (i.e. an MDG with a root node labeled x and a single edge labeled a leading to T) and let P_2 be $y = b$. Then it can be shown that there exists no MDG R such that $\models R \Leftrightarrow (P_1 \vee P_2)$. But in the case where P_1 and P_2 have the same set of abstract primary variables, it is possible to compute an MDG R logically equivalent to $P_1 \vee P_2$.
Our disjunction algorithm is n-ary. It takes as inputs a set of MDGs P_i, $1 \leq i \leq n$, of types $U_i \rightarrow V$, and produces an MDG $R = \mathsf{Disj}(\{P_i\}_{1 \leq i \leq n})$ of type $(\bigcup_{1 \leq i \leq n} U_i) \rightarrow V$ such that

$$\models R \Leftrightarrow (\bigvee_{1 \leq i \leq n} P_i).$$

The algorithm computes the disjunction of its n inputs in one pass.

Relational product. As in the case of disjunction, given two MDGs P_1 and P_2, there does not always exist an MDG R such that $\models R \Leftrightarrow (P_1 \wedge P_2)$. For example, let x be an abstract variable, and let a and b be distinct abstract generic constants. Let P_1 be $x = a$ (i.e. an MDG with a root node labeled x and a single edge labeled a leading to T) and let P_2 be $x = b$. Then it can be shown that there exists no MDG R such that $\models R \Leftrightarrow (P_1 \wedge P_2)$.

But if P_1 and P_2 have no abstract primary variables in common, then it is possible to compute an MDG R logically equivalent to $P_1 \wedge P_2$. The abstract primary variables of R are those of P_1 and P_2. A secondary variable of R is a secondary variable of at least one of P_1, P_2 without being a primary variable of the other. (If a variable has secondary occurrences in one graph and primary occurrences in the other, the secondary occurrences are eliminated by substitution.)

Instead of implementing a conjunction algorithm, we have implemented a relational product algorithm that combines conjunction, existential quantification, and renaming. As in the case of disjunction, we have implemented an n-ary version of the algorithm. It takes as inputs a set of MDGs P_i, $1 \leq i \leq n$, of types $U_i \rightarrow V_i$, a set of variables E to be existentially quantified, and a renaming substitution η, and produces an MDG $R = \mathsf{RelP}(\{P_i\}_{1 \leq i \leq n}, E, \eta)$ such that

$$\models R \Leftrightarrow \left(\left((\exists E)(\bigwedge_{1 \leq i \leq n} P_i) \right) \cdot \eta \right).$$

The algorithm computes the conjunction of the P_i, existentially quantifies the variables in E, and applies the renaming substitution η, all in one pass. For $1 \leq i < j \leq n$, V_i and V_j must not have any abstract variables in common, otherwise the conjunction cannot be computed, because, in general, there is no MDG logically equivalent to the conjunction.

Let us determine the type of the MDG R computed by the algorithm. (It will be useful in Section 2.3.3.) The result of only computing the conjunction would be an MDG of type

$$((\bigcup_{1 \leq i \leq n} U_i) \setminus (\bigcup_{1 \leq i \leq n} V_i)) \rightarrow (\bigcup_{1 \leq i \leq n} V_i).$$

The set E of variables to be existentially quantified must be a subset of $(\bigcup_{1 \leq i \leq n} V_i)$. The result of only computing conjunction and existential quantification would be an MDG of type

$$((\bigcup_{1 \leq i \leq n} U_i) \setminus (\bigcup_{1 \leq i \leq n} V_i)) \rightarrow ((\bigcup_{1 \leq i \leq n} V_i) \setminus E).$$

The domain of η must be a subset of $((\bigcup_{1 \leq i \leq n} V_i) \setminus E)$, and η must preserve the custom symbol order when applied to the set

$$((\bigcup_{1 \leq i \leq n} U_i) \setminus (\bigcup_{1 \leq i \leq n} V_i)) \cup ((\bigcup_{1 \leq i \leq n} V_i) \setminus E).$$

The type of the result R is

$$((\bigcup_{1 \leq i \leq n} U_i) \setminus (\bigcup_{1 \leq i \leq n} V_i)) \rightarrow (((\bigcup_{1 \leq i \leq n} V_i) \setminus E) \cdot \eta).$$

Pruning by subsumption. It takes as inputs two MDGs P and Q of types $U \to V_1$ and $U \to V_2$ respectively, where U contains only abstract variables that do not participate in the custom symbol ordering, and produces an MDG $R = \mathsf{PbyS}(P, Q)$ of type $U \to V_1$ derivable from P by *pruning* (i.e. by removing some of the paths and reducing the resulting graph to satisfy the well-formedness conditions) such that

$$\models R \vee (\exists U)Q \Leftrightarrow P \vee (\exists U)Q. \tag{2.3}$$

The paths that are removed from P are *subsumed* by Q [CZSL94], hence the name of the algorithm.

Since R is derivable from P by pruning, after the formulas represented by R and P have been converted to DNF, the disjuncts in the DNF of R are a subset of those in the DNF of P. Hence $\models R \Rightarrow P$. And, from (2.3)), it follows tautologically that $\models P \wedge \neg(\exists U)Q \Rightarrow R$. Thus we have

$$\models (P \wedge \neg(\exists U)Q \Rightarrow R) \wedge (R \Rightarrow P).$$

We can then view R as approximating the logical difference of P and $(\exists U)Q$. In general, there is no MDG logically equivalent to $P \wedge \neg(\exists U)Q$. If R is F, then it follows tautologically from (2.3) that $\models P \Rightarrow (\exists U)Q$.

2.2.3 Implementation. As in ROBDD packages, we use a *reduction table* (also called *unique table*) to maintain MDG canonicity during the computations, and a *results table* (also called *computed table*) to ensure that each distinct computation is performed only once.

Disjunction is straightforward to implement, given that all the arguments have the same set of abstract variables.

The relational product algorithm is more complex. If an abstract variable x has primary occurrences in one of the MDGs to which RelP is applied, and secondary occurrences in another, then the secondary occurrences are replaced with labels of edges that issue from nodes labeled by the primary occurrences. These substitutions are facilitated by condition 6 in the definition of MDG given in Section 2.2.1. Since terms appearing as edge and node labels can be very large in some cases, we implement them as DAGs, using a reduction table to maximize sharing and assign unique identifiers to all the terms and subterms. We use a results table for substitution. Also, with each MDG node, we keep a list of the abstract variables that participate in the custom symbol ordering and occur in the subgraph rooted at the node. Reordering of cross-terms is necessary after substitution, but is localized, since cross-terms with same cross-operator are consecutive along every path.

The PbyS algorithm is also quite complex. As the algorithm is recursively invoked in a top-down traversal, the edges labels and cross-terms of P are matched against those of Q in order to instantiate the secondary variables of Q. The algorithm must take into account the omission of redundant nodes from P. A path π of P is pruned if there exists an MDG M obtained from (the single-path MDG) π by addition of zero or more redundant nodes, such

that, for every path π' of M, there exists an instantiation π'' of a path of Q such that every node-edge pair of π'' is a node-edge pair of π'.

Detailed descriptions of these three algorithms can be found in [CZSL94].

2.2.4 Other algorithms. Given an MDG P, there does not always exist an MDG R such that $\models R \Leftrightarrow (\neg P)$. For example, there exists no such R if P is $x = a$, where x is an abstract variable and a is an abstract generic constant. However, it is straightforward to compute R in the case where all the nodes in P are labeled by concrete variables or cross-terms. We refer to this special case as *concrete negation.* We shall implement a concrete negation algorithm when the need arises.

We have implemented an algorithm that simplifies an MDG by applying a set of conditional rewrite rules involving the abstract function symbols and cross-operators in the vocabulary of the logic. This algorithm will be described elsewhere.

2.3 Abstract State Machines

The presence of uninterpreted symbols in the logic means that we must distinguish between a state machine M and its abstract description D in the logic. A given abstract description D will determine a machine M for every interpretation ψ. For the purpose of hardware verification we are interested only in finite state machines (FSMs). However, an abstract description will represent infinite as well as finite state machines, since abstract sorts admit infinite interpretations. We call *Abstract State Machine* a state machine given by an abstract description in terms of MDGs, or equivalently DFs, as explained below.

2.3.1 Representing sets using MDGs. Let P be an MDG of type $U \rightarrow V$. Then, for a given interpretation ψ, P can be used to represent the set of vectors

$$Set_V^{\psi}(P) = \{\phi \in \Phi_V^{\psi} \mid \psi, \phi \models (\exists U)P\}.$$

In the next section, MDGs will thus be used in this fashion to represent sets of states and sets of output vectors. We shall also see how MDGs can be used to represent relations.

2.3.2 Describing state machines with MDGs. An abstract description of a state machine M is a tuple $D = (X, Y, Z, F_I, F_T, F_O)$, where

X, Y, Z are disjoint sets of variables, viz. the input, state, and output variables respectively. Let η be a one-to-one function that maps each variable y to a distinct variable $\eta(y)$ obtained, for example, by adorning y with a prime. The variables in $Y' = \eta(Y)$ are used as the next-state variables. X, Y and Z must be disjoint from Y'.

Given an interpretation ψ, an input vector of the state machine M represented by D is a ψ-compatible assignment to the set of input variables

X; thus the set of input vectors, or input alphabet, is Φ_X^ψ. Similarly, Φ_Z^ψ is the output alphabet. A state is a ψ-compatible assignment to the set of state variables Y; hence the state space is Φ_Y^ψ. A state ϕ can also be described by an assignment $\phi' = \phi \circ \eta^{-1} \in \Phi_{Y'}^\psi$ to the next state variables.

F_I is an MDG representing the set of initial states, of type $U \to Y$, where U is a set of abstract variables disjoint from $X \cup Y \cup Y' \cup Z$. Typically, F_I is a one-path MDG where each internal node N is labeled by a variable $y \in Y$, and the edge that issues from N is labeled by the symbolic initial value of y, which can be an individual constant, an abstract generic constant, or an abstract variable $u \in U$. It is possible to specify that two data registers have the same value, but that this common value is arbitrary, by using the same u as symbolic initial value of the abstract state variables representing the two registers.

Given an interpretation ψ, a state $\phi \in \Phi_Y^\psi$ is an initial state iff $\psi, \phi \models (\exists U) F_I$. Thus the set of initial states of the state machine M represented by D is

$$S_I = \{\phi \in \Phi_Y^\psi \mid \psi, \phi \models (\exists U) F_I\} = Set_Y^\psi(F_I).$$

F_T is an MDG of type $(X \cup Y) \to Y'$ representing the transition relation.

Given an interpretation ψ, an input vector $\phi \in \Phi_X^\psi$ and a state $\phi' \in \Phi_Y^\psi$, a state $\phi'' \in \Phi_Y^\psi$ is a possible next state iff $\psi, \phi \cup \phi' \cup \phi'' \circ \eta^{-1} \models F_T$. Thus the transition relation of the state machine M represented by D is

$$R_T = \{(\phi, \phi', \phi'') \in \Phi_X^\psi \times \Phi_Y^\psi \times \Phi_Y^\psi \mid \psi, \phi \cup \phi' \cup (\phi'' \circ \eta^{-1}) \models F_T\}.$$

F_O is an MDG of type $(X \cup Y) \to Z$ representing the output relation.

Given an interpretation ψ, the output relation of the state machine M represented by D is

$$R_O = \{(\phi, \phi', \phi'') \in \Phi_X^\psi \times \Phi_Y^\psi \times \Phi_Z^\psi \mid \psi, \phi \cup \phi' \cup \phi'' \models F_O\}.$$

To recapitulate, for every interpretation ψ of the sorts, constants and function symbols of the logic, the abstract description $D = (X, Y, Z, F_I, F_T, F_O)$ represents the state machine $M = (\Phi_X^\psi, \Phi_Y^\psi, \Phi_Z^\psi, S_I, R_T, R_O)$ with input alphabet Φ_X^ψ, state space Φ_Y^ψ, output alphabet Φ_Z^ψ, set of initial states S_I, transition relation R_T, and output relation R_O.

2.3.3 State exploration. Given an abstract state machine description $D = (X, Y, Z, F_I, F_T, F_O)$ we can compute the set of reachable states of a state machine $M = (\Phi_X^\psi, \Phi_Y^\psi, \Phi_Z^\psi, S_I, R_T, R_O)$ represented by D, for any ψ, using the MDG algorithms mentioned above, while at the same time checking that a given condition on the outputs of the machine, the *invariant*, holds in all the reachable states. The invariant is represented by an MDG C of type $W \to Z$, where W is a set of abstract variables disjoint from X, Y, Y', Z and U. (Recall that F_I is of type $V \to Y$.) For a given ψ, an output vector is deemed to satisfy the invariant iff $\psi, \phi \models (\exists W) C$; thus $Set_Z^\psi(C)$ is the set of output vectors that satisfy the invariant.

The procedure, called ReAn for *Reachability Analysis*, is the result of lifting the algorithm given in [CoBM89b] to the realm of abstract types and MDGs. It can be described by the following pseudo-code:

```
1.    ReAn(D, C)
2.        R := F_I; Q := F_I; K := 0;
3.        loop
4.            K := K + 1;
5.            I := Fresh(X, K);
6.            O := RelP({I, Q, F_O}, X ∪ Y, ∅);
7.            P := PbyS(O, C);
8.            if P ≠ F then return failure;
9.            N := RelP({I, Q, F_T}, X ∪ Y, η);
10.           Q := PbyS(N, R);
11.           if Q = F then return success;
12.           R := PbyS(R, Q);
13.           R := Disj(R, Q);
14.       end loop;
15.   end ReAn;
```

In this pseudo-code, I, N, P, Q and R are program variables that take as values MDGs representing sets of states, and O takes as values MDGs representing sets of output vectors. We will identify the program variables and their values in the following explanations when there is no risk of confusion.

Before each loop iteration, R represents the set of reachable states found so far, while Q represents the frontier set, i.e., a subset of $Set_Y^{\psi}(R)$ containing at least all those states that entered $Set_Y^{\psi}(R)$ for the first time in the previous iteration.

In line 5, $\mathsf{Fresh}(X, K)$ constructs a one-path MDG representing a conjunction of equations $x = u$, one for each abstract input variable $x \in X$, where u is a fresh variable from the set of auxiliary abstract variables U. The value of the loop counter K is used to generate the fresh variables. This one-path MDG is assigned to I, which represents the set of input vectors.

In line 6, the relational product operation is used to compute the MDG representing the set of output vectors produced by the states in the frontier set. The resulting MDG is assigned to O. Then, in line 7, the pruning-by-subsumption operation is used to remove from O paths representing output vectors that satisfy the invariant C. The resulting MDG is assigned to P. In line 8, if P is not F, then the procedure stops and reports failure. We have implemented a counterexample facility that can then be invoked to produce a most general symbolic trace leading to a state for which the outputs do not satisfy the invariant. Examples of such a trace can be found in [ZSTC96]. If P is F, then $Set_Z^{\psi}(O) \subseteq Set_Z^{\psi}(C)$, i.e. every output vector produced by a state in the frontier set satisfies the invariant, and the verification procedure continues.

In line 9, the relational product operation is used again, this time to compute the MDG representing the set of states that can be reached in one state from the frontier set. Note that the MDG Q representing the frontier set is of type $U \rightarrow Y$, the MDG I representing the set of input vectors is of type $U \rightarrow X$, and the MDG F_T representing the transition relation is of type $(X \cup Y) \rightarrow Y'$. The result of taking the conjunction of these three MDGs would be of type $U \rightarrow (X \cup Y \cup Y')$, the result of subsequently removing the variables in $X \cup Y$ by existential quantification would be of type $U \rightarrow Y'$, and the result of subsequently applying the renaming substitution η would be of type $U \rightarrow Y$. The RelP operation performs these three operations in one pass, and assigns the resulting MDG of type $U \rightarrow Y$ to N.

Lines 10 and 11 check whether $Set_Y^{\psi}(N) \subseteq Set_Y^{\psi}(R)$ by the same method used in lines 7 and 8 to check whether $Set_Z^{\psi}(O) \subseteq Set_Z^{\psi}(C)$. If this is indeed the case, then every state reachable from the frontier set was already in $Set_Y^{\psi}(R)$. The fixpoint has been reached and R represents all the reachable states. Therefore, the procedure terminates and reports success. Otherwise the MDG assigned to Q in line 10 represents the new frontier set.

Line 12 simplifies R by removing from it any paths that are subsumed by Q, using PbyS. There may be such paths because Q was not computed earlier as an exact difference. Then line 13 computes the new value of R by taking the disjunction of R and Q, which represents the set of states $Set_Y^{\psi}(R) \cup Set_Y^{\psi}(Q)$, and assigning it to R.

In the general case, this procedure may not terminate and may produce false negatives. These limitations are discusses below, in Section 4.2.3 and Section 4.2.4 respectively.

3. Modeling Hardware with MDGs

A circuit is described at the RT level as a collection of components interconnected by nets that carry signals. Each signal is represented by a variable. Variables denoting control signals have concrete sorts, while variables denoting data values have abstract sorts. We show how various kinds of components can be represented by MDGs through the following examples. The parser in our MDG tools automatically transforms a component predefined in our Prolog-style MDG–HDL [ZhBo95] into its MDG representation.

- Gates: For gates, the input and output signals are always of Boolean sort. Figure 3.1(a) and Figure 3.1(b) show an OR gate and its MDG representation for a particular ordering of the variables. Boolean MDGs are essentially the same as ROBDDs.
- Multiplexer: For a two-way multiplexer as shown in Figure 3.2(a), we may have different MDGs depending on the signals being multiplexed. There is a very compact MDG (Figure 3.2(b)) if x_1, x_2 and y are all of an abstract

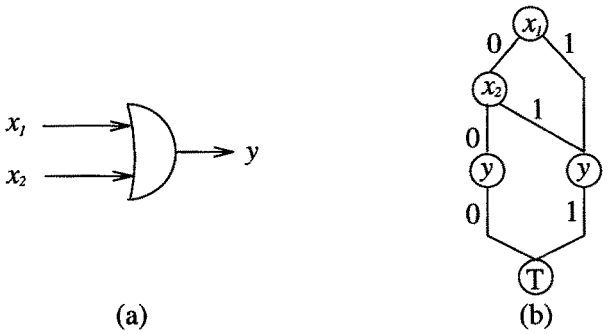

Fig. 3.1. The MDG for an OR gate.

sort. If x_1, x_2 and y are of a concrete sort with enumeration $\{c_i\}_{1 \leq i \leq m}$, then c_i are enumerated in the MDG as shown in Figure 3.2(c).

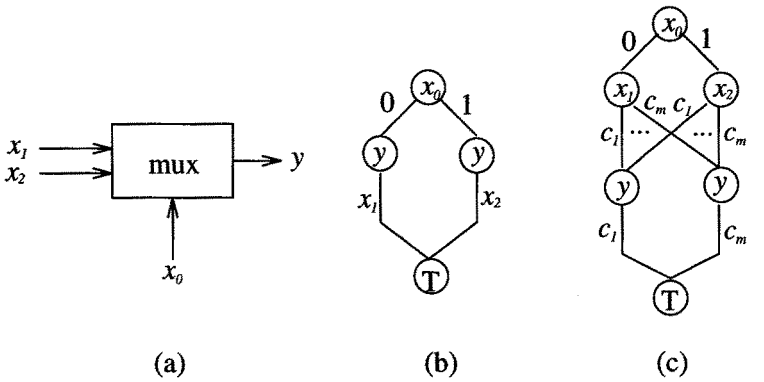

Fig. 3.2. The MDG for a multiplexer.

- Registers: Figure 3.3(a) and Figure 3.3(b) show a register r and its MDG when x and y are of an abstract sort. The variable y' denotes the next state of the register. If x and y are of a concrete sort with enumeration $\{c_i\}_{1 \leq i \leq m}$, we also have to enumerate c_i in the MDG as shown in Figure 3.3(c).
- Control operation: Figure 3.4(a) shows a comparator that produces a control signal y from two data inputs x_1 and x_2. Both x_1 and x_2 are variables of abstract sort while y is a Boolean variable. An uninterpreted cross-operator *eq* is used to denote the functionality of the comparator. If the meaning of *eq* matters, rewrite rules, such as $eq(x, x) \rightarrow 1$ should be used. An MDG of the comparator is shown in Figure 3.4(b).
- Data operation: Data operations are viewed as black boxes and are represented by uninterpreted function symbols. Figure 3.5(a) shows the ALU of the Tamarack-3 microprocessor [Joyc90]. The variables x_1, x_2 and y

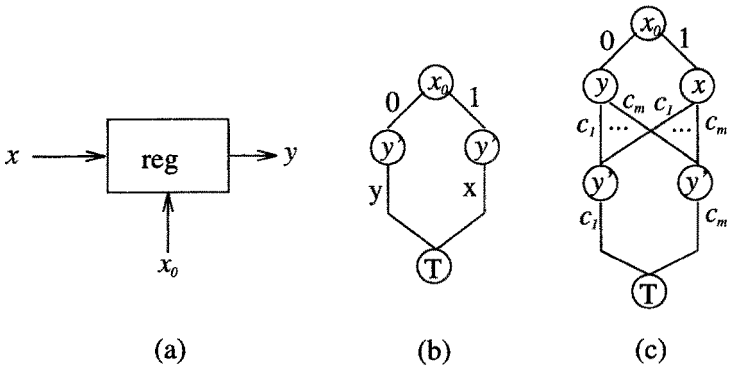

Fig. 3.3. The MDG for a register.

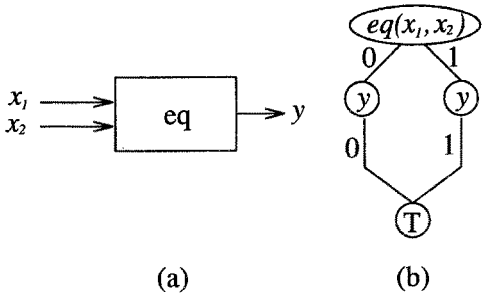

Fig. 3.4. The MDG for a comparator.

representing the data inputs and the output are of an abstract sort, while the variable x_0 representing the control input is of a concrete sort with the enumeration $\{0, 1, 2, 3\}$. Depending on the value of x_0, the ALU can add, subtract, increment, or produce *zero*. The operations are represented by symbols *add*, *sub* and *inc*. The symbol *zero* is a generic constant. The corresponding MDG shown in Figure 3.5(b) is quite compact.

Generally speaking, the behavior of a functional block involving data operations can be described by a directed formula (DF). The DF can then be transformed into an MDG by (i) creating an MDG for each atomic formula; (ii) for a disjunct of DF, conjuncting all the MDGs of its atomic formulas; and (iii) disjuncting all the MDGs representing the disjuncts.

Besides structural descriptions, MDG-HDL can also be used for the description of behavioral specifications. A behavioral description is given by high-level constructs as ITE (If-Then-Else) formulas, CASE formulas or tabular representations. The tabular construct is similar to a truth table but allows first-order terms in rows.

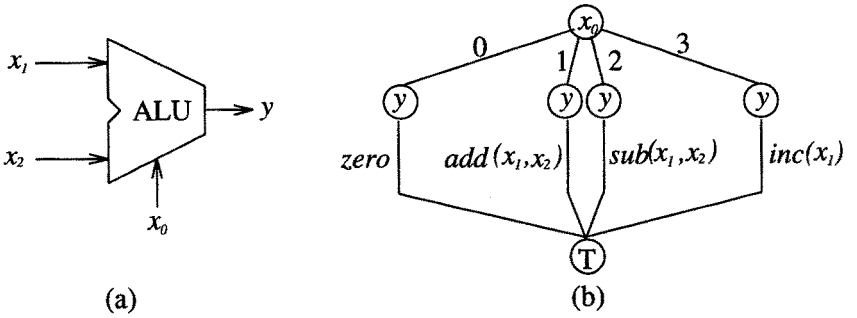

Fig. 3.5. The MDG for an ALU.

4. MDG-based Verification Techniques

We implemented in Prolog an MDG package including algorithms for disjunction, relational product (image computation), pruning by subsumption, and rewriting. We developed a reachability analysis algorithm (abstract implicit enumeration), and provided applications for hardware verification such as combinational circuits verification, safety property checking and equivalence checking of two abstract state machines. The latter two are based on the reachability analysis.

In the following sections, we detail the above applications to hardware verification.

4.1 Combinational Circuits

For *combinational verification*, we take advantage of the fact that MDGs are a canonical representation; we can thus lift the corresponding OBDD technique. Given two combinational circuits to be compared, we compute for each of them an MDG representing its input-output relation by combining the MDGs of the components of the circuit using the relational product operation. The canonicity of MDGs tells us that comparing the functionality of two combinational circuits reduces to computing the MDGs representing their input/output relations. If the two circuits have the same functionality, the two MDGs must represent logically equivalent formulas, and hence they must be isomorphic. By the use of a reduction table in the MDG package, this amounts to checking whether the two MDGs have the same Identification number (ID), a constant-time operation.

Functional comparison of two combinational circuits can also be accomplished using partitioned input/output relations. Instead of computing a single MDG for each circuit it is possible to compute a separate MDG for each output of the circuit. These separate MDGs may be much smaller than a monolithic MDG involving all the outputs. We then check whether the corresponding individual MDGs in the two partitioned relations have the same IDs.

The same technique can be used to compare two sequential circuits when a one-to-one correspondence between their registers exists and is known: it then suffices to compare the combinational parts of the sequential circuits.

4.2 Sequential Circuits

4.2.1 Safety property and equivalence checking. The safety property checking is based on the reachability analysis procedure. Given a state machine M and an invariant C, we check if C holds in all the reachable states of M.

One application of the safety property checking is the *behavioral equivalence* (or input-output equivalence) checking of two sequential circuits. To verify that two machines produce the same sequence of outputs for every sequence of inputs, we feed the same inputs to the two circuits, i.e., we form the product state machine. Then, we perform reachability analysis on the parallel composition using an invariant that asserts the equality of the corresponding outputs in all the reachable states. For machines at different time scales, it is possible to synchronize them first if they have cyclic behavior. Thereafter we can perform reachability analysis on the product machine as usual. This technique can be used for the verification of non-pipelined microprocessor implementations against their instruction-set architecture specifications.

An invariant condition is specified by a combinational circuit whose output signals are named by the variables that occur in the condition. By convention, an assignment of values to those variables satisfies the condition if and only if the outputs of the combinational circuit take those values for *some* assignment of values to the inputs. An MDG representing the invariant is obtained from the MDG representing the functionality of the combinational circuit by existentially quantifying the concrete inputs. The variables representing abstract inputs are left in the graph as implicitly quantified secondary variables. For example, the combinational circuit of Figure 4.1(a), a simple fork, may yield different MDGs depending on the sort of the signals. If x and y are of *bool* sort, then u is existentially quantified and we get the MDG as shown in Figure 4.1(b) which simply represents $x = y$. If x and y are of an abstract sort, then we get an MDG as shown in Figure 4.1(c) which represents the formula $x = u \wedge y = u$. Taking the secondary variable u to be existentially quantified, the invariant becomes $(\exists u)(x = u \wedge y = u)$ which is logically equivalent to $x = y$.

Pruning-by-subsumption is used to check that the invariant is satisfied for the states in each frontier set. If we want to check the equality of two outputs x and y in an output MDG O, we just prune O against Inv which is the same MDG as Figure 4.1(b). This technique makes it possible to state the equality of two abstract signals without having recourse to a cross-operator eq and the rewrite rule $eq(x, x) \rightarrow 1$.

4.2.2 Simple microprocessors. The *instruction set architecture* of a microprocessor is the specification of the effect that each instruction is intended

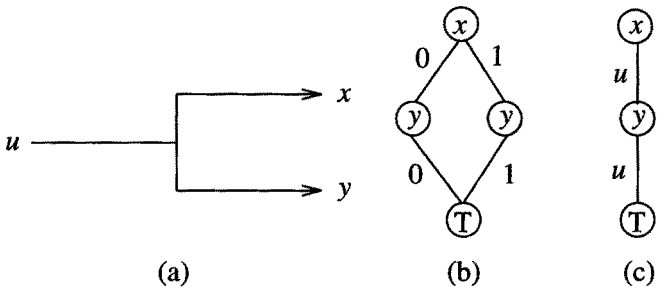

Fig. 4.1. Representation of the invariant $x = y$.

to have on the programmer's model which consists of the visible registers and memory. To verify a microprocessor against its instruction set architecture is to verify that the execution of every instruction has the intended effect.

The control FSM of a microprocessor has a distinguished *ready* state that is the starting point of instruction execution. When the control state is kept in a microprogram counter, the *ready* state is typically $mpc = 0$. We say that the microprocessor itself is in a ready state when the control FSM is in its *ready* state. Precisely stated, the problem is to verify two properties of the circuit C consisting of processor and memory, i.e., that (i) if s_1 and s_2 are consecutive reachable ready states, the visible portion of state s_1 is related to the visible portion of state s_2 as prescribed by the architecture for the executed instruction, and (ii) from every reachable ready state, a ready state is eventually reached again. Currently, we can verify the safety property (i), while property (ii) can be verified if the maximum number of clock cycles before a ready state to be reached from any reachable state is known (i.e., the liveness property is thus converted to a safety property).

To verify (i) we compare C with an ideal state machine C' whose state is the visible state of C and where each transition corresponds to the execution of an instruction as specified by the architecture. We refer to C and C' as the *implementation* and *specification*, respectively. C' is synchronized with C by a *ready* signal extracted from C: when *ready*=1 the specified transition takes place, otherwise C' remains in the same state. We perform reachability analysis on the synchronized composition of the implementation and the specification, checking an invariant that asserts the equality of the visible state in C and C' when *ready*=1. This amounts to verifying (i).

Figure 4.2(a) shows a circuit representing an invariant that asserts the equality of x and y, but only when $mpc = 0$. It is assumed that *mpc* and *u_mpc* have a concrete sort with enumeration $\{0, \ldots, m\}$. The MDG of Figure 4.2(b) is obtained from the circuit by existentially quantifying the concrete input *u_mpc*. The formula which it represents after existentially quantifying the secondary variables u, u_1, u_2 is logically equivalent to

$$mpc = 0 \Rightarrow x = y.$$

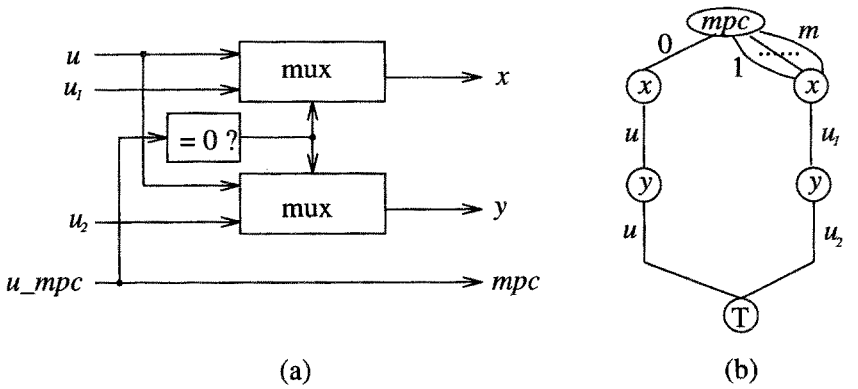

Fig. 4.2. Representation of the invariant "$x = y$ if $mpc = 0$"

4.2.3 Non-termination Problem and Initial State Generalization. There are cases where the set of reachable states is not representable by a finite MDG of type $W \rightarrow Y$, and in such cases the reachability analysis procedures will not terminate. For example, consider a microprocessor having a program counter whose initial value is 0, denoted by a generic constant *zero* of abstract sort. An instruction that does not change the flow of control increments the program counter; assume that an abstract function symbol *inc* is used to represent this. An MDG P_k of type $W \rightarrow Y$ representing the set of states reachable in up to k steps must have at least k disjuncts (state descriptions), containing the equations $y_{\mathrm{pc}} = zero$, $y_{\mathrm{pc}} = inc(zero)$, $y_{\mathrm{pc}} = inc(inc(zero))$, ..., $y_{\mathrm{pc}} = inc^k(zero)$. A DF representing all the reachable states would require an infinite number of disjuncts, for k $\rightarrow \infty$.

In some cases non-termination can be avoided by generalizing the set of initial states so as to obtain a larger set of reachable states that is representable by a finite MDG, while still satisfying the condition to be verified. An important case in which this method is applicable is that of simple microprocessors and similar circuits that exhibit a cyclic behavior. When comparing two state machines derived from two implementations of a processor, or from an implementation and a specification, the initial state of the product machine can be arbitrary, subject only to two constraints: (i) each machine's control state is the one where the instruction cycle begins, and (ii) the corresponding visible registers in both machines have the same initial values. Then the set of reachable states usually has a finite representation because, informally speaking, after an instruction has been executed the product machine goes to a state that is a special case of this initial state. In the case discussed above, non-termination would be avoided by letting the value of the program counter be represented by a variable rather than a constant, which would allow the subsumption check to succeed. This method is referred to as

initial state generalization. We discuss the non-termination problem in more detail in [CZSL94, ZSTC96] and propose several other solutions.

4.2.4 False Negatives. During reachability analysis, it is possible that the invariant holds for the intended interpretation ψ_0 but not for all ψ. The abstract verification will then fail even though the interpreted state machine satisfies the invariant, a false negative result. Yet, when data operations are viewed as *black boxes*, the invariant is expected to hold for every ψ; hence, if the reachability analysis returns "failure", there must be an error in the design. In this sense we say that the verification method is applicable to designs where the data operations are viewed as black boxes.

RTL designs generated by high-level synthesis are usually of this form. This is because high-level synthesis algorithms schedule and allocate data operations without being concerned with the specific nature of the operations.

Another example of well-behaved circuits are processors. A general purpose processor provides data operations for use by the programs running on the processor. It is the programs, not the processor, that make use of the operations. The data operations can be therefore be viewed as black boxes when specifying and verifying the processor. Thus, the class of processor-like circuits is well suited to the above techniques, both from the point of view of termination and from the point of view of false negatives.

We do not know at present whether the problem of verifying that a certain condition holds is decidable when using abstract sorts, completely uninterpreted function symbols and abstract descriptions of state machines.

5. Verification of Benchmark Circuits

In this section we discuss the results of applying abstract implicit state enumeration to three synchronous circuits from the IFIP benchmark suite [Krop94b, Krop94a]. They are the Arbiter, the Greatest Common Divisor (GCD) and the Filter. All the experiments were performed on a SPARC station 20, using our MDG package implemented in Quintus Prolog Version 3.2. The execution times, memory and the number of nodes generated are shown in Table 5.1.

The circuit of the GCD benchmark that we implemented is generic. We used in the datapath abstract signals of type *wordn* to model generic words. The complete circuit is composed of 29 basic components and has a total of 8 state variables. Beside the implementation description, we provided a behavioral specification at the RT level using tabular expressions and abstract state machines. We verified the GCD circuit by checking its equivalence to the behavioral specification. The verification holds for generic words of arbitrary width.

The benchmark FILTER corresponds to the concrete example described in [Krop94a]. It has 5 input values (n = 5) and 3 stages (k = 3). Each stage

Table 5.1. Statistics from benchmark verification

Benchmark	CPU time (in sec)	Memory (in MB)	# MDG Nodes generated
GCD (n-bit)	1.93	1.48	432
FILTER	0.94	0.5	213
Arbiter (4-bit)	1.3	1.4	832
Arbiter (8-bit)	5.0	2.7	2862
Arbiter (16-bit)	4595.0	101.4	202752

is composed of seven components. The circuit has 22 basic components and 12 abstract state variables. Similarly as in the GCD example, we provided a behavioral specification of the FILTER which we checked for equivalence with the the circuit implementation.

The implementation used for the Arbiter appears in [McMi93a]. We constructed 4, 8 and 16 bit versions of this synchronous circuit at the Boolean level (all signals are of a concrete sort). Each cell of the arbiter contains two control registers and a set of logic gates for a total of 10 components per cell. For example, the 16 bit arbiter has 163 basic components and 32 concrete state variables. For each version, we verified two safety properties (properties 1 and 3 in [Krop94a]) which reflect the correct arbitration behavior (Table 5.1 includes the statistics of checking the conjunction of these two properties). Checking liveness properties (property 2 in [Krop94a]) is not currently supported by our MDG tools.

We also experimented with a number of other IFIP benchmark circuits [Krop94a], including the Traffic Light Controller, Adder, Min_Max, Tamarack-3, Multiplier, Divided, and Associative Memory.

For asynchronous circuits, such as Single Pulser and Black-Jack Dealer [Krop94b], we need to verify liveness properties that we cannot do for the moment.

6. Fairisle ATM Switch Fabric: A Case Study in Verification using MDGs

In this section we present a case study of formal verification of the Fairisle 4×4 ATM (Asynchronous Transfer Mode) switch fabric using MDGs. The device is in use for real applications in the Cambridge Fairisle network [Curz94], designed at the Computer Laboratory of the University of Cambridge.

Using a hierarchical approach, we first verified the original gate-level implementation of the switch fabric against an RTL implementation, then we verified the RTL implementation against a behavioral specification given as an abstract state machine (ASM). We thus obtained complete verification

from a high-level behavior down to the gate level. We also verified some specific invariants that reflect the behavior of the fabric in its real operating environment.

6.1 The Fairisle ATM Switch Fabric

The 4×4 Fairisle switch consists of three types of components: the input port controllers, the output port controllers and the switch fabric, as shown in Figure 6.1. It switches ATM cells from the input ports to the output ports. A cell consists of a fixed number of bytes.

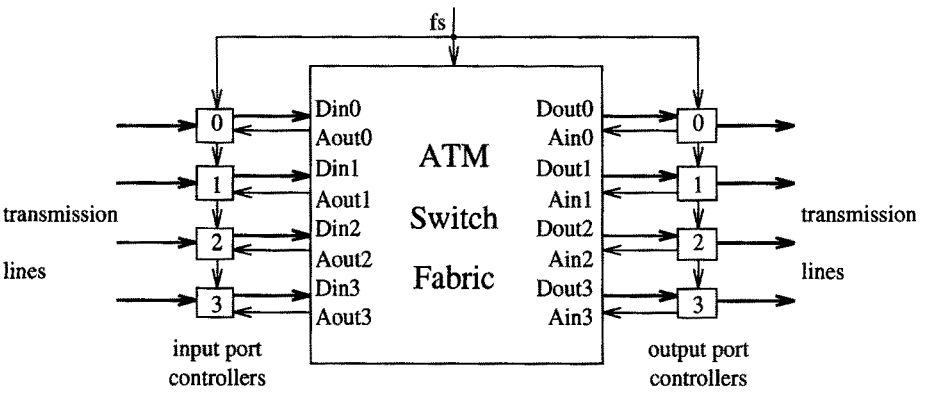

Fig. 6.1. The Fairisle ATM switch

The behavior of the switch is cyclical. In each cycle or frame, the input port controllers synchronize incoming data cells, append control information in the front of the cells in the routing tag (Figure 6.2), and send them to the fabric. The fabric waits for cells to arrive, strips off the tags, arbitrates between cells destined to the same port, sends successful cells to the appropriate output port controllers, and passes acknowledgments from the output port controllers to the input port controllers.

If different port controllers inject cells destined for the same output port controller (as indicated by the route bits in the tag) into the fabric at the same time, then only one will succeed. The others must retry later. The routing tag also includes priority information (priority bit) that is used by the fabric for arbitration which takes place in two stages. High priority cells are given precedence before the remaining cells. The choice within both priorities is made on a round-robin basis. The input controllers are informed of whether their cells were successful using acknowledgment signals. The fabric sends a negative acknowledgment to the unsuccessful input ports, but passes the acknowledgment from the requested output port to the successful input port. The port controllers and the switch fabric all use the same clock, hence bytes

are received synchronously on all links. They also use a higher-level cell frame clock – the frame start signal fs. It ensures that the port controllers inject data cells into the fabric synchronously so that the routing tags arrive at the same time. If no input port raises the active bit throughout the frame then the frame is inactive – no cells are processed. Otherwise it is active.

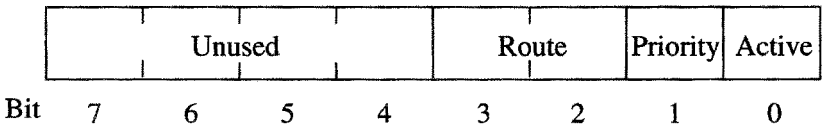

Fig. 6.2. The routing tag (header) of a Fairisle ATM cell

As shown in Figure 6.3, the inputs to the fabric consist of the cell data lines, the acknowledgments that pass in the reverse direction, and the frame start signal fs which is the only external control signal. The outputs consist of the switched data, and the switched and modified acknowledgment signals. The switch fabric is composed of an arbitration unit, an acknowledgment unit and a dataswitch unit. The arbitration unit reads the routing tags, makes arbitration decisions when two or more cells are destined for the same output port, passes the result to the other modules using grant signals, and controls the timing of the other units using output disable signals. The dataswitch performs the actual switching of data from an input port to an output port according to the most recent arbitration decision. The acknowledgment unit passes appropriate acknowledgment signals to the input ports. Negative acknowledgments are sent until arbitration is completed.

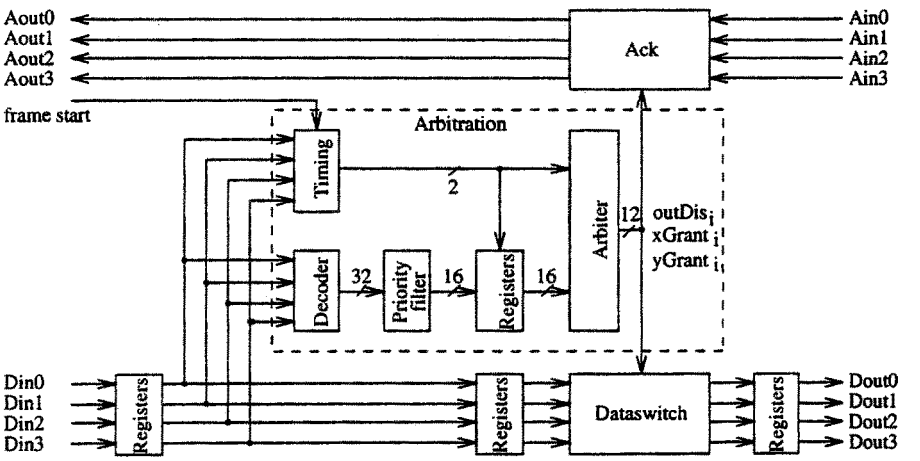

Fig. 6.3. The block diagram of the Fairisle ATM switch fabric

All the design units are repeatedly subdivided until eventually the logic gate level is reached, providing a hierarchy of components. The design has a total of 441 logic gates with two or more inputs and flip-flops.

6.2 MDG Models

6.2.1 Gate and RT Implementations. We described the implementation of the fabric at the gate and the RT levels. In the former case, we directly translated the original Qudos HDL description [Curz94] into our MDG–HDL using the same set of components.

Based on the gate-level description, we produced an RTL implementation by describing the dataswitch using abstract multiplexors instead of logic gates. Here, the data signals $Din_i/Dout_i(i = 0, 1, 2, 3)$ are modeled as n-bit words and are assigned an abstract sort *wordn*. The control fields contained in the cell header, i.e., active, priority and route fields, are extracted from the abstract data signals using cross-operators (Figure 6.4). The ASM model is thus obtained by compiling the abstract description of the RTL implementation.

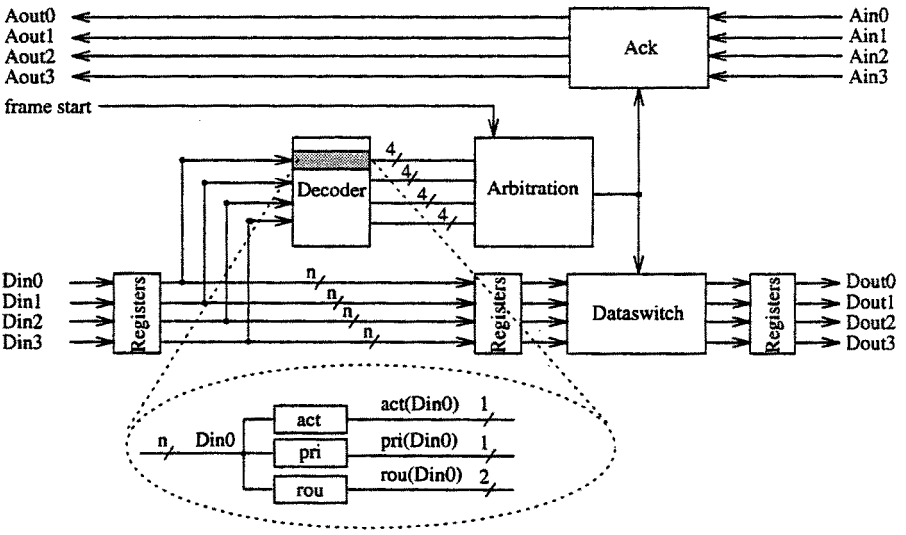

Fig. 6.4. The abstraction model of the switch fabric

6.2.2 Behavioral Specification. The specification of the switch fabric was developed in two forms: a high-level behavioral state machine and a set of invariants reflecting the essential behavior of the switch. The former is described in this section, the latter is discussed in Section 6.3.2.

Starting from a set of timing-diagrams describing the expected behavior of the switch fabric, we derived a complete specification in the form of an abstract state machine. This specification was developed independently of the actual hardware design and includes no restrictions on the frame and cell lengths, and the word width. It reflects the complete behavior of the fabric under the assumption that the environment maintains certain timing constraints on the arrival of the frame start signal and the cell headers.

To verify the RTL implementation against the ASM of the behavioral specification, we make the corresponding input/output signals to be of the same sort and use the same function symbols to extract the control information (active, priority and route fields) from the header.

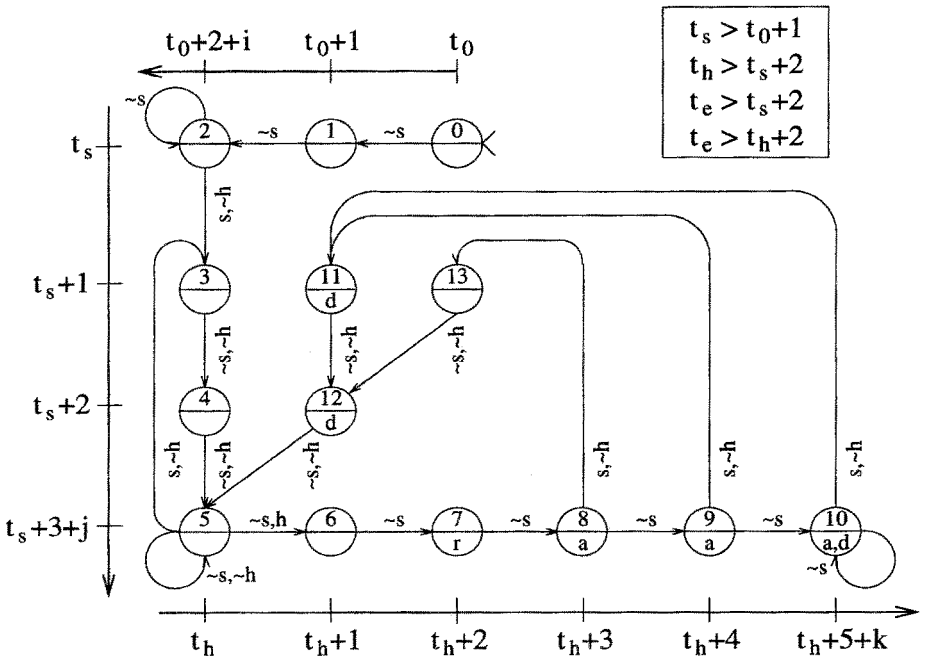

Fig. 6.5. The ASM behavioral specification of the ATM

A schematic representation of the ASM of the 4 by 4 fabric is shown in Figure 6.5. The symbols t_0, t_s, t_h and t_e represent the initial time, the time of arrival of the frame start signal, the time of arrival of the routing bytes and the time of the end of a frame, respectively. There are 14 states: States 0, 1 and 2 along the time axis t_0 describe the initial behavior of the switch fabric. States 2, 3, 4 and 5 along the time axis t_s describe the behavior of the switch on the arrival of the fs signal. States 6 to 13 along the time axis t_h describe the behavior of the switch fabric after the arrival of the headers. The waiting loops in states 2, 5 and 10 are shown by the non-zero natural numbers i, j

and k, respectively. Figure 6.5 also includes many metasymbols used to keep the presentation simple. For instance, the symbols s and h denote the arrival of frame start fs and of the routing tag (header), respectively.

Inside a circle, the symbols r, a and d indicate various operations that take place in the states: round-robin arbitration, the output of acknowledgments and the output of data, respectively. The absence of those symbols means that there is no computation and the default value is output. The operations are defined by separate state machines. We omit their descriptions here since there is nothing special about them and they are quite long to present. What needs to be mentioned, however, is the sort definition of variables. The data signals Din_i and $Dout_i$ (i=[0..3]) are defined as an abstract sort *wordn*. The acknowledgement signals Ain_i and $Aout_i$ (i=[0..3]) are of sort *bool*. More details can be found in [LTZS96].

6.3 Verification

6.3.1 Equivalence Checking. For verifying the equivalence of the gate-level implementation and the abstract (RTL) hardware model, the abstract n-bit words were instantiated to 8 bits using *uninterpreted* functions which decode abstract data to Boolean data [TZSC96]. Equivalence checking of the RTL implementation and the behavioral specification was performed for an arbitrary word width n and any frame size and cell length [LTZS96].

By combining the above two verification steps, we hierarchically obtained a complete verification of the switch fabric from the high-level behavior down to the gate-level implementation. The experimental results on a SPARC station 10 are recapitulated in Table 6.1, including the CPU time, memory usage and the number of MDG nodes generated.

Table 6.1. The ATM experimental results for equivalence checking

Verifications	Time (Sec)	Mem (MB)	#Nodes
Gate-Level to RT-Level	183	22	183300
RT-Level to Beh.-Level	2920	150	320556

No errors were discovered in the implementation. For the sake of experimentation, however, we injected several errors into the implementation: (1) We exchanged the inputs to the JK Flip-Flop that produces the output disable signal. This prevented the circuit from resetting. (2) We used the priority information of the input port 0 to control the input port 2. (3) We used an AND gate instead of an OR gate within the acknowledgment unit producing a faulty $Aout_0$ signal. These three errors were detected by verifying the RTL implementation model against the behavioral specification. Table 6.2 shows

the experimental results including times for reachability analysis on the specification and counterexample, memory usage and the number of MDG nodes generated.

Table 6.2. Verification of some faulty implementations of the ATM

Experiments	Reachability (Sec)	Counterex. (Sec)	Mem (MB)	#Nodes
Error 1	11	9	1	2462
Error 2	850	450	120	150904
Error 3	600	400	105	147339

6.3.2 Invariant Checking. Although the ASM describes the complete behavior of the switch fabric, we partially validated (in an early stage of the project) the fabric implementation by property checking. This is useful as it gives quick confidence check at low cost. Sample properties are correct circuit reset and correct data routing.

We consider the behavior of the fabric when operating in the intended real Fairisle switch environment. The switch generates frame start signal fs (Figure 6.3) at every $64th$ clock cycle. Initially, it should wait at least 2 clock cycles to let the fabric reset before it can generate the first fs signal. The header of a cell is generated at the $9th$ clock cycles after fs is set.

This cyclic behavior can be simulated as an *environment state machine* having 68 states as shown in Figure 6.6. The machine generates the frame

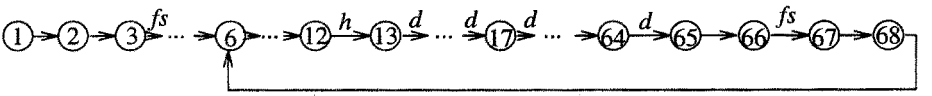

Fig. 6.6. The environment state machine of the ATM

start signal fs, the headers h and the data d in the states as indicated in the figure. Normally, d is a fresh abstract variable representing data in the cell; and h can be instantiated according to the property to be verified. We also assume that the firstfs signal is generated at the 3rd clock cycle after power on. States 1 to 5 are related to the initialization of the fabric. States 6 to 68 represent the cyclic behavior of the fabric, where one cycle corresponds to one frame. With this diagram, we can map the time points to states in a similar way as we explained in the preceding section. In this case, $t_s = 3$ or 66; $t_h = 12$; and $t_e = 66$. Then, e.g., $t_h + 5$ to $t_e + 2$ are essentially the states between 17 and 68 when the remaining data of the cell following the header are switched to the output port. It can be checked that this state machine is

an instance of the general timing state machine (Figure 6.5) with cell length of 53 and frame size of 64.

Below, we list properties that we verified and give their ITE expressions. The state variable c of the environment state machine is of a concrete sort having the enumeration [1..68].

$P1$: From $t_s + 3$ to $t_h + 4$, the default value is put on the data output port 0.
if $(c \in [6..16])$ **then** $Dout_0 =$ zero **else** don't-care.

$P2$: From $t_s + 1$ to $t_h + 2$, the default value is put on the acknowledgment output port 0.
if $(c \in [4..14, 67, 68])$ **then** $Aout_0 = 0$ **else** don't-care.

$P3$: From $t_h + 5$ to $t_e + 2$, if input port 0 chooses output port 0 with the priority bit set in the header and no other input port has its priority bit set, then the value on $Dout_0$ will be Din'_0 which is the input of Din_0 four clock cycles earlier.
if $(c \in [17..68]) \wedge (priority[0..3] = [1, 0, 0, 0]) \wedge (route[0] = 0)$ **then** $Dout_0 = Din'_0$ **else** don't-care. ($priority[0..3]$ are the priority bits from all the input ports and route[0] represents the routing bits for input port 0)

$P4$: From $t_h + 3$ to t_e, if input port 0 chooses output port 0 with the priority bit set in the routing tag, and no other input port has its priority bit set, the value on $Aout_0$ will be the input of Ain_0.
if $(c \in [15..66]) \wedge (priority[0..3] = [1, 0, 0, 0]) \wedge (route[0] = 0)$ **then** $Aout_0 = Ain_0$ **else** don't-care.

These invariants can be easily represented using MDGs. To verify them, we compose the fabric with the environment state machine as shown in Figure 6.7. As there is a 4-clock-cycle delay for the cells to reach the output ports, a delay circuit is used to remember the input values that are to be compared with the outputs. Hence, we can state the properties in terms of the equality between Din'_0 and $Dout_0$ (e.g. P3). Combining these machines (dashed frame in Figure 6.7), we obtain the required platform for checking the invariants. The above properties easily detected the three introduced design errors. The experimental results are reported in Table 6.3.

Table 6.3. Verification of properties $P1$ – $P4$

Verifications	Time (Sec)	Mem (MB)	#Nodes
P1	202	15	30295
P2	183	15	30356
P3	143	14	27995
P4	201	15	33001
Error 1 by P1	49	8	16119
Error 2 by P3	77	11	24001
Error 3 by P4	82	11	24274

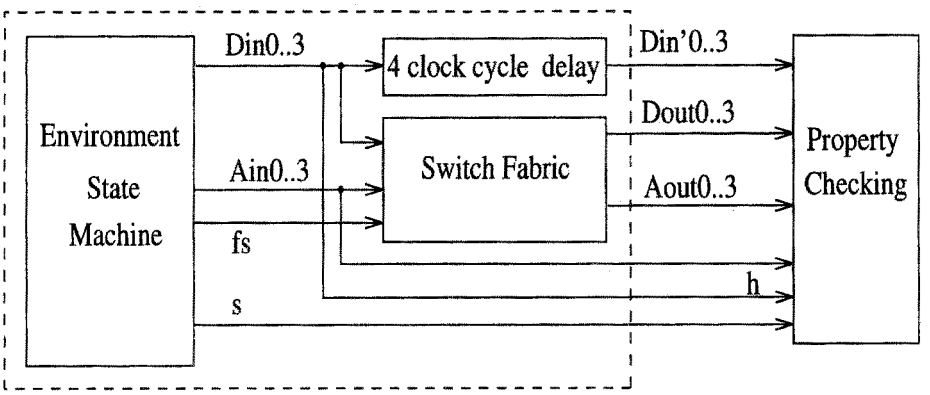

Fig. 6.7. The composite state machine for invariant checking on the ATM switch fabric

7. Conclusions and Future Work

We presented a verification methodology that makes it possible to verify sequential circuits automatically at the RT level, using abstract sorts and uninterpreted function symbols. It is based on a new kind of decision graphs, Multiway Decision Graphs (MDG). This approach allows data signals to be represented by a single variable of abstract sort rather than by 32 or 64 Boolean variables. We also described a set of algorithms for manipulating MDGs, and shown how they can be used for combinational verification, invariant and behavioral equivalence checking of sequential circuits using abstract implicit state enumeration.

Our work has shown that the use of abstract sorts for formal verification can produce interesting results by raising the level of abstraction at which the problem is stated. The contribution of MDGs beyond the use of abstract sorts is that they allow to use at a higher level of abstraction some of the ROBDD techniques that have been successful at the Boolean level.

We provided experimental results for a set of benchmarks obtained using a prototype MDG package implemented in Prolog. We demonstrated that formal verification of a 4×4 ATM switch fabric can be conducted automatically using the MDG tools.

There are many opportunities for further work in formal verification using the MDG representation of first-order formulas:

- We are developing model checking algorithms for an appropriate first-order temporal logic.
- We are exploring the links between theorem proving systems and MDG-based tools. There are two possible approaches to their integration. (1) We can embed the model checker as a specialized decision procedure in a theorem prover. This makes the theorem proving software more efficient and powerful. (2) MDG-based model checking can proceed and complete

successfully when output checking and state set inclusion can be decided by rewriting and syntactic matching. When this is not possible, we could prove the specific subgoal using a theorem prover. For systems containing complex structures, such as loops, we have to combine model checking, inductive proofs, and rewriting to accomplish the verification task effectively.

Acknowledgement. We would like to thank Nancy Boulerice, Ying Xu and Dan Voicu for carrying out the experimental work on benchmarks. The work was partially supported by an NSERC Canada Strategic Grant No. STR0167079 and the experiments were carried out on workstations provided by the Canadian Microelectronics Corporation.