Preprocessing for Property Checking of Sequential Circuits on the Register Transfer Level

Vorverarbeitung für die Überprüfung von Eigenschaften Sequentieller Schaltungen auf der Register-Transfer-Ebene

Vom Fachbereich Elektrotechnik und Informationstechnik
der Universität Kaiserslautern
zur Verleihung des akademischen Grades
Doktor der Ingenieurwissenschaften (Dr.-Ing.)
genehmigte Dissertation

von

Dipl.-Inf. Raik Brinkmann

geboren in Brehna

Tag der Einreichung:	2. Juli 2003
Tag der mündlichen Prüfung:	21. November 2003
Dekan des Fachbereichs:	Prof. Dr.-Ing. Gerhard Huth
Vorsitzender der Prüfungskommission:	Prof. Dr.-Ing. Norbert Wehn
Erster Berichterstatter:	Prof. Dr.-Ing. Wolfgang Kunz
Zweiter Berichterstatter:	Prof. Dr. Wolfram Büttner

D386, Universität Kaiserslautern, 2003

This work is dedicated to Christiane.

Zusammenfassung

Motivation

Integrierte digitale Schaltkreise (ICs) realisieren seit langem schnell und kompakt Schlüsselfunktionen, besonders in der Informations-, Unterhaltungs- und Kommunikationstechnik, sowie in der Fertigungs- und Automobilindustrie. Der Trend, immer größere vollständige Systeme auf einem einzigen IC zu integrieren, ist in allen Gebieten zu beobachten. Die Verfügbarkeit von solchen 'Systems on a Chip' (SoC) wird einerseits durch die fortschreitende Erhöhung der Integrationsdichten der verwendeten Halbleiter bestimmt, andererseits von der Möglichkeit vorentwickelte Schaltungsteile, als sog. 'Intellectual-Property-' (IP-) Blöcke, zu integrieren.

Sprünge in der technologischen Entwicklung von einer Halbleitergeneration zur nächsten, führen bei traditionellen Hableiterprodukten, wie Speichern, zu Phasen des Nachfragemangels einerseits und Versorgungsengpässen andererseits. Diese treten als sog. 'Halbleiter-Zyklen' hervor, welche die Industrie ökonomisch belasten. Da die Nachfrage nach Systemen weniger stark schwankt als beispielsweise bei Speicherprodukten, versucht die Halbleiterindustrie durch Fokussierung auf SoC, diese Zyklen zu überbrücken.

Aus technologischer Perspektive können bereits 20 Millionen Gatter und mehr auf einem IC integriert werden. Es sind bereits ICs mit einer Milliarde Gattern in Sicht. Die Fähigkeiten, sequentielle Schaltungen solcher Größenordnung im Zeit- und Kostenrahmen zu entwerfen, begrenzt aber das Potential der SoC-Technik. Um diese Produktivitätslücke zu schließen, muß für die verwendeten Blöcke dabei extreme Qualität erzielt werden.

Angenommen ein SoC besteht aus sechs Blöcken, deren korrekte Funktion nach gegenwärtiger Methodik rudimentär durch Simulation überprüft wurde. Da sich Fehler in den Komponenten im Gesamtsystem fortpflanzen, ist dessen korrekte Funktion wahrscheinlich nicht gewährleistet (Sind beispielsweise die Blöcke mit einer Wahrscheinlichkeit von 90% korrekt, ergeben sich nur etwa 50% für das System.). Dieses potentielle Fehlverhalten muss dann im Systemtest mühsam ausgeschlossen werden. Abgesehen davon, dass Fehler in den Blöcken durch Systemsimulation oft schwierig zu finden sind, sind die Sytemtests sehr zeitaufwendig, da sie sich nicht auf die Sytemaspekte der Funktion konzentrieren können. Sollen nun Systeme mit 50 oder gar 100 Blöcken integriert werden, ist dies nur noch möglich, wenn sie den höchstmöglichen Qualitätsansprüchen bezüglich ihrer Funktion genügen. Andernfalls wäre die Systemverifikation und damit die gesamte Entwicklung zu kostspielig.

Heute wird die Funktion eines Blocks zumeist durch Simulation überprüft. Dies kann jedoch aufgrund der inherenten Unvollständigkeit des Ansatzes praktisch nicht zur erforderlichen Qualität führen. Um nur einfache Qualitätsstandards zu gewährleisten, werden zwischen 60 – 80% des gesamten Entwicklungsaufwandes auf die Simulation verwendet. Die Produktivitätslücke ist daher vor allem eine Verifikationskrise.

Formale Methoden bieten die einzige Möglichkeit, die erforderliche Qualität zu erzielen, da sie vollständig sind. Insbesondere die automatische Überprüfung von formal spezifizierten Eigenschaften durch 'Bounded-Model-Checking' hat in der letzten Zeit an Bedeutung gewonnen, da es die automatische Verifikation ganzer Blöcke erlaubt. Dabei bieten die verwendeten Eigenschaften eine orthogonale Sicht auf die Funktion der Schaltung. Diese sind jedem Hardware-Entwickler zugänglich, da sie kompakt und, im Gegensatz zu anderen formalen Methoden, leicht verständlich sind.

Dieser Technik sind jedoch aus Komplexitätsgründen noch nicht alle Klassen von sequentiellen Schaltungen zugänglich. Dies betrifft insbesondere größere Schaltungen mit regulärer Struktur, die beispielsweise durch die Verwendung von Speichern und Bussystemen hervor tritt. Ziel der vorliegenden Arbeit ist es, die Anwendbarkeit von Bounded-Model-Checking auf diese Klasse von Schaltungen auszudehnen.

Überblick

Die Arbeit beschreibt einen neuen Ansatz zur Vereinfachung von aussagenlogischen Beweisproblemen, wie sie beim Bounded-Model-Cheking (BMC) üblicherweise auftreten. Dieser Ansatz ermöglicht es, bisher unlösbare Beweisaufgaben aus dem BMC in der industriellen Praxis zu lösen, sowie bereits lösbare Beweisaufgaben wesentlich effizienter zu bewältigen.

Eine Beweisaufgabe besteht darin, zu überprüfen, ob eine Bool'sche Funktion konstant wahr ist, bzw. ob eine diese Funktion repräsentierende aussagenlogische Formel allgemein gültig ist. Derartige Überprüfungen lassen sich zwar mit Techniken wie Bool'scher Erfüllbarkeit (SAT) oder Binären Entscheidungsdiagrammen (BDDs) im Prinzip lösen, doch stoßen diese Verfahren in der industriellen Praxis regelmäßig an Grenzen. Ziel der Arbeit ist es, diese Grenzen zu erweitern. Dies wird erreicht, indem das Beweisproblem vor einem Beweisversuch auf eine neuartige Weise reduziert wird.

Die Reduktion findet auf einer dem Ursprungsproblem nahen Repräsentation der Bool'schen Funktion als sog. Bitvektor-Term statt, da dies eine besonders effiziente und weitreichende Reduktion begünstigt. Die Reduktion basiert einerseits darauf, die Term-Repräsentation zu vereinfachen, andererseits auf einer speziellen Zerlegung des Beweisproblems in Co-Faktoren der repräsentierten Funktion. Diese Co-Faktoren werden jeweils einzeln vereinfacht und wenn möglich durch automatische Symmetriebetrachtungen zu Äquivalenzklassen zusammengefasst. In vielen Fällen gelingt das Zusammenfassen zu einer einzigen Äquivalenzklasse. Es ist im Folgenden ausreichend, aus jeder Äquivalenzklasse einen beliebigen Repräsentanten auszuwählen und für diesen den Beweis durchzuführen. Alternativ können die Repräsentanten ihrerseits weiter zerlegt, vereinfacht und zu neuen Äquivalenzklassen zusammengefasst werden. Auf diese Weise werden nacheinander Variable eliminiert, und die resultierenden Beweisprobleme massiv vereinfacht.

Reduktion und Zusammenfassung basieren auf Heuristiken und Semi-Entscheidungsverfahren für Äquivalenz und Permutationsäquivalenz von Bitvektortermen bzw. den repräsentierten Bool'schen Funktionen. Diese Verfahren beruhen ihrerseits auf Term-Rewriting- und Graphenalgorithmen.

Die vorgestellte Reduktionsmethode ermöglicht bereits in einer prototypischen Implementierung die Demonstration messbarer Verbesserungen der Gesamtperformanz für BMC in praktisch relevanten Beispielen um ein bis zwei Größenordnungen. Weitere Steigerung durch Verbesserung der Implementierung und der Heuristiken scheinen durchaus möglich, bedürfen jedoch eines gewissen Implementierungsaufwandes, der den Rahmen dieser Arbeit gesprengt hätte.

Der entwickelte Ansatz sowie die erzielten Resultate werden im Folgenden beschrieben.

Problemkonstruktion auf der Register-Transfer-Ebene

In dieser Arbeit wird die Methode des sog. Bounded-Interval-Model-Checking, die eine Abwandlung des bekannten BMC-Ansatzes darstellt, zur Verifikation sequentieller (digitaler) Schaltungen benutzt. Sequentielle (digitale) Schaltungen werden zur Überprüfung von Eigenschaften vermittels BIMC durch Mealy-Maschinen modelliert. Mealy-Maschinen sind endliche Automaten mit Ausgabe- und Zustandsübergangsfunktion über ein Ein- und Ausgabealphabet. Eine bestimmte Teilmenge der Zustände des Automaten wird als Menge der Startzustände angesehen. Eine Schaltung mit n Eingängen, m Ausgängen und k Latches hat dann als Ein- und Ausgabealphabet die Mengen $\mathbb{B}^n$ und $\mathbb{B}^m$, sowie die Zustandsmenge $\mathbb{B}^k$. Die Bool'schen Ausgabe- und Zustandsübergangsfunktionen $f_O : \mathbb{B}^n \times \mathbb{B}^k \to \mathbb{B}^m$ und $f_S : \mathbb{B}^n \times \mathbb{B}^k \to \mathbb{B}^k$ werden gewöhnlich als Bool'sche Terme über Bool'sche Ein-, Ausgabe- und Zustandsvariablen $\overline{\imath} = (i_1, \ldots, i_n)$, $\overline{o} = (o_1, \ldots, o_m)$ und $\overline{s} = (s_1, \ldots, s_k)$ repräsentiert. Basierend auf einem diskreten Zeitmodell ist das sequentielle Verhalten der Mealy-Maschine durch die Bedingungen $\forall \mathsf{t} \in \mathbb{N}_0 : \overline{o}_\mathsf{t} = f_O(\overline{\imath}_\mathsf{t}, \overline{s}_\mathsf{t})$, und $\forall \mathsf{t} \in \mathbb{N}_0 : \overline{s}_{\mathsf{t}+1} = f_S(\overline{\imath}_\mathsf{t}, \overline{s}_\mathsf{t})$ definiert, wobei eine Abarbeitungsfolge am Zeitpunkt $\mathsf{t} = 0$ mit einem Startzustand beginnt.

Für die Überprüfung der korrekten Funktion werden zunächst Eigenschaften der Form $p(\overline{\imath}_{\mathsf{t}+0}, \ldots, \overline{\imath}_{\mathsf{t}+\mathsf{n}}, \overline{o}_{\mathsf{t}+0}, \ldots, \overline{o}_{\mathsf{t}+\mathsf{n}}, \overline{s}_{\mathsf{t}+0}, \ldots, \overline{s}_{\mathsf{t}+\mathsf{n}})$ über beschränkte Zeitintervalle $[\mathsf{t}, \mathsf{t}+\mathsf{n}]$ durch eine Eigenschaftssprache spezifiziert, die einer Hardware-Beschreibungssprache (HDL) ähnlich ist. Durch rekursives Einsetzen obiger Ausgabe- und Zustandsübergangsbedingungen wird eine Bool'sche Funktion $p'(\overline{\imath}_{\mathsf{t}+0}, \ldots, \overline{\imath}_{\mathsf{t}+\mathsf{n}}, \overline{s}_{\mathsf{t}+0})$ konstruiert, so dass aus $p' = 1$ die Gültigkeit der Eigenschaft folgt. Dies kann automatisch durch Bool'sche Beweiser überprüft werden, z.B. durch einen SAT-Solver und den Beweis, dass es keine Belegung der Variablen gibt, so dass $\neg p' = 0$ gilt. Gilt eine Eigenschaft nicht, so wird mit der gefundenen Belegung ein Gegenbeispiel produziert.

Die Repräsentation der obigen Ausgabe- und Zustandsübergangsfunktionen wird in der Regel direkt durch 'Gate-Level-Synthese' aus einer vorhandenen Schaltungsbeschreibung in VHDL oder Verilog-HDL erzeugt. Durch den Syntheseprozess geht ein Großteil der dort verfügbaren strukturellen Information für eine spätere Strukturanalyse und eine darauf basierende Reduktion des Problems $p' = 1$ verloren.

Um die Struktur für eine spätere Reduktion zu erhalten, werden obige f_O, f_S und p stattdessen als Bitvektor-Funktionen aufgefasst und als Bitvektor-Terme repräsentiert. Ein Bitvektor ist, als Vektor von Bits, durch seine Breite charakterisiert. Jeder Bitvektor-Variablen $\mathbf{x}$ wird deshalb eine Breite n zugeordnet, die ihren Wertebereich $\mathbb{B}^n$ kennzeichnet. Eine Bitvektor-Funktion $f : \mathbb{B}^{n_1} \times \mathbb{B}^{n_k} \to \mathbb{B}^n$ ist im Wesentlichen ein Vektor von n Bool'schen Funktionen, $f_1 : \mathbb{B}^{n_1+\cdots+n_k} \to \mathbb{B}, \ldots, f_n : \mathbb{B}^{n_1+\cdots+n_k} \to \mathbb{B}$. Bitvektor-Terme werden zu deren Repräsentation über Bitvektor-Variablen unter Verwendung von Bitvektor-Operationen aufgebaut. Formal lassen sich solche Konstruktionen leicht durch mehrsortige Algebren behandeln. Die in dieser Arbeit vorgestellte Bitvektor-Algebra ist durch eine kleine aber funktional vollständige Menge von Gleichungen definiert, was später das gleichungsbasierte Schließen erlaubt. Syntaktisch läßt sie sich leicht durch definierte Operationen erweitern, wodurch sich die Spezifika vershiedener HDLs erfassen lassen. Diese Erweiterung erlaubt die Behandlung von Melay-Maschinen und Eigenschaften direkt auf der Register-Transfer-Ebene. In den daraus konstruierten BIMC-Problemen bleibt deren Struktur für die Reduktion erhalten. Dabei wird die im HDL-Modell natürlich vorhandene Gruppierung von Bits in Bitvektoren zugrunde gelegt.

Ansätze zur Problemreduktion

Abstrahierend von der Vorstellung zeitbehafteter Variablen, besteht ein BIMC-Problem darin, zu überprüfen, ob eine gegebene Bool'sche Funktion f konstant 1 ist. Bekanntermaßen ist dieses Problem Co-NP-vollständig. Zu seiner Lösung können entweder Suchverfahren wie SAT, oder kanonische Repräsentationen wie BDDs herangezogen werden. Diese benötigen im schlechtesten Fall für die Überprüfung entweder exponentiell viel Zeit (SAT) oder Speicher (BDDs), abhängig von der Zahl der involvierten Variablen und der Art der Eingangsrepräsentation, z.B. als AND-Inverter-Graph, oder Bitvektor-Term. Ziel einer Reduktion ist es, die Eingangsrepräsentation geeignet zu vereinfachen, z.B. durch Elimination von Variablen.

Durch die Ausnutzung algebraischer Zusammenhänge, die in Form von Gleichungen gegeben sind, ist es für Bitvektor-Terme leichter möglich, größere zusammenhängende Subfunktionen zu vereinfachen, als dies für Bool'sche Funktionen möglich ist. Zum Beispiel lässt sich der Bitvektor-Term $(\mathbf{a}_{[n]} * \mathbf{b}_{[n]}) - (\mathbf{b}_{[n]} * \mathbf{a}_{[n]}))$ leicht zu $\mathbf{0}_{[n]}$ vereinfachen. Dabei ist dies für $n = 1$ genauso effizient wie für $n = 32$ (für Bool'sche Terme ist dies nicht der Fall). Dazu werden Verfahren für Bool'schen Terme, wie Konstanten-Propagierung und Normalisierung, geeignet erweitert. Diese Erweiterung basiert auf mehrsortigen Term-Ersetzungssystemen und der Darstellung von Bitvektor-Termen als gerichtete azyklische Graphen (DAG) mit geeignet bezeichneten Knoten, wobei alle isomorphen Untergraphen zusammengefasst werden (sharing). Diese Repräsentation erlaubt eine Speicher sparende und effiziente Implementierung von Term-Rewriting-Systemen auf einem DV-System. Einfache Normalisierungen, wie in obigem Beispiel, sind in den Term-Erzeugungsprozess integriert, ähnlich wie dies für BDDs üblich ist. Nach dieser Vorverarbeitung lässt sich ein BIMC-Problem p', das als Bitvektor-Term repräsentiert ist, leicht in eine Bool'sche Funktion f übersetzen, für die auf übliche Weise $f = 1$ überprüft wird.

Die oben beschriebene Art der Vorverarbeitung ist notwendig, aber nicht ausreichend, um reguläre Schaltungen effizienter zu behandeln. Dies liegt im Wesentlichen an Symmetrien, die aus der regulären Struktur des Problems (nicht notwendigerweise seiner Repräsentation), resultieren.

Sei beispielsweise $f : \mathbb{B}^3 \to \mathbb{B}$ mit $f(x, y, z) = ite(x, y \to (y \vee z), z \to (z \vee y))$ eine Funktion für die $f = 1$ zu überprüfen ist. Dann sind $f|_{x=1} = y \to (y \vee z)$ and $f|_{x=0} = z \to (z \vee y)$ symmetrische Unterprobleme, die exakt auf die gleiche Weise gelöst werden könnten.

Gegenwärtige Verfahren sind jedoch nur sehr begrenzt in der Lage, solche Zusammenhänge automatisch zu erkennen und auszunutzen. Dies gilt insbesondere, wenn die Unterprobleme über verschiedene Variablenmengen aufgebaut werden, oder mehrere Variablen belegt werden müssen, bevor die Symmetrie zu Tage tritt. Die hier vorgestellte Symmetriereduktion basiert im Prinzip auf folgender Beobachtung:

Offensichtlich sind sich in obigem Beispiel $f|_{x=1}$ und $f|_{x=0}$ sehr ähnlich. Tatsächlich haben die Terme $y \to (y \vee z)$ und $z \to (z \vee y)$ eine identische Struktur, nur die Variablen tragen verschiedene Namen. Unter der Variablenpermutation π, die x fixiert, und y und z vertauscht, gilt $f|_{x=0} = \pi(f|_{x=1})$. Wir nennen 0 und 1 im Folgenden symmetrische Werte für die Variable x, falls es eine solche Permutation π gibt. Offensichtlich ist die Frage, ob $f = 1$ gilt, in diesem Fall unabhängig von der Belegung von x, d.h., $h = 1$ g.d.w. $h|_{x=0} = 1$ g.d.w. $h|_{x=1} = 1$. Folglich kann die Variable x mit einem beliebigen Wert belegt, und dabei der Suchraum halbiert werden. (Trotz symmetrischer Werte für

x, ist die Funktion f im Allgemeinen jedoch nicht unabhängig von x, wie das Beispiel $f = ite(x, y, z)$ zeigt. Andererseits folgt aus $f = 1$ bereits, dass für alle Variablen, die in der Repräsentation von f vorkommen, die Werte symmetrisch sind.)

Durch sukzessive Anwendung der obigen Reduktion auf alle Variablen, ergibt sich entweder, dass f konstant ist ($f = 1$ oder $f = 0$), oder dass es eine Variable x' gibt, für die kein solches π existiert. Da somit f von x' abhängt, also nicht konstant ist, kann jedoch $f = 1$ nicht gelten. Offensichtlich ist der praktische Nutzen obiger Reduktion abhängig davon, wie leicht sich entscheiden lässt, ob zwei Werte für eine Variable symmetrisch sind oder nicht. Die identische Struktur der Teilprobleme im obigen Beispiel legt nahe, dass solche Situationen automatisch auf syntaktische Weise erkannt werden können. Das Problem ist jedoch im Allgemeinen äquivalent zu der Frage, ob es für zwei Bool'sche Funktionen h und g eine Variablen-Permutation π mit $h = \pi(g)$ gibt. Dieses Problem ist als 'Boolean-Isomorphism-', 'Matching-', 'Latch-Correspondence-', oder 'Permutation-Equivalence-Problem' aus der Schaltungsverifikation bekannt. Da es sich dabei wieder um ein Co-NP-vollständiges Problem handelt, kommen unvollständige Verfahren zur Symmetriebestimmung zum Einsatz.

Die Erweiterung der Reduktionsmethode auf Bitvektor-Funktionen führt zum Begriff der Symmetrierelation, die eine Äquivalenzrelation auf den Werten einer Bitvektor-Variablen ist. Die im Folgenden vorgestellten Verfahren zur Symmetriebestimmung sind nicht vollständig, was dazu führt, dass Symmetrien eventuell nicht erkannt werden, d.h., dass die berechneten Symmetrierelationen eventuell feiner als die tatsächlichen sind. Für die Reduktion folgt daraus die Notwendigkeit zur Überprüfung eines Repräsentaten jeder Äquivalenzklasse. Das Konzept der Symmetrierelation erlaubt jedoch die effiziente Kopplung verschiedener Verfahren, was die Qualität der gefundenen Relationen verbessert. Erweiterungen bezüglich der Kombination von Symmetrierelationen über verschiedene Variablen bilden schließlich die Grundlage für einen vollständigen Branch-And-Bound-Algorithmus zur Lösung von BIMC-Problemen mit heuristischer Symmetriereduktion.

Verfahren zur Überprüfung von Permutationsäquivalenz

Die Verfügbarkeit guter Verfahren zur Bestimmung von Permutationsäquivalenz für zwei Bool'sche Funktionen ist von entscheidender Bedeutung für den Erfolg der oben beschriebenen Reduktion. Denn davon hängt ab, ob gute Approximationen der tatsächlichen Symmetrierelation effizient gefunden werden können.

Verschiedene hinreichende Kriterien für Permutationsäquivalenz von zwei Bool'schen Funktionen h und g sind denkbar. Eine Klasse entsteht durch die Beschränkung auf einfache Äquivalenz, d.h. auf $\pi = id$. In dieser Klasse finden sich die bereits bekannten Verfahren zur Entscheidung ob $f = 1$ gilt, da $h = g$ g.d.w. $\neg(h \oplus g) = 1$ sowie die bereits beschriebenen Reduktionen von Bitvektor-Termen durch Term-Ersetzung (Normalisierung). Basierend auf weiteren Mengen von Gleichungen, die jeweils eine Unteralgebra der Bitvektor-Algebra definieren, lassen sich weitere Reduktionsverfahren durch Bitvektor-Term-Ersetzung konstruieren. Verfahren für verschiedene solcher Mengen können durch Nacheinanderausführung zu neuen Verfahren kombiniert werden. Da keine Vollständigkeit verlangt ist, muss für alle Verfahren lediglich deren Termination sicher gestellt werden.

Die andere Klasse bilden Verfahren, bei denen die Menge der Gleichungen stark

eingeschränkt oder sogar leer ist, jedoch eine nichttriviale Variablenpermutation bestimmt wird. Für den Extremfall einer leeren Gleichungsmenge lässt sich das Problem auf Graphenisomorphie zurückführen und damit entscheiden. Zwei Terme, die unter dieser Bedingung gleich sind, heißen syntaktisch symmetrisch. Dieser Ansatz kann für Gleichungen, die nur die Permutation von Argumenten eines Funktionssymbols beschreiben, wie z.B. bei Kommutativität, erweitert werden.

Die Kombination von Verfahren beider Klassen ist möglich und sinnvoll. So führt beispielsweise eine Normalisierung mit anschließender Überprüfung auf Syntaktische Symmetrie oft zum Erfolg.

Schließlich wurden weitere Verfahren entwickelt, die die gleichzeitige Untersuchung von mehr als zwei Termen auf gegenseitige Permutationsäquivalenz erlauben. Dies ist insofern wichtig, als für Bitvektor-Terme in der Regel mehr als zwei Co-Faktoren pro Variable zu untersuchen sind.

Ausblick

Das vorgestellte Reduktionsschema wurde erfolgreich auf bisher unlösbare BIMC-Probleme aus der Industrie angewendet. So konnte beispielsweise die Chache-Funktion eines Embedded-Processor-Core erfolgreich verifiziert werden. Für bereits lösbare Probleme ergab sich eine erhebliche Steigerung der Effizienz.

Jedoch sind noch zahlreiche weitere Verbesserungen möglich. So könnte beispielsweise die prototypische Implementierung von Teilen der Term-Ersetzungssysteme durch automatisches Übersetzen in eine Hochsprache wie C++ wesentlich verbessert werden. Weiterhin scheint es möglich, die Bedingung zur Symmetriereduktion so abzuschwächen, dass die Konstantheit eines Co-Faktors die eines anderen impliziert, nicht jedoch umgekehrt.

Andere Anwendungsgebiete der hier vorgestellten Techniken zur Symmetriebestimmung könnten zum Beispiel im Äquivalenzvergleich von Schaltungen auf der Register-Transfer-Ebene liegen, da es auch hier Permutationsäquivalenz zu entscheiden gilt.

Insgesamt konnte diese Arbeit einen wichtigen Teil der Probleme lösen, die gegenwärtig eine weitere Ausdehnung der Anwendbarkeit von BIMC behindern. Weitere Anstrengungen sind erforderlich. So ist die effiziente Verifikation von Schaltungen mit hohem Anteil arithmetischer Operationen im Datenpfad nach wie vor ein ungelöstes Problem. Die hier vorgestellte Methode zur Problemkonstruktion auf der Register-Transfer-Ebene könnte dazu einen wichtigen Beitrag leisten.

Abstract

As the sustained trend towards integrating more and more functionality into systems on a chip can be observed in all fields, their economic realization is a challenge for the chip making industry. This is, however, barely possible today, as the ability to design and verify such complex systems could not keep up with the rapid technological development. Owing to this productivity gap, a design methodology, mainly using pre designed and pre verifying blocks, is mandatory. The availability of such blocks, meeting the highest possible quality standards, is decisive for its success. Cost-effective, this can only be achieved by formal verification on the block-level, namely by checking properties, ranging over finite intervals of time. As this verification approach is based on constructing and solving Boolean equivalence problems, it allows for using backtrack search procedures, such as SAT. Recent improvements of the latter are responsible for its high capacity.

Still, the verification of some classes of hardware designs, enjoying regular substructures or complex arithmetic data paths, is difficult and often intractable. For regular designs, this is mainly due to individual treatment of symmetrical parts of the search space by backtrack search procedures used. One approach to tackle these deficiencies, is to exploit the regular structure for problem reduction on the register transfer level (RTL).

This work describes a new approach for property checking on the RTL, preserving the problem inherent structure for subsequent reduction. The reduction is based on eliminating symmetrical parts from bitvector functions, and hence, from the search space.

Several approaches for symmetry reduction in search problems, based on invariance of a function under permutation of variables, have been previously proposed. Unfortunately, our investigations did not reveal this kind of symmetry in relevant cases. Instead, we propose a reduction based on symmetrical values, as we encounter them much more frequently in our industrial examples.

Let f be a Boolean function. The values 0 and 1 are symmetrical values for a variable x in f iff there is a variable permutation π of the variables of f, fixing x, such that $f|_{x=0} = \pi(f|_{x=1})$. Then the question whether $f = 1$ holds is independent from this variable, and it can be removed. By iterative application of this approach to all variables of f, they are either all removed, leaving $f = 1$ or $f = 0$ trivially, or there is a variable x' with no such π. The latter leads to the conclusion that $f = 1$ does not hold, as we found a counter-example either with $x' = 0$, or $x' = 1$.

Extending this basic idea to vectors of variables, allows to elevate it to the RTL. There, self similarities in the function representation, resulting from the regular structure preserved, can be exploited, and as a consequence, symmetrical bitvector values can be found syntactically. In particular, bitvector term-rewriting techniques, isomorphism procedures for specially manipulated term graphs, and combinations thereof, are proposed.

This approach dramatically reduces the computational effort needed for functional verification on the block-level and, in particular, for the important problem class of regular designs. It allows the verification of industrial designs previously intractable.

The main contributions of this work are in providing a framework for dealing with bitvector functions algebraically, a concise description of bounded model checking on the register transfer level, as well as new reduction techniques and new approaches for finding and exploiting symmetrical values in bitvector functions.

Acknowledgments

Afoot and light-hearted I take the open road,
Healthy, free, the world before me,
The long brown path before me leading wherever I choose.
(...).

– Walt Whitman – Song of the Open Road –

I would like to express my sincere thanks to my supervisors for reviewing this thesis. Special thanks go to Wolfram Büttner for his comments and guidance, as well as for his dedication to the entire Formal Verification Group at Siemens and Infineon. I thank Wolfgang Kunz and Otto Mayer for their kind cooperation throughout the work on this thesis, which was often performed across the miles.

I wish to thank the Siemens Corporation for their financial support and for allowing me access to their equipment. Many thanks to Infineon Technologies for granting me permission to take time off work during completion of this thesis and I am indebted to both for the great opportunity to spend a year and a half at their subsidiaries in San Jose, California whilst doing my research.

I owe gratitude to numerous people with whom I have worked. I would like to express special thanks to Reinhard Enders, Martin Müller-Brahms, Karl Stroetmannn, and Peter Warkentin for the stimulating discussions and their endurance in reviewing my thesis. I am also grateful for the joint work with Peer Johannsen and Jörg Lohse during my time in San Jose.

Last, but not least, I would like to express my profound gratitude to Christiane for her love, which has given me strength all the way from the very start of my research to finishing this thesis. I am also deeply grateful to my parents for their support and advice. I am indebted to every one of my friends who has helped me throughout the whole of this time.

Contents

Chapter 1

Introduction

1.1 Hardware Systems

Today, integrated digital circuits are used in a vast number of application fields, most notably computers, consumer electronics, communications, and industry. The trend towards integrating more functionality with higher speed and lower power consumption, can be observed in all these fields. The availability of such Systems on a Chip (SoC) is driven by the integration density of the semiconductors used, plus the ability to utilize pre-designed standard functionalities, i.e. Intellectual Property (IP) blocks.

The increase in chip integration is mainly driven by the demand for ever larger and faster memories. This increase is, however, not continuous. Instead, it is related to the step-wise transitions from one generation of semiconductors to the next. This leads to supply shortages before these transitions, and to demand shortages after these transitions. This interrelation is reflected by the well known 'semiconductor cycles'. During times of short demand, the chip industry regularly faces enormous economic down-turns although the demand for complete systems does not vary as much as the demand for memory chips. The chip industry attempts to bridge demand shortages by focusing on the integration of complete systems on a single chip. Today, from a technological standpoint using 0.11μ technology, entire hardware systems can be integrated onto a single chip comprising as many as 20 million gates. Nonetheless, the SoC approach is barely economical today, since the development costs are still too high, as the following analysis illustrates.

Integration density of semiconductors grows on average at a rate of 66% per year (Moore's Law). This benefits memory products, as they can be easily scaled. However, the ability to design more complex hardware grows only at a mere 21% per year. This leads to the so called 'productivity gap'. The ability to design hardware is basically determined by the ability to validate the functional correctness of hardware designs, as the latter activity absorbs 70% – 80% of the total development effort. Therefore the availability of suitable validation tools is the decisive factor for the economic success of the SoC approach.

The economic realization of SoC is a challenge for the chip making industry today. Owing to the existing productivity gap, a block based design methodology, using up to 80 – 90% pre designed and pre verified blocks, is mandatory. Currently this is the only way of setting up the required large functionalities with economical viability. As will be shown in the following example, the required quality of a SoC cannot be left to system

verification.

Let's assume a SoC design consists of six blocks. According to the current design practice, each block has been validated rudimentarily. After this validation, each block is correct with a probability of, say, 90%. Hence, the whole SoC design is correct with a probability of, at best, as little as $(0.9)^6 \approx 0.5$. The possibility for misbehavior is therefore 50%, which has to be reduced to an acceptable value during system verification. This has to be done along with its actual task, that being to verify system aspects. Simulation is used for system verification but for the simulation of a single block, the surrounding system is largely ballast. Because of this, these simulation runs are to time consuming and test too few functional aspects. Moreover, finding block level errors on system level is often difficult. As is widely known, correcting errors, long after they have been introduced, is expensive. Therefore deficiencies in verification are responsible for the productivity gap.

If systems with 50 to 100 blocks are to be built, blocks with an extremely low tendency towards functional misbehavior are necessary. Each design block has to meet the highest possible quality standards. Otherwise, system verification, and thus the whole design, will be too expensive, since the effects described above are leveraged.

As will be shown, the required level of quality can be reached by formal methods only. This analysis of the needs of hardware design, determines the scientific goal of this thesis.

1.2 Block Level Verification

The availability of blocks with extreme low error probability is a decisive factor for the economical realization of SoCs. The error probability of a block is determined by the functional correctness of its design.

Typically, the functionality of a hardware design is first described manually, using a hardware description language, such as VHDL (cf. [vhd93]) and Verilog (cf. [ver95]). Apart from flaws in the specification or architecture, this manual description of functionality is the basic source for functional misbehavior of hardware designs.

Functional correctness of a design means that it behaves as expected. In the same way as for system verification nowadays, in order to validate the functionality of a design on the block level, simulation (see e.g. [Ber00]) is normally used. First, sequences of input values, along with the expected output values of the design under test, are to be specified. For each step of such a sequence, a simulator calculates output values of the design, depending upon the input values and the internal state of the design. Subsequently, the calculated and the expected output values are compared, and violations are reported. Simulation is a fully automated process and can therefore be used effectively by hardware designers.

Unfortunately this process is far from being exhaustive. The whole set of specified input sequences can cover only a small portion of all imaginable possibilities, even for relatively small designs. Simulation is incomplete, and some critical misbehavior might remain undetected. The required level of correctness cannot be produced by block level simulation. Added to this, specifying the desired behavior by sequences manually, is cumbersome, time consuming, and error-prone in itself. Automatic test generation tools (cf. e.g. [LLR+00, spe]) ease this problem by using symbolic specifications instead of examples. However, apart from mainly targeting system verification and not the block level, the fundamental problem of incompleteness remains.

For these reasons, the desired correctness of design blocks cannot be reached by simulation, but instead only by using formal methods. Formal methods are sophisticated, highly automated methods for mathematical proofs. They provide exhaustive, i.e. complete functional verification.

For formal methods it can be observed that there is a trade-off between the expressiveness of the employed logic and the complexity of the resulting verification task. In the past, the claims of formal methods, with regard to expressiveness of the input languages and the universality of their application, were quite high. However, it was impossible to solve real-world problems with reasonable effort. Over time, these high claims had to be withdrawn more and more, in order to find practical solutions for block size designs.

Theorem proving (see e.g. [BM84, Fit90, NPW02]), for example, can be used for verification of block size designs. However, theorem proving is rarely automatic. It typically requires a lot of specialized knowledge about logic and significant user-interaction. As this knowledge is frequently not available in hardware design teams, and as user interaction is always time consuming, this approach is too expensive. Instead, full automation of the verification process and ease of use are necessary for acceptance, and for productive industrial application.

Property checking of finite state machines w.r.t. temporal formulae is less powerful than theorem proving, although it is fully automatic (cf. [CES83, LP85]). Explicit and symbolic model checking (see e.g. [McM93]) suffer from the well known 'state-space-explosion problem', which leads to performance problems, making them impractical for flat block level verification.

Logically, Bounded Model Checking (BMC) (see [BCCZ99, BCC+99]), as described below, attacks the state-space-explosion problem. Although it is even less powerful than symbolic model checking, it is practical for block level verification of hardware designs. Therefore, BMC is an enabling technology for the SoC approach.

1.3 Bounded Model Checking

Within the BMC-approach, the expected behavior of a design is formally described by a bounded-time temporal specification, which is a set of properties grasping interesting behavior over a short period of time. Even though such properties are not as expressive as general temporal logic formula used in symbolic model checking, they are sufficient for many practical applications. BMC directly employs the cycle-based verification scheme, which is very intuitive to hardware designers.

To verify a property, a so called 'BMC-problem' is generated from the design and a property to be verified. A BMC-problem is represented by a Boolean function f. In order to solve BMC-problems, it is necessary to check whether this Boolean function is constantly 1. If $f = 1$ holds, then the property holds for the design. If $f = 1$ does not hold, then the property does not hold for the design, and a counter-example is generated explaining why the design does not exhibit the behavior specified by the property.

If f was explicitly given by a truth table, the task of deciding whether it is constant would be trivial. However, the size of such tables grows exponentially with the number of variables involved, and as BMC-problems can contain several thousand variables, a BMC-problems are usually represented more compactly as 'Boolean terms'. In general, it

is not trivial to decide whether two Boolean terms represent the same Boolean function, since the Boolean equivalence problem is Co-NP-complete (see e.g. [BRS98]).

Theoretically, this can be solved by computing a normal form, e.g. using binary decision diagrams (cf. [Bry86]). However, ordered binary decision diagrams tend to grow very rapidly (at exponential rate) in size, thus, computation of such decision diagrams for Boolean formulas often ceases to compute due to lack of memory.

For this reason BMC-problems are usually tackled by solving the dual problem, the satisfiability problem for a Boolean formula in conjunctive normal form, using backtrack search algorithms for Boolean satisfiability (SAT) such as variations of the DLL-procedure (cf. [DLL62]), or advanced test pattern generation (ATPG) (see e.g. [FS83]).

Although all known procedures have exponential worst case complexity, they are practical in many cases. They use clever search heuristics, utilizing specifics of hardware designs, which enables the skipping of certain parts of the search space. DLL-based SAT is especially sophisticated in doing so (cf. [MSS96, BS97]), therefore, even though other approaches are possible, BMC-problems are usually translated into conjunctive normal form, and a SAT solver is used to check validity.

Recent advances in SAT solver technology (cf. [MMZ+01]) have pushed the limits of BMC further. This makes the BMC-approach even more attractive and give it its high capacity. Today the complexity limits of BMC allow verification of typical design blocks of control oriented nature. The approach has lately become very popular in the semiconductor industry lately (see e.g. [CFG+01]), and this is mainly due to its high capacity, as well as to the more intuitive way of specifying the desired behavior of a design. However, some important classes of hardware designs, including large regular designs, as well as arithmetic designs, cannot yet be handled robustly.

Regular designs form a large and important class of hardware designs. They are usually constructed by repetitive use of some basic building blocks. Often many of these building blocks are treated uniformly. Take for example the cells of a memory, participants in a communication network, or masters on a bus waiting for a slave device to become available. If this uniformity is reflected by the internal structure of the design, it is called 'regular'.

Regular designs can contain many instances of similar building blocks, such as memories, bus-systems, multiplexers, and arbiters and thus have many state variables. For example, even a small memory with 8 address lines and 16 data lines has at least 2^{4096} states. This often leads to a very large search space for the search procedure and to a high degree of symmetry. It can make BMC impractical, as it often leads to exponential runtime behavior, and consequently to inefficiency.

This thesis is focused on speeding up BMC by exploiting the underlying regularity of design and property. This exploitation depends on the representation of BMC-problems as bitvector terms directly on the register transfer level (RTL), thus preserving the syntactic correlation of single bits in bitvectors and bitvector terms naturally inhabiting HDLs. To receive such a representation, the BMC-approach is lifted to the RTL.

1.4 Preserving Structure

Structure, in form of functional units and their operations, can only be exploited if it is explicitly available in the representation of design and property. Hardware description

languages, in which the functionality of a design is first formulated, operate on the register transfer level. On the register transfer level structure is explicit. It is prevalent in data types and functional units. Bitvectors (vectors of bits) are the prevailing data type. The register transfer level is also called word level, since bitvectors can be viewed as words over the alphabet $\{0, 1\}$. We want to exploit this with the aim of reducing computational effort for solving BMC-problems.

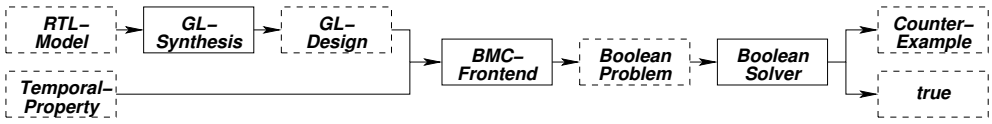


Figure 1.1: Standard BMC-Flow

The standard BMC-flow as depicted in Fig. 1.1, however, destroys much of the structure as the formal representation for a design is constructed in the following way: a register transfer level model of a design, given in some hardware description language, is first translated into a gate-level netlist, using a process called 'gate-level synthesis'. The gate-level netlist is basically a representation of the design, where state variables and transition relations are represented by latches and primitive Boolean operators (gates) like *AND*, *OR* and *NOT*, over single bits. Therefore the gate-level is also called 'bit-level'.

Because the design is flattened into a bit level representation, the syntactic correlation of the single bits, available on the register transfer level, is lost. Therefore, the regular structure of a design is not explicit in its formal representation. In order to exploit the structure of a design, it must be preserved.

If, in contrast to the standard approach, the design is synthesized into a register transfer level representation, a word-level representation of the BMC-problem can be constructed, and the structure is preserved. The resulting word-level representation of the bounded model checking problem is called 'register transfer level bounded model checking problem' (RTL-BMC-problem). An RTL-BMC-flow is illustrated in Fig. 1.2.

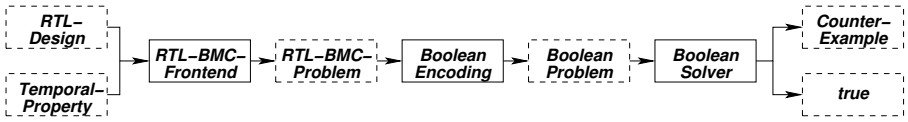


Figure 1.2: RTL-BMC-Flow

1.5 Preprocessing on the Register Transfer Level

The structural information preserved by the RTL-BMC-Flow above can now be used to reduce the size of RTL-BMC-problems before solving them. Such a reduction is of most practical use, if it runs fully automatic and is seamlessly integrated into an RTL-BMC-tool. A possible data flow of such a tool is shown in Fig. 1.3.

Using normalization techniques, based on term rewriting (cf. e.g. [BN98]), is the first step in allowing for more global reductions and normalizations than on the bit-level, e.g.

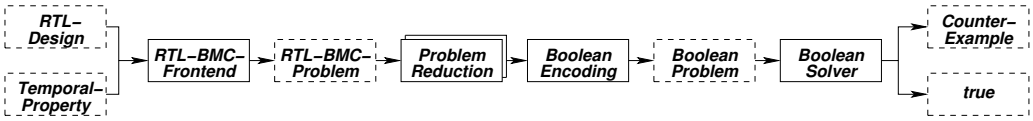


Figure 1.3: RTL-BMC-Flow with Problem Reduction

for arithmetic expressions or memory arrays. This is due to the fact that semantic dependencies within the function representation can be discovered and used syntactically, as bitvector terms are a much more compact representation than Boolean terms. However, doing so requires dealing with bitvector functions algebraically which is much more complex than for Boolean functions owing to the number of operations and different bitwidths of signals. This thesis provides a suitable framework. Using rewriting techniques for bitvector terms developed here it is possible to reduce the size of RTL-BMC-problems efficiently before solving them. These rewrite procedures are also used later for detecting and eliminating symmetries in the search space of RTL-BMC-problems.

Other reduction approaches on the register transfer level, such as uninterpreted functions (see e.g. [HIKB96, VB98, BBCZ98, BGV99, PRSS99]) and automated abstraction techniques (see e.g. [CGL92, HB95, Joh01, MA02, Joh03]), might be applied in conjunction.

After such a reduction has been applied, the RTL-BMC-problem is transformed into a Boolean function which is subsequently checked using standard techniques. Alternatively, solvers operating on the register transfer level, e.g. based on word-level decision diagrams (see e.g. [SBW98, HD99, CKZR02]) or Integer-Linear-Programming (cf [FDK98, ZKC01, BD02], can be applied directly to the RTL-BMC-problem.

1.6 Symmetry Reduction

Analysis of the structure of regular designs has shown that symmetry is an important structural feature that should be exploited to make the BMC-approach even more powerful. This thesis focuses on this aspect. To keep it simple, the basic ideas for symmetry reduction are described on the bit-level below and are extended in the respective chapters.

Self similarities in the structure of a design under verification lead to symmetries in the state-space of the corresponding formal model and thus to symmetries in the search space of the resulting verification problem. Symmetries in search problems, and in particular in BMC-problems, can make the search heuristics ineffective, in that symmetrical parts of the search space are considered individually. The idea of symmetry reduction is to prevent this, i.e. to treat symmetrical parts uniformly.

Several approaches for symmetry reduction in search problems, based on the following intuitive notion of symmetry, have been proposed, (e.g. [BFP86, CGLR96, ARMS02]). A Boolean function f over some variables including x and y is symmetrical in x and y iff f stays invariant under permutation of x and y. Consider for example the function g:

$$\begin{aligned} g : \{0,1\} \times \{0,1\} \times \{0,1\} &\rightarrow \{0,1\} \\ g(x,y,z) &\mapsto (x \wedge y) \vee z \end{aligned}$$

In g, x and y are interchangeable since $\wedge$ is a commutative operation. Hence x and y are

symmetrical variables of g. This kind of symmetry is either exploited by adding symmetry breaking predicates to the search problem (as in [CGLR96, ARMS02]), or by altering the search procedure (see e.g. [BFP86]). The basic variable swapping above has been extended to more complex notions of symmetry in Boolean functions (cf. [ARMS02]). However, as they all reduce to variable permutations, the set of all symmetries of a function forms a group. Often it can be described compactly by a set of generators.

Unfortunately, our investigations did not reveal this kind of symmetry in relevant cases. Instead, we found the following much more frequently in our industrial examples: Let f be a Boolean function in variables X. The values 0 and 1 are symmetrical values for a variable x in f iff there is a renaming (permutation) π of the variables X of f fixing x, such that $f|_{x=0} = \pi(f|_{x=1})$. As an example consider the function h below.

$$\begin{aligned} h : \{0,1\} \times \{0,1\} \times \{0,1\} &\rightarrow \{0,1\} \\ h(x,y,z) &\mapsto ite(x, y \rightarrow (y \vee z), z \rightarrow (z \vee y)) \end{aligned}$$

Now we have $h|_{x=1} = y \rightarrow (y \vee z)$ and $h|_{x=0} = z \rightarrow (z \vee y)$. Obviously $h|_{x=1}$ and $h|_{x=0}$ are very similar. In fact the Boolean terms $y \rightarrow (y \vee z)$ and $z \rightarrow (z \vee y)$ have the same structure. They only differ in the names of variables. Under a variable permutation π, permuting y and z while fixing x, they can be mapped onto each other, i.e. $h|_{x=0} = \pi(h|_{x=1})$. Furthermore, it is easy to see that both $h|_{x=1} = 1$ and $h|_{x=0} = 1$ are tautological. Therefore, with $\pi' = id$, $h|_{x=0} = \pi'(h|_{x=1})$ follows immediately. However, in general (e.g. when h is a BMC-problem) it is a computationally complex task to decide whether such π exists. This problem is known as 'Boolean isomorphism' (see e.g. [AT96]) or 'permutation equivalence problem'.

If symmetrical values of the function f to be checked for constantness are known, they can be exploited as follows. If the values 0 and 1 are symmetrical for some variable x in some Boolean function f, then the question whether $f = 1$ holds, is independent from this variable. Therefore it can be removed, i.e. we can conclude that $f = 1$ holds iff $f|_{x=1} = 1$, and there exists some π fixing x such that $f|_{x=0} = \pi(f|_{x=1})$. For h above we found such a permutation π, permuting y and z while fixing x, and can conclude: $h = 1$ iff $h|_{x=0} = 1$ iff $h|_{x=1} = 1$. Thus x can be eliminated from the problem and, either $h|_{x=1} = 1$ or $h|_{x=0} = 1$, are to be considered further.

By subsequent application of this approach to all variables of f, they are either all removed, leaving $f = 1$ or $f = 0$ trivially, or there is variable x' for which there is no such π. The latter case leads ultimately to the conclusion that f is not constant.

Since the Boolean permutation equivalence problem is Co-NP-complete (cf. [BRS98]), the naive approach sketched above is doomed to fail in practice. Therefore fast and reliably heuristics as described in this thesis are needed in order to make it work. They rely on combinations of rewrite heuristics and graph isomorphism procedures operating on the register transfer level. First results presented in this thesis were reported in [Bri01].

The idea of exploiting symmetry to improve verification is not new. For example, symmetry in the transition relation of state machines can be used to reduce the state space for explicit and symbolic model checking by calculating a quotient structure, i.e. a smaller automaton (cf. e.g. [CFJ93, ES93, MHB98]). These methods require the user to specify the symmetries.

In [ID93] a method is proposed where a syntactic feature called 'scalar set variables' is used to specify symmetries in the state space of a design. At first sight, this reduction

scheme has some similarities with the one proposed here, as variables are replaced with concrete values. However, as these scalar set variables can be used in only a few syntactic contexts in order to maintain the structural symmetry of the design, the approach can not be used within the context of Verilog- and VHDL-based verification.

In contrast, our approach has many advantages. Since it is defined semantically, it does not suffer from syntactic restrictions on the variables containing symmetries. Any possible variable in the function to be checked can be taken as candidate for symmetry reduction. In fact, it allows even much more complex relationships, for example, between values of different variables. At the same time, syntactic self similarities in the design and property can be exploited as efficiently.

An early approach using permutation equivalence for reduction can be found in [Gel59]. Here syntactic symmetry, i.e. identity of formulas under variable permutation, is used. We apply an extension of this syntactic symmetry to many-sorted terms as a criterion for permutation equivalence.

In the context of equational inference, so called 'stratified subterm permutation groups' are computed in order to efficiently deal with symmetries in commutative theories (cf. [AP01]). This notion of symmetry is similar to our approaches for syntactic symmetry modulo commutativity, however different symmetry finding algorithms are used.

In [Moh99] an approach is described using signatures for variables in Boolean functions to identify corresponding pairs in permutation equivalent functions. Limitations of use are described in [MMM95, Moh99]. To our knowledge, extensions for bitvector functions have not yet been developed.

Symmetry breaking, as proposed in [CGLR96], has been applied to reduce the search space for SAT problems (see e.g. [ARMS02]). Although we do not use symmetry breaking, their symmetry detection scheme is similar to ours, in that it uses colored undirected graphs for detecting them. However, instead of of working on bitvector terms, they work on a Boolean formulas in conjunctive normal form.

1.7 Outline of this Thesis

This thesis proposes a novel method for BMC on the register transfer level, along with reduction techniques. These techniques allow for heuristic minimization of the resulting search problems. They are mainly based upon normalization of bitvector functions, represented by many-sorted terms, as well as upon symmetry reduction by removal of symmetrically valued variables. It is divided into three parts. Part one describes basic techniques, such as reducing and solving the Boolean equivalence problem, BMC and basic symmetry reduction techniques on the Boolean level. Part two elevates these techniques to the register transfer level, using bitvector functions and bitvector terms as basic structure, and extends them further. Evidence for its practical applicability is given. The last part contains additional information, namely long proofs, a formal description of the extensions to the basic bitvector algebra, the rewrite rules used, and a collection of basic algorithms.

(Note that there are other possible ways, than strictly sequential, of reading the chapters of this document, and these can be derived from the dependencies depicted in Fig. 1.4.)

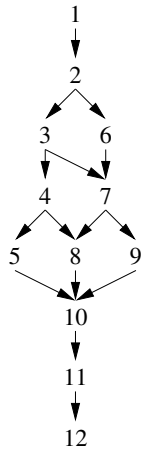


Figure 1.4: Dependency Graph of Chapters

1.7.1 Part One – Bit-Level Methods

Chapter 2 describes the algebraic treatment of Boolean functions and their representations by Boolean terms. It provides the justification for solving subsequent verification tasks at appropriate algebraic levels. The results are extended to bitvector functions in Chapter 6.

Being able to solve the equivalence problem for large Boolean functions is a basic requirement for the application of BMC. Some basic decision procedures, such as canonically representing Boolean functions and backtrack search, are described in Chapter 2. The fundamentals of term rewriting are reviewed and are subsequently applied for preprocessing Boolean equivalence problems. They are extended to bitvector functions in Chapter 7.

A variation of BMC, bounded interval model checking (BIMC), is described in Chapter 4. As for BMC, the verification problem is reduced to the Boolean equivalence problem, for which preprocessing techniques and decision procedures have already been provided. It is extended to the register transfer level in Chapter 8 enabling more efficient problem reduction.

The basic approach for exploiting symmetrical valued variables in Boolean functions is explained in Chapter 5. It is shown that it is possible to use semi-decision procedures for the Boolean permutation equivalence problem for finding symmetrical values. They can be combined, yielding more powerful techniques. Extending the basic concept of symmetrical values to symmetrical value vectors, prepares the application of this approach on the register transfer level in Chapter 10.

1.7.2 Part Two – Register Transfer Level Methods

Chapter 6 extends the concepts of algebras introduced in Chapter 2 to many-sorted algebras, providing the framework for dealing with bitvector functions. A complete set of bitvector axioms is developed, allowing the construction of bitvector algebras. Their syntax is extended by defined operations in order to conveniently deal with common HDL-operations. Term graphs, i.e. suitably labeled directed acyclic graphs, allow subsequently the efficient representation and manipulation of bitvector terms.

Extending the concept of rewriting in Chapter 7 preprocessing techniques for the Boolean equivalence problem are elevated to bitvector terms. Conversely, the equivalence problem for bitvector functions can be reduced to the Boolean equivalence problem.

This allows these, and subsequently provided (Chapter 8) reduction techniques, to be applied for preprocessing BMC-problems on the register transfer level, before solving the latter on the Boolean level using standard methods.

In order to later extend the symmetry reduction approach of Chapter 5, the permutation equivalence problem for bitvector functions is investigated in Chapter 9. Here several criteria implying permutation equivalence are developed, along with decision and semi-decision procedures, including rewriting and graph morphism based algorithms. The bitvector term-rewriting techniques, developed in Chapter 7, are combined with isomorphism procedures for specially manipulated term graphs which are based on existing algorithms, solving the automorphism problem for colored undirected graphs.

The concept of approximate symmetry relations, introduced in Chapter 10, allows the efficient combination of these techniques in the context of symmetry reduction of RTL-BMC-problems. It features stepwise refinement of an approximate symmetry relation for a set of bitvector variables, giving rise to a practical reduction algorithm operating on the register transfer level.

Aspects of implementing symmetry reduction for BMC on the register transfer level within a model checking tool, are treated in Chapter 11. The practical relevance of the overall approach is discussed using relevant industrial designs and properties, which where previously intractable for BMC.

Conclusions from the presented results are drawn in chapter 12, and possible extensions are outlined.

1.7.3 Contributions

The main contributions of this work are in providing a framework for dealing with bitvector functions algebraically, a concise description of BMC on the register transfer level, new reduction techniques and new approaches for finding and exploiting symmetrical values. These methods dramatically reduce the computational effort needed for functional verification on the block-level and, in particular, for the important problem class of regular designs. It is fully automatic and was integrated into the RTL-BMC-tool *RtProp*, which was developed along with this thesis. *RtProp* provides a platform for exploration of future register transfer level BMC-techniques.

Part I

Bit-Level Methods

Chapter 2

Algebraic Framework for Boolean Functions

In this chapter the algebraic framework for dealing with Boolean functions and their representations as Boolean terms is reviewed. It provides the justification for solving certain subtasks of hardware verification (mainly comparing Boolean functions) at appropriate (i.e. mostly more abstract) algebraic levels.

First the basic concept of algebras is reviewed (see 2.1). Elements of algebras of the same class can be connected by homomorphisms. Algebraic properties of classes of algebras can be described by identities which define congruences on their elements (see 2.2). Free algebras are the most general objects in their class and provide an easy way of constructing homomorphisms between increasingly richer algebras (see 2.3). These concepts are used in Chapter 3 for comparing Boolean functions and for providing preprocessing techniques. They are extended to bitvector functions in Chapter 6.

Subsequent notation, definitions, results, and examples are provided to the extend and at the level of detail as required for the purpose of this thesis.[1]

2.1 Algebras

Algebras are the basic concept for studying the relations between a set of operations over the same carrier. For a non-empty set A and $n \in \mathbb{N}_0$ we define A^n as the set of n-tuples of elements from A with $A^0 := \{\emptyset\}$. A function $f : A^n \to A$ is an *n-ary operation on A*, where n is the *arity* of f. A 0-ary operation is a *constant*.

Definition 2.1 (Algebra)
Let A be a set, the *carrier*, and let $F = \{f_1, \ldots, f_i, \ldots, f_m\}$ be m n_i-ary operations on A. Then we call $\mathcal{A} = (A, F)$ an *algebra*. The vector $(n_1, \ldots, n_i, \ldots, n_m)$ is termed the *signature* of $\mathcal{A}$.

Boolean algebras play an important role since they provide the foundation for algebraic description and verification of logic circuits.[2] Some examples for Boolean algebras

[1] For a good introduction into abstract algebra in the context of term rewriting see for example [BN98].

[2] We assume that the reader is familiar with the concept of Boolean algebras. The exact definition of their algebraic properties, defining this class, are given later by a set of identities in Example 2.7.

are given below.

Examples 2.2
The following examples for algebras all share the same signature $(2, 2, 1, 0, 0)$. In slight abuse of notation we use the same names for the operations on the respective carriers to indicate their connection.

1. Let $F = \{\wedge, \vee, \neg, 0, 1\}$ be operations on the carrier $\mathbb{B} := \{0, 1\}$, where 0, 1 are constants, $\neg$ is a unary operation, and $\wedge$, $\vee$ are binary operations, such that $a \wedge b := \min\{a, b\}$, and $a \vee b := \max\{a, b\}$, and $\neg a := 1 - a$. Then the two element algebra $\mathbf{B} := (\{0, 1\}, F)$ is the Boolean algebra with the smallest carrier.

2. The set of *bitvectors* of fixed length n, $\mathbb{B}^n$, is turned into a Boolean algebra $\mathbf{B}^n := (\mathbb{B}^n, \{\wedge, \vee, \neg, (0, \ldots, 0), (1, \ldots, 1)\})$ by component wise extension of Boolean operations on $\mathbb{B}$ to n-tuples, e.g. $(a_{n-1}, \ldots, a_0) \vee (b_{n-1}, \ldots, b_0) := (a_{n-1} \vee b_{n-1}, \ldots, a_0 \vee b_0)$. Note that the all-zero and all-one vectors $(0, \ldots, 0)$, and $(1, \ldots, 1)$ serve as constants.

3. Algebra of Boolean functions: For $n \in \mathbb{N}$ let $\mathbb{B}^{\mathbb{B}^n}$ be the set of all Boolean functions in n variables. Let $F = \{\wedge, \vee, \neg, 0, 1\}$ be Boolean operations on $\mathbb{B}^{\mathbb{B}^n}$ which are defined pointwise by means of the Boolean operations on $\mathbb{B}$ above, e.g. for all $f, g \in \mathbb{B}^{\mathbb{B}^n}$, and for all x in $\mathbb{B}^n$, $(f \vee g)(x) := f(x) \vee g(x)$. Then $\mathbf{B}^{\mathbf{B}^n} := (\mathbb{B}^{\mathbb{B}^n}, F)$ is the algebra of Boolean functions, which is again a Boolean algebra. The elements of $\mathbf{B}^{\mathbf{B}^n}$ can be represented by truth tables, sums of products, products of sums, ordered binary decision diagrams (see 3.1.3), etc. Note that for $n = 0$ this algebra collapses to $\mathbf{B}$ above.

4. Algebra of Boolean terms: Given a set $X = \{x_1, \ldots, x_n\}$ of *variables*. Let $F = \{\wedge, \vee, \neg, 0, 1\}$ be a set of operations. The set of Boolean terms $T_{\mathbb{B}}(X)$ over X is recursively defined such that each variable is a term, $0, 1$ are terms, and if t_1, t_2 are terms so are $t_1 \wedge t_2$, $t_1 \vee t_2$, and $\neg t_1$. The operations on $T_{\mathbb{B}}(X)$ are defined such that the application of an operation to a term yields the respective term, for example $\wedge(t_1, t_1) := t_1 \wedge t_2$. The algebra $\mathcal{T}_{\mathbb{B}}(X) := (T_{\mathbb{B}}(X), F)$ is then the *algebra of Boolean terms* on X. (If two terms s and t are identical this is denoted as $s \equiv t$ which is formally defined in 3.2.1.)

Algebras sharing certain aspects may be collected in a class. The algebras in the examples above are all members of the class of algebras having the same signature as the Boolean algebra $\mathbf{B}$.

In general all algebras having the same signature can be constructed as follows. Given a set of *function symbols* F a *signature specification* $\Sigma : F \to \mathbb{N}_0$ is a function associating each function symbol with the signature of a compatible operation of the same name. We call an algebra $\mathcal{A} = (A, F)$ a Σ-*algebra*, if it employs the set F of function symbols as operations on its carrier A and these operations have the signatures specified by Σ. The operation associated with a function symbol f is then the *interpretation* of this function symbol in $\mathcal{A}$, formally denoted as $[\![f]\!]^{\mathcal{A}}$. This notation allows to separate the realms of syntax (function symbols) from semantics (operations). However, it will be clear from the context in most cases whether we reason syntactically or semantically and therefore usually write f instead of $[\![f]\!]^{\mathcal{A}}$.

Example 2.3
Let $F = \{\wedge, \vee, \neg, 0, 1\}$ be the set of Boolean function symbols, and let $\Sigma_{\mathbf{B}} : F \to \mathbb{N}_0$ be a signature specification with $\Sigma_{\mathbf{B}} = (\wedge \mapsto 2, \vee \mapsto 2, \neg \mapsto 1, 0 \mapsto 0, 1 \mapsto 0)$. Then every $\Sigma_{\mathbf{B}}$-algebra has the Boolean signature $(2, 2, 1, 0, 0)$. Thus each algebra in Example 2.2 is a $\Sigma_{\mathbf{B}}$-algebra, i.e. an algebra with this Boolean signature.

Given a signature specification Σ, and set of variables X, the set of Σ-*terms over* X is denoted as $T_\Sigma(X)$. They are build recursively from variables, function symbols and other terms as shown for Boolean terms in Example 2.2. The *algebra of* Σ-*terms*, $\mathcal{T}_\Sigma(X)$, is constructed accordingly. (See Example 2.2.).

In a sense to be made precise later (see 2.3), the algebra of Boolean terms described in Example 2.2 above is the most general object in this class. Another example is the class of all (finite) Boolean algebras. In this class the algebras with carrier $\mathbb{B}^{\mathbb{B}^n}$ are the most general objects.

2.2 Homomorphisms Between Algebras

Now it is shown how the elements of different algebras with the same signature interact with each other under homomorphisms.

Definition 2.4 (Homomorphism)
Given two algebras $\mathcal{A} = (A, F^{\mathcal{A}})$ and $\mathcal{B} = (B, F^{\mathcal{B}})$ with the same signature, a function $\phi : A \to B$ is called a *homomorphism* from $\mathcal{A}$ into $\mathcal{B}$ iff for all n-ary operations $f^{\mathcal{A}}$ on A, there is an n-ary operation $f^{\mathcal{B}}$ on B, such that for all arguments $a_1, \ldots, a_n \in A$, $\phi(f^{\mathcal{A}}(a_1, \ldots, a_n)) = f^{\mathcal{B}}(\phi(a_1), \ldots, \phi(a_n))$.

Example 2.5
Let X be a set of variables. Let $\varphi : X \to \mathbb{B}$ be a mapping, called *variable assignment*. Then φ can be extended to a homomorphism φ from $\mathcal{T}_{\mathbb{B}}(X)$ into $\mathbf{B}$, such that φ maps variables to their assignments, constant symbols 0 and 1 to the respective constants, i.e. $\varphi(0) = 0$, and $\varphi(1) = 1$, and all other terms to the respective operation application in $\mathbf{B}$, e.g. for $t \equiv t_1 \wedge t_2$, $\varphi(t) := \varphi(t_1) \wedge \varphi(t_2)$.[3] By this construction φ is a mapping sending any Boolean term to one of the constants $\{0, 1\}$. The image of t under such a homomorphism, $\varphi(t)$ is called *valuation of the term* t *under the variable assignment* φ *in the algebra* $\mathbf{B}$.

Given an algebra $\mathcal{A} = (A, F)$, an equivalence relation $\approx$ on A is called *congruence* on A iff for all n-ary operations $f \in F$ on A, $a_1 \approx b_1, \ldots, a_n \approx b_n$ implies $f(a_1, \ldots, a_n) \approx f(b_1, \ldots, b_n)$. The congruence $\approx$ partitions the elements of A into congruence classes $[a]_\approx := \{b \in A : a \approx b\}$. Let $A/_\approx := \{[a]_\approx : a \in A\}$ be the set of congruence classes. Let $F_\approx$ be a set of operations on $A/_\approx$ defined by $f_\approx([a_1]_\approx, \ldots, [a_n]_\approx) := [f(a_1, \ldots, a_n)]_\approx$. Then $\mathcal{A}/_\approx = (A/_\approx, F_\approx)$ is called the *quotient algebra of* $\mathcal{A}$ *modulo* $\approx$. This quotient algebra has the same signature as $\mathcal{A}$.

Identities play an important role as axioms describing the algebraic properties of classes of algebras since congruences can be generated by identities. An *identity* is a pair (a, b) of elements of A.

[3] Formally this would read as $\varphi(t) := [\![\wedge]\!]^{\mathbf{B}}(\varphi(t1), \varphi(t2))$, where $[\![\wedge]\!]^{\mathbf{B}}$ is the Boolean and-operation of the algebra $\mathbf{B}$ associated with the function symbol $\wedge$.

Quotient algebras of algebras of terms are of special interest. We usually write $s = t$ for an identity $(s,t) \in E \subseteq T_\Sigma(X) \times T_\Sigma(X)$. Given a set of identities $E = \{(a_1, b_1), \ldots, (a_1, b_n)\}$ the relationship between algebras with the same signature is described by the following terms.

Definition 2.6 (Models and Holding Identities)
Let $\Sigma : F \to \mathbb{N}$ be a signature specification, let X be a set of variables. Let s, t be Σ-terms over X. The identity $s = t$ *holds* in the Σ-algebra $\mathcal{A}$, written $\mathcal{A} \models s = t$ iff for all homomorphism $\phi : \mathcal{T}_\Sigma(X) \to \mathcal{A}$ we have $\phi(s) = \phi(t)$. Given a set of identities E, $\mathcal{A}$ is a *model* of E, written $\mathcal{A} \models E$, iff every identity of E holds in $\mathcal{A}$. The identity $s = t$ is a *semantic consequence of* E, denoted as $E \models s = t$, iff it holds in all models of E.

The congruence $\approx_E := \{(s,t) \in T_\Sigma(X) \times T_\Sigma(X) : E \models s = t\}$ on $T_\Sigma(X)$, is the *equational theory* induced by E. It is the smallest[4] (w.r.t. set inclusion) congruence on $T_\Sigma(X)$, comprising E, and defines a quotient algebra of the algebra of terms.

Example 2.7

1. Boolean algebra: Let X be a finite set of variables, and let $E_{\mathbf{B}}$ be the set of all *Boolean identities* for the algebra of Boolean terms $\mathcal{T}_{\mathbb{B}}(X)$, as given below:

   (a) $a \wedge b = b \wedge a$, $a \vee b = b \vee a$ (commutative laws)

   (b) $a \wedge (b \wedge c) = (a \wedge b) \wedge c$, $a \vee (b \vee c) = (a \vee b) \vee c$ (associative laws)

   (c) $a \wedge (b \vee c) = (a \vee c) \wedge (a \vee c)$, $a \vee (b \wedge c) = (a \wedge c) \vee (a \wedge c)$ (distributive laws)

   (d) $a \wedge a = a$, $a \vee a = a$ (laws of idempotency)

   (e) $a \wedge 1 = a$, $a \vee 0 = a$ (identity elements)

   (f) $a \wedge 0 = 0$, $a \vee 1 = 1$ (dominant elements)

   (g) $a \wedge \neg a = 0$, $a \vee \neg a = 1$ (laws of complementarity)

   (h) $\neg(\neg a) = a$ (law of involution)

   (i) $\neg(a \wedge b) = \neg a \vee \neg b$, $\neg(a \vee b) = \neg a \wedge \neg b$ (De Morgan's laws)

   Then by construction the quotient algebra $\mathcal{T}_{\mathbf{B}}(X) := \mathcal{T}_{\mathbb{B}}(X)/_{\approx E_{\mathbf{B}}}$ is a Boolean algebra.

2. Let A be a set and let $F = \{\wedge, \vee, \neg, 0, 1\}$ be a set of operations on A, where 0, 1 are constants, $\neg$ is a unary operation on A, and $\wedge$, $\vee$ are binary operations on A. Then $\mathcal{A} = (A, F)$ is a *Boolean algebra* iff for all a, b in A the Boolean identities $E_{\mathbf{B}}$ above hold in this algebra. In fact the Boolean algebras $\mathcal{T}_{\mathbf{B}}(X)$, $\mathbf{B}$, and $\mathbf{B}^{\mathbf{B}^n}$ are all models of $E_{\mathbf{B}}$. The identity $x \vee \neg(x \wedge y) = 1$ holds in every Boolean algebra. It is a semantic consequence of $E_{\mathbf{B}}$.

3. Again let X be a finite set of variables, and for the algebra of Boolean terms $\mathcal{T}_{\mathbb{B}}(X)$ let $E = \{x \vee y = y \vee x,\ x \vee (y \vee z) = (x \vee y) \vee z\}$, specifying commutativity and associativity of $\vee$. Then $\mathcal{T}_{\mathbb{B}}(X)/_{\approx_E}$ is an algebra of the same signature as $\mathcal{T}_{\mathbb{B}}(X)$, but exhibiting an 'addition', i.e. a commutative and associative operation.

[4] There always exists a smallest congruence comprising E.

Note that the set of identities above is by far not the only way of defining a Boolean algebra.[5] There are many different ways of constructing the elements of a Boolean algebra by various operations. Switching between the resulting algebras on the same carrier depends on the question at hand. To improve readability the following abbreviations are defined (for $a, b, c \in A$):

$$\begin{aligned} a \oplus b &:= (a \wedge \neg b) \vee (\neg a \wedge b) \\ a = b &:= \neg(a \oplus b) \\ a \Rightarrow b &:= \neg a \vee b \\ \mathrm{ite}(a, b, c) &:= (a \wedge b) \vee (\neg a \wedge c) \end{aligned}$$

2.3 Free Algebras

The following concept makes the idea of most general objects in a class of algebras precise and also provides an easy way to build homomorphisms.

Definition 2.8 (Free Algebra)
Let C be a class of algebras with the same signature. Let $\mathcal{A} = (A, F^{\mathcal{A}})$, $\mathcal{B} = (B, F^{\mathcal{B}})$ be algebras of C. Then $\mathcal{B}$ is called *subalgebra of* $\mathcal{A}$ iff $B \subseteq A$ and for all $b_1, \ldots, b_n \in B$ we have $f^{\mathcal{A}}(b_1, \ldots, b_n) = f^{\mathcal{B}}(b_1, \ldots, b_n) \in B$. If $X \subseteq A$ then the *subalgebra of* $\mathcal{A}$ *generated by* X is the smallest subalgebra $\mathcal{B} = (B, F^{\mathcal{B}})$ of $\mathcal{A}$ such that $X \subseteq B$. If $B = A$ we say $\mathcal{A}$ is *generated by* X. The algebra $\mathcal{A}$ in C is called *free in* C *with generating set* X, iff $\mathcal{A}$ is generated by X, and for every algebra $\mathcal{B} = (B, F^{\mathcal{B}})$ in C above, every mapping $\phi : X \to B$ can be extended to a homomorphism form $\mathcal{A}$ into $\mathcal{B}$.

This extension is unique.

Examples 2.9 (Free Algebras)

1. Let X be a set of variables, let F be a set of function symbols, and let $\Sigma : F \to \mathbb{N}_0$ be a signature specification. Then the algebra of Σ-terms $\mathcal{T}_\Sigma(X)$ is free with generating set X in the class of all Σ-algebras. In particular $\mathcal{T}_{\mathbb{B}}(X)$ is free in the class of algebras with the signature specified by $\Sigma_{\mathbf{B}}$.

2. Let $\approx_E$ be the semantic congruence relation on $T_{\mathbb{B}}(X)$, derived from a set of identities $E \subseteq T_{\mathbb{B}}(X) \times T_{\mathbb{B}}(X)$, then $\mathcal{T}_{\mathbb{B}}(X)/_{\approx_E}$ is free with generating set $X/_{\approx_E} := \{[x]_{\approx_E} : x \in X\}$ in the class of algebras satisfying E. (In particular $\mathcal{T}_{\mathbf{B}}(X) = \mathcal{T}_{\mathbb{B}}(X)/_{\approx_{E_{\mathbf{B}}}}$, and thus $\mathbf{B}^{\mathbf{B}^n}$, are free in the class of all (finite) Boolean algebras.) Note that if E is not trivial, i.e. if it does not contain identities of the form $x = y$ for distinct variables x and y, then $|X/_{\approx_E}| = |X|$.

3. Let $\mathcal{A}$ be an algebra, let $\approx$ be a congruence, and let $\mathcal{A}/_{\approx}$ be the associated quotient algebra. By construction the mapping $\phi_{\approx} : a \to [a]_{\approx}$ is a (surjective) homomorphism from $\mathcal{A}$ into $\mathcal{A}/_{\approx}$. In particular, let $E_{\mathbf{B}}$ be the set of Boolean identities (Example

[5] It is even possible to define a Boolean algebra by a single identity [MVF+02].

$$\begin{tikzcd}
 & \mathcal{T}_{\mathbb{B}}(X) \arrow[dl, "\phi_{\approx_{E_{\mathbf{B}}}}"'] \arrow[dr, "\phi"] & \\
\mathcal{T}_{\mathbf{B}}(X) \arrow[rr, shift left, "\phi_{\mathbf{B}}"] & & \mathbf{B}^{\mathbf{B}^n} \arrow[ll, shift left, "\phi_{\mathbf{B}}^{-1}"]
\end{tikzcd}$$

Figure 2.1: Isomorphic Boolean Algebras

2.2), then $\phi_{\approx_{E_{\mathbf{B}}}} : \mathcal{T}_{\mathbb{B}}(X) \to \mathcal{T}_{\mathbf{B}}(X)$, with $\phi_{\approx_{E_{\mathbf{B}}}} : t \to [t]_{\approx_{E_{\mathbf{B}}}}$ is such a homomorphism (mapping Boolean terms to their congruence classes modulo the set of Boolean axioms).

4. For a finite set of variables X with $|X| = n$, $\mathcal{T}_{\mathbf{B}}(X)$, the algebra of Boolean terms $\mathcal{T}_{\mathbb{B}}(X)$ modulo $E_{\mathbf{B}}$, and $\mathbf{B}^{\mathbf{B}^n}$, the algebra of Boolean functions in n variables, are isomorphic.

   Let $\pi_i : \mathbb{B}^n \to \mathbb{B}$ be the i'th projection of $\mathbb{B}^n$ onto $\mathbb{B}$, i.e. $\pi_i(x_1, \ldots, x_n) = x_i$. The mapping $\phi : X \to \mathbb{B}^{\mathbb{B}^n}$ with $\phi(x_i) \to \pi_i$, extends to a surjective homomorphism (also denoted by ϕ) from $\mathcal{T}_{\mathbb{B}}(X)$ onto $\mathbf{B}^{\mathbf{B}^n}$, sending any Boolean term t to the Boolean function built inductively according to the term structure of t (E.g. for $t \equiv t_1 \wedge t_2$, $\phi(t) = \phi(t_1) \wedge \phi(t_2)$). Thus, every Boolean term denotes a unique Boolean function.

   Let $\phi_{\mathbf{B}} : [t]_{\approx_{E_{\mathbf{B}}}} \to \phi(t)$. Obviously all terms in $[t]_{\approx_{E_{\mathbf{B}}}}$ are mapped to $\phi(t)$. For t_1 and t_2 with $[t_1]_{\approx_{E_{\mathbf{B}}}} \neq [t_2]_{\approx_{E_{\mathbf{B}}}}$ we have $\phi(t_1) \neq \phi(t_2)$. Conversely, every Boolean function can be written as product of minterms. Hence, $\phi_{\mathbf{B}}$ is a bijection, and by construction ϕ and $\phi_{\mathbf{B}}$ are homomorphisms. Therefore $\mathcal{T}_{\mathbf{B}}(X)$ and $\mathbf{B}^{\mathbf{B}^n}$ are isomorphic. (Using $\phi_{\approx_{E_{\mathbf{B}}}}$ from above, the situation is depicted in Fig. 2.1.)

5. Let $X = \{X_1, X_2, X_3\}$ and consider the sequence of sets of identities

$$E_{\emptyset} = \emptyset \subseteq E_{\wedge} \subseteq E_{\wedge,\vee} \subseteq \cdots \subseteq E_{\mathbf{B}}$$

   on $T_{\mathbb{B}}(X)$, where:

$$\begin{aligned}
E_{\wedge} &= \{x \wedge y = y \wedge x,\ x \wedge (y \wedge z) = (x \wedge y) \wedge z, \\
&\quad\ x \wedge \neg x = 0,\ x \wedge 1 = x,\ x \wedge 0 = 0\} \\
E_{\vee} &= \{x \vee y = y \vee x,\ x \wedge (y \vee z) = (x \vee y) \vee z, \\
&\quad\ x \vee \neg x = 1,\ x \vee 0 = x,\ x \vee 1 = 1\} \\
E_{\wedge,\vee} &= E_{\wedge} \cup E_{\vee}
\end{aligned}$$

   Then $\approx_{E_{\emptyset}}, \approx_{E_{\wedge}}, \approx_{E_{\wedge,\vee}}, \ldots, \approx_{E_{\mathbf{B}}}$ is a sequence of congruences in $T_{\mathbb{B}}(X)$ defining a sequence of free algebras of increasingly richer algebraic structure. (See Table 2.1.) The sequence of congruences $\approx_{E_{\emptyset}}, \approx_{E_{\vee}}, \approx_{E_{\wedge,\vee}}, \cdots, \approx_{E_{\mathbf{B}}}$ defines another sequence of algebras. These algebras are all generated by X and are each free in their respective classes.

Table 2.1: Free $\Sigma_{\mathbb{B}}$-Algebras Generated by X with Identities and Congruences

Algebra	Identities	Congruence
$\mathcal{T}_{\mathbb{B}}(X)$	$E = E_{\emptyset}$	$\emptyset$
$\mathcal{T}_{\mathbb{B}}(X)/\approx_{E_\wedge}$	$E_\wedge$	$\approx_{E_\wedge}$
$\mathcal{T}_{\mathbb{B}}(X)/\approx_{E_\vee}$	$E_\vee$	$\approx_{E_\vee}$
$\mathcal{T}_{\mathbb{B}}(X)/\approx_{E_{\wedge,\vee}}$	$E_{\wedge,\vee}$	$\approx_{E_{\wedge,\vee}}$
$\mathcal{T}_{\mathbf{B}}(X) = \mathcal{T}_{\mathbb{B}}(X)/\approx_{E_{\mathbb{B}}}$	$E_{\mathbf{B}}$	$\approx_{E_{\mathbf{B}}}$

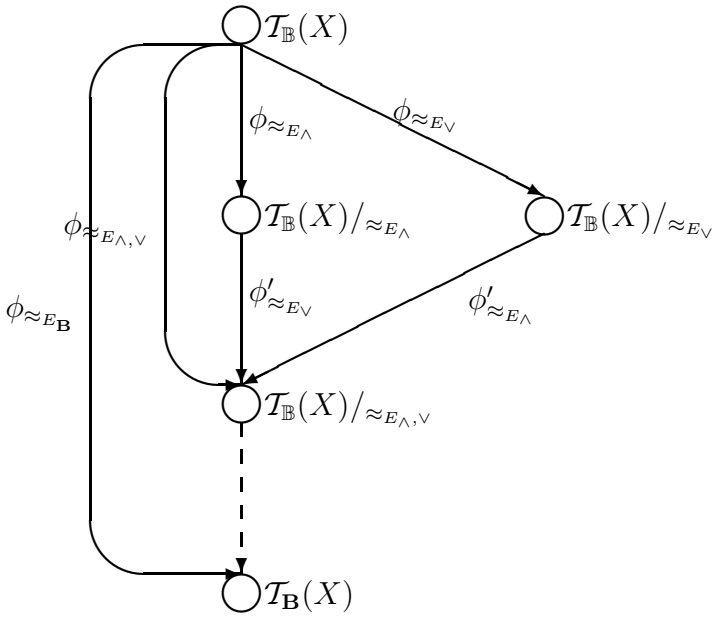


Figure 2.2: Sequences of Increasingly Richer Algebras with Homomorphisms

The generation process of this sequence can be described as subsequent application of homomorphisms as depicted in Fig. 2.2. Note that $\phi'_{\approx_{E_\vee}}$ and $\phi'_{\approx_{E_\wedge}}$ are defined on the congruence classes modulo $\approx_{E_\wedge}$ and $\approx_{E_\vee}$, respectively, and are extended from $\phi_{\approx_{E_\vee}}$ and $\phi_{\approx_{E_\wedge}}$. This diagram provides the justification for working at the appropriate algebraic level to solve certain tasks and interpret the results in related algebras. Also, the fact that above algebras are free allows to easily construct homomorphisms, isomorphisms and automorphisms from mappings defined on the respective generating sets. For example let $\mathcal{A}$ be a model of $E_{\mathbf{B}}$ and let s, t be Boolean terms, then $E_\wedge \models s = t$ implies $\mathcal{A} \models s = t$.

The results above allow to interchangeably reason about Boolean functions either in $\mathbf{B}^{\mathbf{B}^n}$, or in $\mathcal{T}_{\mathbf{B}}(X)$. This is important, since it will allow for reasoning syntactically about Boolean functions as described in the next chapter.

Chapter 3

Comparing Boolean Functions

Being able to compare large Boolean functions is one of the most important prerequisites for verification of sequential circuits. In this chapter semantic and syntactic decision and semi-decision procedures for solving Boolean equivalence and satisfiability problems are discussed.

First in 3.1 some standard decision procedures for the Boolean equivalence problem are reviewed, namely backtrack search and canonical representations of Boolean functions. Since all these procedures require exponential time or space in the worst case, syntactic preprocessing based on rewriting is proposed as a method for problem reduction (see 3.2). These methods are extended to bitvector functions in Chapter 7. The combination of this preprocessing with decision procedures is discussed in 3.3.

3.1 Decision Procedures for Boolean Equivalence

In this section the Boolean equivalence problem is stated (see 3.1.1) and two main ideas for solving it are reviewed briefly, namely backtrack search using the Davis-Logeman-Loveland (DLL) procedure (see 3.1.2) and canonically representing Boolean functions by reduced ordered binary decision diagrams (see 3.1.3). These methods are used later for solving the equivalence problem of bitvector functions (see 7.1).

3.1.1 The Boolean Equivalence Problem

Recall that $\mathcal{T}_{\mathbf{B}}(X)$ (with $|X| = n$) and $\mathbf{B}^{\mathbf{B}^n}$ are isomorphic. Given two Boolean functions $h, g \in \mathbb{B}^{\mathbb{B}^n}$, and terms $s, t \in \mathcal{T}_{\mathbb{B}}(X)$ representing them, the question whether h and g are the same, i.e. whether $\mathcal{T}_{\mathbf{B}}(X) \models s = t$, is called *Boolean equivalence problem*, which is a *decision problem*. It is commonly to be solved in verification of combinational and sequential circuits, including symbolic and bounded model checking, since many tasks in verification reduce to the Boolean equivalence problem.

Example 3.1
Let $h(x_1, \ldots, x_3) = (x_1 \wedge x_3) \vee (\neg x_1 \wedge \neg x_2) \vee (\neg x_1 \wedge x_2 \wedge x_3)$ and let $g = 1$, then the question whether $(x_1 \wedge x_3) \vee (\neg x_1 \wedge \neg x_2) \vee (\neg x_1 \wedge x_2 \wedge x_3) = 1$ holds is a Boolean equivalence problem.

The Boolean equivalence problem can be normalized to the question whether a term for a Boolean function represents 1, i.e. to $\mathcal{T}_{\mathbf{B}}(X) \models \neg(s \oplus t) = 1$, since $h = g$ iff $h \oplus g = 0$ iff $1 \oplus h \oplus g = 1$.

Note that if a truth table for f is given instead, the question whether $f = 1$ is trivial. (Just walk through this table and check whether the value of f is always 1). However a truth table for f is an exponential representation in the number of the arguments of f. (For n arguments it has 2^n entries, one for each combination of arguments.) Thus exponentially many comparisons have to be made to decide whether $f = 1$. Therefore Boolean function are usually represented more compactly by terms (as introduced in Chapter 2).

The Boolean equivalence problem is decidable, as the terms s and t are equivalent in $\mathcal{T}_{\mathbf{B}}(X)$ iff all their valuations in $\mathbf{B}$ are equal, i.e.

$$\mathcal{T}_{\mathbf{B}}(X) \models s = t \Leftrightarrow \forall \varphi : X \to \mathbb{B} : \varphi(s) = \varphi(t)$$

where for each φ, the valuation of a term t under φ in $\mathbb{B}$, $\varphi : \mathcal{T}(X) \to \mathbb{B}$ is a homomorphism (Example 2.5) defined inductively over the term structure. (The normal form $\mathcal{T}_{\mathbf{B}}(X) \models \neg(s \oplus t) = 1 \Leftrightarrow \forall \varphi : X \to \mathbb{B} : \varphi(\neg(s \oplus t)) = 1$ can be constructed easily applying the not and exclusive-or operations onto the terms s and t.) This gives rise to a simple decision procedure, *exhaustive simulation*, where $\varphi(\neg(s \oplus t)$ is computed and compared with 1 for all φ. Obviously this is done in 2^n steps and is equivalent to walking through the truth tables for h and g, however without storing them somewhere explicitly.

Practical decision procedures rely on a (more compact) representation of Boolean functions than truth tables, or try to avoid enumerating all assignments explicitely. Two important examples are reduced ordered binary decision diagrams and backtrack search procedures for the Boolean satisfiability problem.[1] Both are described in more detail below. The Boolean equivalence problem is inherently complex (Co-NP-complete[2].). While ROBDDs shift this complexity into the representation, DLL-procedures shift it into the complexity of the search. In general neither approach can avoid the inherent complexity.

3.1.2 Backtrack Search Procedures

Backtrack search for Boolean satisfiability is one of the earliest ideas put to practical use for solving Boolean equivalence problems. They are commonly referred to as *Davis-Logeman-Loveland (DLL) procedures*[3]. For an overview on the methods described below as well as related approaches the reader is referred to [KMSM02]. The DLL-procedure is based on the following observation: A function $f \in \mathbb{B}^{\mathbb{B}^n}$ is constantly 1 iff for all $x_1, \ldots, x_n \in \mathbb{B}$ it holds that $f(x_1, \ldots, x_n) = 1$, or conversely iff there are no $x_1, \ldots, x_n \in \mathbb{B}$ such that $f(x_1, \ldots, x_n) = 0$. This is the case iff there are no such $x_1, \ldots, x_n$, satisfying

[1] It is well knows that the Boolean equivalence problem can be transformed into to the Boolean satisfiability problem, and vice versa.

[2] For an overview on the complexity classes of some classical equivalence relations on Boolean functions see e.g. [BRS98]

[3] Such a procedure was first described in [DLL62]. It is to be distinguished from the Davis-Putnam (DP) procedure described in [DP60] which is based on resolution and represents a different research direction. However, modern DLL implementations such as [MMZ+01] can also be seen as implementations of the DP-procedure, since they implicitly construct a resolution proof [ZM03].

$\neg(f(x_1, \ldots, x_n)) = 1$, or short:

$$f = 1 \Leftrightarrow \neg\exists x_1, \ldots, x_n \in \mathbb{B} : \neg f(x_1, \ldots, x_n) = 1$$

This gives rise to backtracking search procedures, implicitly exploring the search space of the $\mathbb{B}^n$ possible values for $x_1, \ldots, x_n$, and testing whether they are satisfying $\neg f(x_1, \ldots, x_n) = 1$. If no such solution exists, $f = 1$ holds.

The idea is to represent $f \in \mathbb{B}^{\mathbb{B}^n}$ as Boolean term (Boolean formula) t over n variables $x_1, \ldots, x_n$, in conjunctive normal form (CNF), i.e. as a product of sums. A Boolean term t is in CNF if it is a conjuction of clauses $c_1, \ldots, c_k$, a clause c_i is a disjunction of literals $l_{i,1}, \ldots, l_{i,p}$, and a literal is either a variable $x_{i,j}$ or its negation $\neg x_{i,j}$. Note that a Boolean term in CNF denotes a unique Boolean function, however, a Boolean function can be represented by many different Boolean terms in CNF.

Example 3.2 (CNF)
Consider the function $f(x_1, \ldots, x_3) = (x_1 \wedge x_3) \vee (\neg x_1 \wedge \neg x_2) \vee (\neg x_1 \wedge x_2 \wedge x_3)$. To check whether $f = 1$ holds we check $\neg f = 1$ for satisfiability. Since f can be represented by the Boolean term $(x_1 \wedge x_3) \vee (\neg x_1 \wedge \neg x_2) \vee (\neg x_1 \wedge x_2 \wedge x_3)$ which is in disjunctive normal form (DNF) the CNF for $\neg f$ can be computed easily by applying DeMorgan's law twice yielding $(\neg x_1 \vee \neg x_3) \wedge (x_1 \vee x_2) \wedge (x_1 \vee \neg x_2 \vee \neg x_3)$. Obviously there are three clauses $\neg x_1 \vee \neg x_3$, $x_1 \vee x_2$ and $x_1 \vee \neg x_2 \vee \neg x_3$.

In general, Boolean terms can be translated into CNF by subsequent application of DeMorgan's laws, and laws of involution and distributivity, however, this leads to an exponential blow up in the size of the term. Instead, the following approach is taken: Let t be a Boolean term, an equivalent term t' in CNF is constructed from t. For each non-terminal subterm s, a fresh Boolean variable x_s is introduced, (otherwise $x_s \equiv s$) and clauses are conjunctively added to t':

1. If $s \equiv \neg u$, we have $x_s = \neg x_u$, i.e. $x_s \Rightarrow \neg x_u$ and $\neg x_u \Rightarrow x_s$, i.e. add the clauses $x_s \vee x_u$ and $\neg x_s \vee \neg x_u$ to t'.

2. If $s \equiv u \wedge v$, we have $x_s = x_u \wedge x_v$, i.e. $x_s \Rightarrow x_u$, $x_s \Rightarrow x_v$, and $x_u \wedge x_v \Rightarrow x_u$, thus adding $\neg x_s \vee x_u$, $\neg x_s \vee x_v$ and $\neg x_u \vee \neg x_v \vee x_s$ to t'.

3. If $s \equiv u \vee v$, we have $x_s = x_u \vee x_s$, i.e. $x_s \Rightarrow x_u \vee x_v$, $x_u \Rightarrow x_s$, and $x_v \Rightarrow x_s$, thus adding $\neg x_s \vee x_u \vee x_v$, $\neg x_u \vee x_s$, and $\neg x_v \vee x_s$.

This approach leads only to a blow up by factor 3 in the number of subterms of t'.[4]

Now backtrack search is applied to a given CNF formula to find a satisfying assignment to its variables or prove its unsatisfiability. At the beginning all variables are unassigned. During the search process the values 0 or 1 are successively assigned to the variables $x_1, \ldots, x_n$. Thus, variables can be either unassigned, 0, or 1. Given a (partial) assignment to its variables a clause can be checked for satisfiability as follows. It is satisfied if there is a literal in this clauses which is equal to 1. If all clauses are satisfied, a solution is found. If all literals in a clause are 0, the clause is conflicting. It is satisfiable as long it contains a free literal (a literal for an unassigned variable). If a clause contains only one free literal and all other literals are 0 the value of this literal is implied and the clause is unit.

[4]This approach was first described in [PG86].

Starting from an empty assignment (all variables are unassigned), the DLL-procedure performs the following basic steps:

1. A new value is assigned to an unassigned variable, which is called making a decision. This yields a new assignment, exploring a new part of the search space, since the algorithm keeps track on the assignments already made and always tries a new one, until all assignments are explored. (The decisions made are usually recorded on a decision stack.)

   If there is a conflicting clause under the current assignment, the decision is taken back and a new decision is made. If all assignments have been tried and no solution has been found, the procedure returns *unsatisfiable*. Otherwise the next step is performed.

2. The effect of the decision is propagated. The current assignment is extended by iteratively computing the implications of the current assignment. (This is called Boolean constraint propagation (BCP) and relies on the unit clause rule which states that a literal is implied by a clause, iff all other literals in this clause are 0.) This process stops if either a conflict occurs, i.e. a variable is implied both 0 and 1, or there are no more implications possible.

   (a) In the first case the whole remaining subspace leads to conflicts and is therefore not explored further. Instead the conflict is resolved by taking back the last decision and all assignments resulting from BCP (backtracking). Then a new decision is made. (first step)

   (b) In the latter case there are either all variables assigned, for which the procedure terminates with *satisfiable*, or a new decision is made (first step).

Note that if there is a conflict (conflicting clause or conflicting literals) the whole remaining subspace of unassigned variables is unsatisfiable, for which this subspace is not explored by the algorithm. This is the reason for not exploring all combinations of values in many practical cases.

Example 3.3 (Execution of a DLL-Procedure)
Given the three clauses $c_1 \equiv \neg x_1 \vee \neg x_3$, $c_2 \equiv x_1 \vee x_2$ and $c_3 \equiv x_1 \vee \neg x_2 \vee \neg x_3$ from Example 3.2, one possible execution trace shown in Table 3.1 illustrates the procedure. It finds a satisfying solution $x_1 = 1$, $x_2 = 1$, and $x_3 = 0$. It is easy to see that $f(1, 1, 0) = 0$ which disproves $f = 1$.

There are many variants of the DLL-procedures, which are usually much more sophisticated then the basic scheme above. They include for example variable selection heuristics [MMZ+01], conflict driven learning [MSS96], non chronological back tracking [MSS96], and random restarts. (For an overview see for example [KMSM02].) However, any such procedure has an exponential run time in the number of variables in the worst case, since then all assignments have to be tried.

3.1.3 Canonically Representing Boolean Functions

Another approach to the Boolean equivalence problem is to represent Boolean functions in a canonical way. A representation is canonical if two functions are identical iff their

Table 3.1: An Execution Trace of a DLL-Procedure

Action	Assignment	c_1	c_2	c_3
Start	$\emptyset$	$\neg x_1 \vee \neg x_3$	$x_1 \vee x_2$	$x_1 \vee \neg x_2 \vee \neg x_3$
Decision	$x_2 = 1$	$\neg x_1 \vee \neg x_3$	$x_1 \vee 1$	$x_1 \vee \neg 1 \vee \neg x_3$
BCP	$x_2 = 1$	$\neg x_1 \vee \neg x_3$	1	$x_1 \vee \neg x_3$
Decision	$x_2 = 1, x_3 = 1$	$\neg x_1 \vee \neg 1$	1	$x_1 \vee \neg 1$
BCP(Conflict)	$x_2 = 1, x_3 = 1$, $(x_1 = 0/1)$	$\neg x_1$	1	x_1
Back-Track	$x_2 = 1$	$\neg x_1 \vee \neg x_3$	1	$x_1 \vee \neg x_3$
Decision	$x_2 = 1, x_3 = 0$	$\neg x_1 \vee \neg 0$	1	$x_1 \vee \neg 0$
BCP	$x_2 = 1, x_3 = 0$	1	1	1
Decision	$x_2 = 1, x_3 = 0, x_1 = 1$	1	1	1
BCP(SAT)	$x_2 = 1, x_3 = 0, x_1 = 1$	1	1	1

representations are. This approach is most notably followed by reduced ordered decision diagrams (ROBDDs) [Bry86]. Given a fixed order on variables, every Boolean function can be canonically represented by a ROBDD. Canonicity means that for any $f : \mathbb{B}^n \to \mathbb{B}$, there is exactly one ROBDD representing it. So the decision problem for $f = 1$ is trivial, since f's ROBDD must be 1.

The ROBDD representation of a Boolean function is based on the *Shannon expansion*. For any $f \in \mathbb{B}^{\mathbb{B}^n}$, and for all $x_1, \ldots, x_n \in \mathbb{B}$, f is expanded as follows:

$$\begin{aligned} f(x_1, \ldots, x_n) &= (x_i \wedge f_{x_i}) \vee (\neg x_i \wedge f_{\bar{x}_i}) \\ f_{x_i} &:= f(x_1, \ldots, x_{i-1}, 1, x_{i+1}, \ldots, x_n) \\ f_{\bar{x}_i} &:= f(x_1, \ldots, x_{i-1}, 0, x_{i+1}, \ldots, x_n) \end{aligned}$$

where f_{x_i} and $f_{\bar{x}_i}$ are the positive and negative cofactor of f at x_i, respectively. Given a total order $<$ on the variables $x_1, \ldots, x_n$, the function f can be expanded iteratively according to this order.

Example 3.4 (Shannon Expansion)
Let $f(x_1, \ldots, x_3) = (x_1 \wedge x_3) \vee (\neg x_1 \wedge \neg x_2) \vee (\neg x_1 \wedge x_2 \wedge x_3)$, Let $x_1 < x_2 < x_3$ be the order on the variables then the BDD for f is constructed as follows:

$$\begin{aligned} f &= (x_1 \wedge x_3) \vee (\neg x_1 \wedge \neg x_2) \vee (\neg x_1 \wedge x_2 \wedge x_3) \\ f_{\bar{x}_1} &= (\neg x_2) \vee (x_2 \wedge x_3) \\ f_{x_1} &= x_3 \\ f_{\bar{x}_1 \bar{x}_2} &= 0 \\ f_{\bar{x}_1 x_2} &= x_3 \\ f_{x_1 x_2} &= x_3 \\ f_{x_1 \bar{x}_2} &= x_3 \\ \cdots &= \cdots \end{aligned}$$

This expansion can be represented by a directed tree with the first variable at the root node, the expanded cofactors for the next variable as children, and the constants 0 and 1

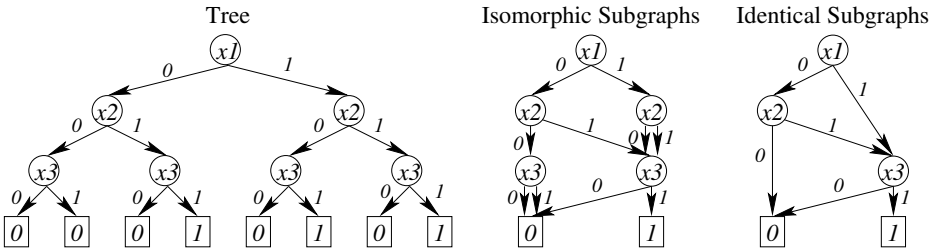


Figure 3.1: ROBDD Construction for $(x_1 \wedge x_3) \vee (\neg x_1 \wedge \neg x_2) \vee (\neg x_1 \wedge x_2 \wedge x_3)$, and Ordering $x_1 < x_2 < x_3$

at the leave nodes. If isomorphic subterms are shared and nodes with identical children are removed, one yields the ROBDD[5] for f. Note that the ROBDD representation of a Boolean function depends on the variable ordering chosen.

Example 3.5 (ROBDD Construction)
The resulting binary decision diagrams for the last example (as tree, sharing isomorphic subgraphs, and after removing identical subgraphs (ROBDD)) are shown in Fig. 3.1. As can be seen the resulting ROBDD consists of more than just one terminal node with label 1. Therefore $f = 1$ does not hold. (To find a counterexample just traverse the ROBDD from terminal 0 upwards to the root.) Note that in practice ROBDDs are usually built from bottom up, thus avoiding to construct the full tree before sharing isomorphic subgraphs. For $f = (x_1 \wedge x_3) \vee (\neg x_1 \wedge \neg x_2) \vee (\neg x_1 \wedge x_2 \wedge x_3)$ this could be achieved by first computing the ROBDDs for $f_1 = (x_1 \wedge x_3)$, $f_2 = (\neg x_1 \wedge \neg x_2)$, and $f_3 = (\neg x_1 \wedge x_2 \wedge x_3)$, and then computing ROBDDs for $f_4 = f_1 \vee f_2$ and finally $f = f_4 \vee f_3$.

ROBDDs have some very attractive properties, for example, computing the ROBDDs for $f \wedge g$, and $f \vee g$ from given ROBDDs for f, and g requires only polynomial time. The computation of $\neg f$, and the decisions $f \neq 1$ and $f = 1$ require only constant time. However, the ROBDD representation for most Boolean functions is exponential in the number of variables. Therefore, if the ROBDD for f is computed from a term representation bottom up for each subterm, the intermediate results can have exponential size, even though $f = 1$. Therefore the applicability of ROBDDs for deciding the equivalence of Boolean functions is limited in practice. In comparison to ROBDDs, SAT trades exponential space requirements of ROBDDs for exponential runtime. In more general terms the inherent problem complexity (NP-complete) is shifted from the representation into the search issue.

BDDs can be considered to be term graphs with If-Then-Else operations in each node. The procedures used to construct ROBDDs from given term graphs can be seen as application of rewrite rules and sharing. This idea is also followed in the context of Boolean expression diagrams [AH97]. They are transformed into ROBDDs by stepwise application of transformation rules while sharing isomorphic subgraphs. If this construction runs out of memory the resulting term graph is not canonical. However, if two such term graphs

[5]Note that a ROBDD for a function $f \in \mathbb{B}^{\mathbb{B}^n}$ and variables $x_1, \ldots, x_n$, sharing isomorphic subgraphs, can be interpreted as finite state machines with the root node as start state, one accepting state (the terminal node 1), and one rejecting state (the terminal 0), accepting words w of length n over the alphabet $\{0, 1\}$ iff $f(w_1, \ldots, w_n) = 1$.

are identical they represent the same function. This gives rise to syntactic semi-decision procedures for preprocessing Boolean equivalence problems. As similar approach for syntactic preprocessing and problem reduction the general concepts of term rewriting are described in the next section.

3.2 Syntactic Approach

Instead of solving the Boolean equivalence problem semantically, i.e. in $\mathbf{B}^{\mathbf{B}^n}$ or $\mathcal{T}_{\mathbf{B}}(X)$, it can sometimes be solved by deciding the semantic equivalence relation on Boolean terms $\approx_{E_{\mathbf{B}}}$ syntactically. Syntactic approaches rely on an axiomatic description of the properties of the underlying algebra by identities (see 2.2). Recall that two terms are equal in such an algebra, iff they are equivalent modulo these identities. In particular, let X be a set of variables with $|X| = n$. Since $\mathbf{B}^{\mathbf{B}^n}$ and $\mathcal{T}_{\mathbf{B}}(X)$ are isomorphic, the question whether $f = g$ can be decided in $\mathcal{T}_{\mathbf{B}}(X)$. Given an isomorphism[6] $\phi_{\mathbf{B}}$ between them, let $s \in \phi_{\mathbf{B}}^{-1}(f)$, and $t \in \phi_{\mathbf{B}}^{-1}(g)$, then the question is whether $s = t$ holds in $\mathcal{T}_{\mathbf{B}}(X)$, or whether $E_{\mathbf{B}} \models s = t$.

A syntactic decision, whether two terms are equivalent can be based on syntactic derivations using equational logic, or on reductions as described below. The latter are better suited for computation and are the basis of term rewriting as a decision procedure.

The remainder of this section is organized as follows. After clarifying the structure of terms, we describe the basic operations for manipulating them, i.e. replacement and substitution, and introduce the syntactic consequence relation on terms (see 3.2.1). Then the notion of reducing terms is formalized by introducing reduction relations on sets of terms, w.r.t. a given set of identities (see 3.2.3). The reflexive-symmetric-transitive closure of this relation coincides with the semantic equivalence relation induced by this set of identities. Therefore term rewrite systems (see 3.2.4) allow for decision procedures for this equivalence relation if they satisfy certain properties, namely termination (see 3.2.5) and confluence (see 3.2.6). Finally we show how such systems can be constructed from given sets of identities automatically and discuss possible drawbacks and extensions (see 3.2.7).

The techniques described here are used in conjunction with the procedures from the last section in 3.3. They are extended to many-sorted algebras used in the context of preprocessing for the equivalence problem of bitvector function in 7.2. Subsequent results are provided to the extent required for the purpose of this thesis. For a detailed description the reader is referred to standard text books on term rewriting, such as [Bac91, Ave95, BN98, DJ94].

For the rest of this section let X be a finite set of variables, let $\Sigma : F \to \mathbb{N}_0$ be a signature specification (see 2.1), and let $T_\Sigma(X)$ denote the set of all Σ-terms over X.

3.2.1 Manipulating Terms

Replacement and substitution are the basic operations performed when manipulating terms in syntactic decision procedures. Before describing them in detail a more detailed notion of the structure of a term is required.

[6] See Example 2.5 and Fig. 2.1, Pg. 18

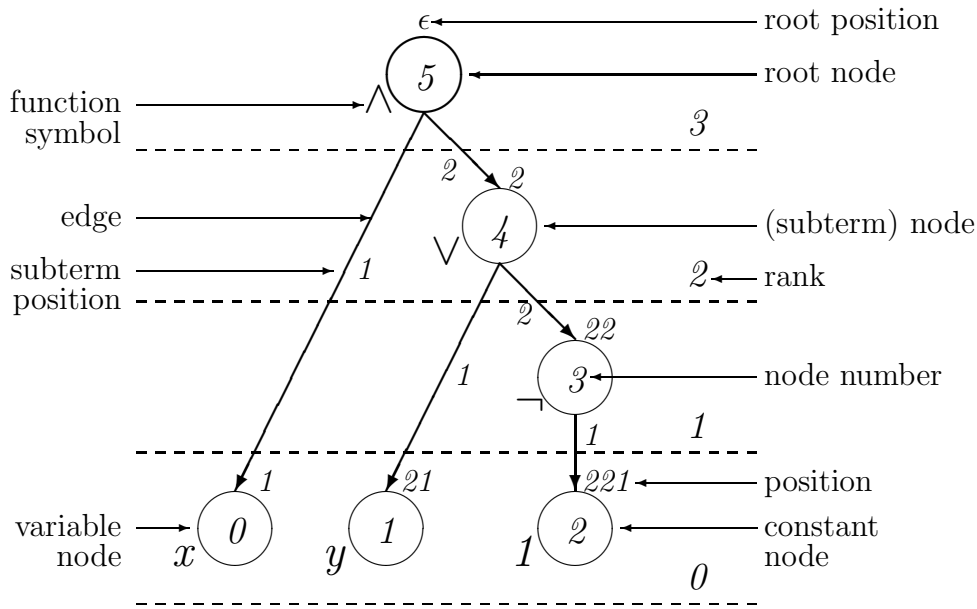


Figure 3.2: Representation $x \wedge (y \vee \neg 1)$ as Labeled Directed Tree

Let X be a set of variables, let s, t be terms. The set of *positions* of the term t is a set $Pos(t)$ of strings over the alphabet of positive integers, where ϵ denotes the empty string, called *root position*, such that:

1. If $t = x \in X$ then $Pos(x) := \{\epsilon\}$.
2. If $t = f(t_1, \ldots, t_n)$ then $Pos(t) := \{\epsilon\} \cup \bigcup_{i=1}^{n} (ip : p \in Pos(t_i))$. The *subterm of* t *at position* p is denoted as $t|_p$ with $t|_\epsilon := t$, and $t|_{ip} := t_i|_p$.

Then two terms s, t are *identical* iff they have the same positions and the (variable or function) symbols are identical at each position, which is denoted as $s \equiv t$. The *rank* $rk(t)$ of a subterm t is recursively defined, such that $rk(x) := 0$ and $rk(f(t_1, \ldots, t_n)) := max(rk(t_1), \ldots, rk(t_n)) + 1$. The set of *variables in* t is denoted as $Var(t)$. The term t is called *ground* (or a *ground term*) iff it contains no variables ($Var(t) = \emptyset$). We call the position $p \in Pos(t)$ a *variable position* if the subterm of t at p, $t|_p$ is a variable. The size $|t|$ of t is the cardinality of $Pos(t)$.

Example 3.6
The positions in $t \equiv x \wedge (y \vee \neg 1)$ are ϵ, 1, 2, 21, 22, and 221. The subterm at position 22 is $\neg 1$, its rank is 1, and it is a ground term. The positions 1 and 21 are variable positions, and the set of variables of t, $Var(t)$ is $\{x, y\}$. The size of t is 5.

Terms can be represented by labeled directed trees where nodes are labeled with function symbols and positions, arrows point to arguments of the function, and edges are labeled with subterm positions. These trees are a special case of term graphs introduced in generality later for bitvector terms (see 6.4).

Example 3.7
The term $x \wedge (y \vee \neg 1)$ is depicted as labeled directed tree in Fig. 3.2.

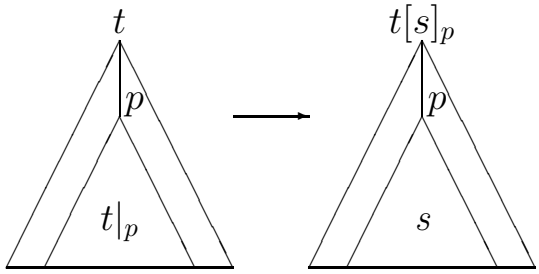


Figure 3.3: Replacing Term t at Position p with Term s

Definition 3.8 (Replacement and Substitution)
Let $s, t \in T_\Sigma(X)$ be terms. The term t with its subterm $t|_p$ *replaced* by a term s, as illustrated in Fig. 3.3, is denoted as $t[s]_p$, where

$$\begin{aligned} t[s]_\epsilon &:= s, \\ f(t_1, \ldots, t_n)[s]_{iq} &:= f(t_1, \ldots, t_i[s]_q, \ldots, t_n) \end{aligned}$$

A *substitution* is a function $\sigma : X \to T_\Sigma(X)$ mapping variables to terms. Substitutions are naturally extended to non variable terms such that $\sigma(f(t_1, \ldots, t_n)) = f(\sigma(t_1), \ldots, \sigma(t_n))$. If x is a variable, then x substituted with s in t is denoted as $t[x/s]$. If there is a substitution σ such that $\sigma(t) = s$ then s is an *instance* of t.

(Substitution is a special kind of replacement where variables are replaced with terms at all positions they occur.)

Example 3.9
$t \equiv x \wedge (y \vee \neg 1)$ replaced with 0 at position 21, is $x \wedge (0 \vee \neg 1)$. This is a substitution of y with 0, which can be denoted as $t[y/0]$.

3.2.2 Syntactic Consequences

Given a set of identities E, then the syntactic consequence relation $=_E$ can be derived from E by closing the relation E under reflexivity, symmetry, transitivity, and substitution operations on terms, as indicated by the derivation rules below:

$$\frac{(s,t) \in E}{s =_E t} \qquad \frac{}{t =_E t} \qquad \frac{s =_E t}{t =_E s}$$

$$\frac{s =_E t \quad t =_E u}{s =_E u} \qquad \frac{s =_E t}{\sigma(t) =_E \sigma(s)} \qquad \frac{s_1 =_E t_1 \ \cdots \ s_n =_E t_n}{f(s_1, \ldots, s_n) =_E f(t_1, \ldots, t_n)}$$

Given a set of identities E, we say that a term s is a *syntactic consequence* of t in E iff s can be derived from t using the identities in E and the inference rules above, which is written as $E \vdash s = t$. An alternative approach of defining this relation is based on the reduction relation, described next.

3.2.3 Reduction

Given an identity $l = r$, we call l the *left-hand side (lhs)* and r the *right-hand side* of the identity $l = r$. Replacing an instance of l occurring in a term s with an instance of

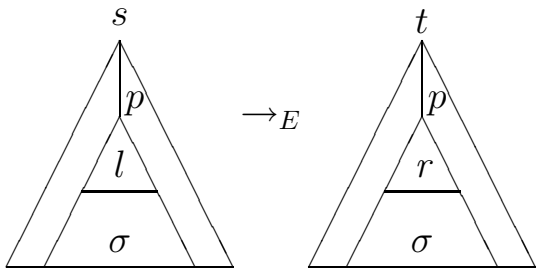


Figure 3.4: Reduction $s \rightarrow_E t$ using $l = r$ at Pos. p with Subst. σ.

r yields an equivalent term t, a reduced version of s. Reduction is the basic 'operation' performed subsequently during rewriting where one term is reduced to another that is smaller w.r.t. the reduction relation defined below.

Definition 3.10 (Reduction Relation)
Let E be a set of identities. Let $s, t \in T_\Sigma(X)$ be terms. Let $(l, r) \in E$ be an identity, let p be a position in s and let σ be a substitution, such that $s|_p = \sigma(l)$ and $t = s[\sigma(r)]_p$. Then s *reduces* to t, written $s \rightarrow_E t$. The relation $\rightarrow_E \subseteq T_\Sigma(X) \times T_\Sigma(X)$ is called *reduction relation for* E. This is illustrated in Fig. 3.4.

Example 3.11
Consider $E = \{x \vee (y \vee z) = (x \vee y) \vee z\}$ on $T_{\mathbb{B}}(X)$, which reduces $s \equiv \neg 1 \vee (1 \vee 1)$ to $t \equiv (\neg 1 \vee 1) \vee 1$, with $p = \epsilon$, $l \equiv x \vee (y \vee z)$, $r \equiv (x \vee y) \vee z$, $\sigma = \{x \mapsto \neg 1, y \mapsto 1, z \mapsto 1\}$.

Let $\rightarrow_E$ be a reduction relation. Then the inverse ($\leftarrow_E$) of $\rightarrow_E$, the symmetric ($\leftrightarrow_E$), reflexive ($\overset{=}{\rightarrow}_E$), transitive ($\overset{*}{\rightarrow}_E$), and reflexive-symmetric-transitive closure ($\overset{*}{\leftrightarrow}_E$) of $\rightarrow_E$ are reduction relations. (For formal definitions and properties of reduction relations see [BN98].) It can be shown that $E \vdash s = t$ iff $s \overset{*}{\leftrightarrow}_E$ t. This means that $E \vdash s = t$ can be proven by application of the identities in E. If $E \vdash s = t$ then this is justified by a *proof*, i.e. a finite derivation of the form $s \leftrightarrow_E s_1 \leftrightarrow_E \cdots \leftrightarrow_E s_n \leftrightarrow_E t$.

Birkhoff's theorem states that the syntactic and semantic consequence relations induced by a set of identities E coincide, which allows to reason either semantically or syntactically when deciding the Boolean equivalence problem.

Theorem 3.12 (Birkhoff)
Given a set of variables X, a set of identities $E \subseteq T_\Sigma(X) \times T_\Sigma(X)$, and two terms $s, t \in T_\Sigma(X)$. Then $s \approx_E t$ iff $s \overset{*}{\leftrightarrow}_E t$. (see e.g. [BN98])

In other words $\mathcal{T}_\Sigma(X)/_{\approx_E} \models s = t$ iff $E \models s = t$ iff $E \vdash s = t$ iff $s \overset{*}{\leftrightarrow}_E t$.

Example 3.13
Let $E_\vee$ be a subset of the Boolean identities $E_{\mathbf{B}}$ (Example 2.7) describing properties of $\vee$, i.e.

$$\begin{aligned} E_\vee \;=\; & \{x \vee y = y \vee x,\ x \vee (y \vee z) = (x \vee y) \vee z, \\ & x \vee \neg x = 1,\ x \vee 0 = x,\ x \vee 1 = 1\} \end{aligned}$$

then $\neg a \vee (a \vee b)$ can be reduced to 1 by the reduction $\neg a \vee (a \vee b) \rightarrow_{E_\vee} (\neg a \vee a) \vee b \rightarrow_{E_\vee} 1 \vee b \rightarrow_{E_\vee} 1$, which is a proof for $\mathcal{T}_{\mathbb{B}}(X)/_{\approx_{E_\vee}} \models \neg a \vee (a \vee b) = 1$. Note that the identity $\neg a \vee (a \vee b) = 1$ must hold in every Boolean algebra for reasons given in Example 2.9.

Depending on the set of identities E it is in some cases possible to automatically construct proofs for $s \overset{*}{\leftrightarrow}_E t$ by means of the reduction relation $\rightarrow_E$ and finite descending chains $s \rightarrow_E s_1 \rightarrow_E \cdots \rightarrow_E z$ and $z \leftarrow_E \cdots \leftarrow_E t_1 \leftarrow_E t$. To reason about reductions and reduction relations the following terminology is introduced.

Let $\rightarrow_E$ be a reduction relation, then s is *reducible* iff there is a t such that $s \rightarrow_E t$, otherwise s is *irreducible*, t is the *normal form of* s, denoted as $s \downarrow_E$, iff there is a unique t such that $s \overset{*}{\rightarrow}_E t$, and t is irreducible, t is a *direct successor* of s iff $s \rightarrow_E t$, t is a *successor* of s iff $s \overset{+}{\rightarrow}_E t$, and s and t are *joinable* iff there is a z such that $s \overset{*}{\rightarrow}_E z \overset{*}{\leftarrow}_E t$, in which case we write $s \downarrow_E t$. A reduction $\rightarrow_E$ is called

1. *confluent* iff $t_1 \overset{*}{\leftarrow}_E s \overset{*}{\rightarrow}_E t_2 \Rightarrow t_1 \downarrow_E t_2$.

2. *terminating* iff there is no infinite descending chain $a_0 \rightarrow_E a_1 \rightarrow_E \cdots$

3. *convergent (normalizing)* iff it is both confluent and terminating.

If $\rightarrow_E$ is convergent then for each term t there is a unique smalles element w.r.t. $\rightarrow_E$, its normal form $t \downarrow_E$.

Theorem 3.14
If $\rightarrow_E$ is convergent then $s \overset{*}{\leftrightarrow}_E t$ iff $s \downarrow_E = t \downarrow_E$. (see e.g. [BN98])

If we can compute $s \downarrow_E$ and $t \downarrow_E$, then $\approx_E$ is decidable. This is the case if E is finite and $\rightarrow_E$ is convergent. Term rewriting, which is described next, is concerned with the construction of systems deciding $\approx_E$ by means of the relation $\overset{*}{\leftrightarrow}_E$.

3.2.4 Term Rewrite Systems

Term rewriting relies on the subsequent application of rewrite rules, which are directed identities.

Definition 3.15 (Rewrite Rule and Term Rewrite System)
A *rewrite rule* is a pair $l \rightarrow r$ of terms such that l is not a variable and $Var(r) \subseteq Var(l)$. The terms l and r will be called left- and right-hand-side of the rule, respectively. A set of rewrite rules is called a *term rewrite system* (TRS), and is a binary relation R on terms.

A term l *matches* a term t iff there is a substitution σ, such that $l\sigma \equiv t$. Then t is called an *instance* of l, and σ is called a *match*. The problem of finding matches is the matching problem which has to be solved repeatedly when rewriting a term. A *redex* (reducible expression) is an instance of a left-hand-side of a rule. Given a TRS R, then $\rightarrow_R$ is a reduction relation. By the terminology for reduction relations defined before, a term s reduces to a term t, written $s \rightarrow_R t$ if there is a rewrite rule $l \rightarrow r$, and a position p in s, such that $s|_p$ matches l and $s[l\sigma]_p = t$. If s reduces to t, then $s \rightarrow_R t$ is called a *rewrite step*. The redex of this rewrite step is $l\sigma$. For two terms s and t a convergent

term rewrite system computes normal forms $s \downarrow_R$ and $t \downarrow_R$ such that $(s \downarrow_R) \equiv (t \downarrow_R)$ iff $s \overset{*}{\leftrightarrow}_E t$.[7]

Example 3.16
Given a set of identities E a TRS can be derived from E by directing the identities. Let $E = \{\text{ite}(c, a, a) = a,\ \text{ite}(0, a, b) = b,\ \text{ite}(1, a, b) = a\}$. Then a TRS derived from E is $R = \{\text{ite}(c, a, a) \to a,\ \text{ite}(0, a, b) \to b,\ \text{ite}(1, a, b) \to a\}$. Let $s \equiv \text{ite}(\text{ite}(0, 0, 1), x, y)$ and $t \equiv \text{ite}(z, x, x)$, we want to show that $s \overset{*}{\leftrightarrow}_E t$. Applying the rules R it follows that $s \to_R \text{ite}(1, x, y) \to_R x$, and $t \to_R x$. Since x is irreducible we have $(s \downarrow_R) = x$ and $(t \downarrow_R) = x$, i.e. $s \downarrow_R t$, which implies $s \overset{*}{\leftrightarrow}_E t$.

The TRS in the example above is convergent. However, in general such a derivation does not lead to either confluent, terminating, normalizing, or convergent TRSs.

Example 3.17
Let $E_\vee$ be the set of Boolean identities for $\vee$, i.e.

$$\begin{aligned} E_\vee = \{ \quad & x \vee y = y \vee x,\ x \wedge (y \vee z) = (x \vee y) \vee z, \\ & x \vee \neg x = 1,\ x \vee 0 = x,\ x \vee 1 = 1 \qquad \} \end{aligned}$$

Let R be constructed from E by directing the identities:

$$\begin{aligned} R_\vee = \{ \quad & x \vee y \to y \vee x,\ x \wedge (y \vee z) \to (x \vee y) \vee z, \\ & x \vee \neg x \to 1,\ x \vee 0 \to x,\ x \vee 1 \to 1 \qquad \} \end{aligned}$$

Then $R_\vee$ is not terminating since $\to_{R_\vee}$ allows infinite descending chains $s_0 \to s_1 \to \cdots$ such as $a \vee b \to b \vee a \to a \vee b \to \cdots$.

3.2.5 Termination

Given a finite set of rewrite rules R, it is in general undecidable whether R is terminating or confluent. There are however some criteria from rewriting theory ensuring termination and confluence. As a consequence for some classes of rewrite systems, derived from subsets of $E_{\mathbf{B}}$, the properties of termination and confluence can be established giving rise to practical semi-decision procedures for $\approx_{E_{\mathbf{B}}}$.

In cases where two term s and t cannot be joined by subsequent application of $\to_R$, their normal forms $s \downarrow_R$ and $t \downarrow_R$ are at least simpler than s and t. This can possibly reduce the complexity of subsequently applied semantic decision procedures like ROBDDs and SAT depending on the size of the terms and the number of variables in terms. The idea of simpler can be formalized in the notion of a reduction order on terms.

Definition 3.18 (Reduction Order)
Let $T(X)$ be terms over X with operations F. Let $>$ be a strict order on $T(X)$ that is *compatible with the operations F on $T(X)$* and that is *closed under substitutions*, i.e. for $s_1, s_2 \in T(X)$ with $s_1 > s_2$ we have for all operations $f \in F$, $f(t_1, \ldots, s_1, \ldots, t_n) > f(t_1, \ldots, s_2, \ldots, t_n)$ and for all substitutions $\sigma : X \to T(X)$, $\sigma(s_1) > \sigma(s_2)$. Then $<$ is called rewrite order on $T(X)$. If $<$ is well-founded, it is called *reduction order* on $T(X)$.

[7] Depending on the identities E such term rewrite systems may be difficult to find or even do not exist, e.g. if $\approx_E$ is undecidable. Then $s \overset{*}{\leftrightarrow}_E t$ is undecidable by term rewriting.

Examples 3.19 (Reduction Orders)

1. Size order

   The size of a term t, $|t|$ is the number of positions in t. Let $|t|_x$ denote the number of occurrences of the variable x in t, then the order $>_{sz}$ on $T(X)$ with

   $$s >_{sz} t \Leftrightarrow |s| > |t| \wedge \forall x \in X : |s|_x \geq |t|_x$$

   is a reduction order on $T(X)$.

2. Lexicographic path order

   Let $>$ be a strict order on the function symbols F then the *lexicographic path order* $>_{lpo}$ on terms $T(X)$ induced by $>$ is defined such that $s >_{lpo} t$ iff:

   (a) $t \in Var(s)$ and $s \neq t$, or

   (b) $s \equiv f(s_1, \ldots, s_m)$, $t \equiv g(t_1, \ldots, t_n)$, and

      i. $s_i \geq_{lpo} t$ for some i, or

      ii. $f > g$, and for all j, $s >_{lpo} t_j$, or

      iii. $f = g$, $s >_{lpo} t_j$ for all j, and there is i such that $s_1 = t_1, \ldots, s_{i-1} = t_{i-1}$, and $s_i >_{lpo} t_i$.

For example, $f(x, y) >_{sz} x$, and $f(x, y) >_{lpo} x$, but $f(f(x, y), x) \not>_{sz} f(x, f(y, x))$, though $f(f(x, y), x) >_{lpo} f(x, f(y, x))$.

Other important reduction orders are for example the recursive path order and the Knuth-Bendix order (see e.g. [BN98]). Note that the rank order $>_{rk}$, which is defined such that $s >_{rk} t$ iff $rk(s) > rk(t)$, or if $rk(s) = rk(t)$ and $f > g$, is not a reduction order, since it is not closed under substitutions. Reduction orders have the following important property. If there exists a reduction order $>$ on $T(X)$, and for all rules $l \to r \in R$ we have $l > r$, then R is terminating. For terminating rewrite systems confluence is decidable as well. We will show below, how this can be done.

3.2.6 Confluence

Given a terminating rewrite system R the question of confluence is the question whether the relation $\to_R$ is convergent. Recall that a term s is reducible with a rule $l \to r$, iff there is a position p in s such that $s|_p \equiv l\sigma$ for some substitution σ. If there are two rules $l_1 \to r_1$ and $l_2 \to r_2$ for which s is reducible to t_1 and t_2 this may lead to local divergence of the relation $\to_R$ if $t_1 \not\equiv t_2$.

Examples 3.20 (Local Divergence)
Let $R = \{(x \wedge y) \wedge z \to x \wedge (y \wedge z), w \wedge \neg w \to 1\}$. Then $\to_R$ is locally divergent for the following terms.

1. $a \wedge (\neg a \wedge b) \ \leftarrow_R \ (a \wedge \neg a) \wedge b \ \to_R \ 1 \wedge b$.

2. $a(\wedge \neg(a \wedge b)) \ \leftarrow_R \ (a \wedge b) \wedge \neg(a \wedge b) \ \to_R \ 1$.

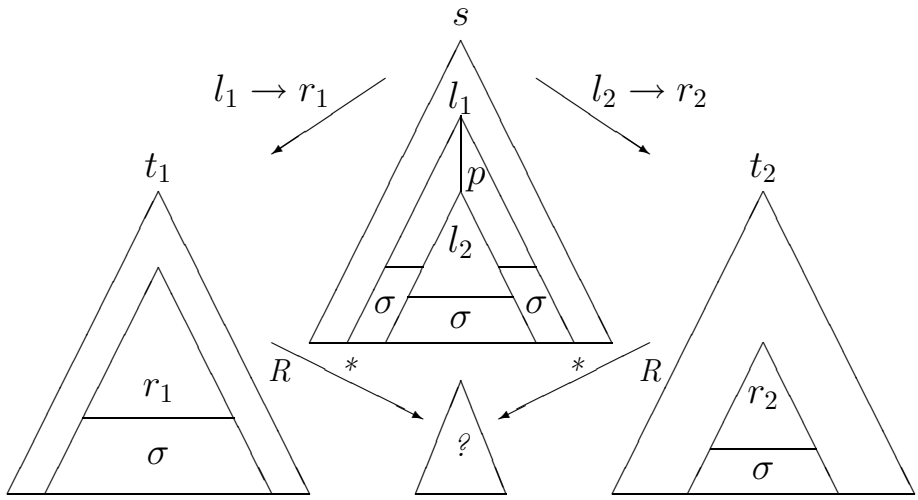


Figure 3.5: Critical Overlap of l_1 and l_2

Obviously there are many different terms for which a pair of rules $l_1 \to r_1$ and $l_2 \to r_2$ may lead to local divergence. If the l_1 and l_2 overlap this is critical for the global confluence, because the reduced terms t_1 and t_2 may not be joinable. In general, l_2 overlaps l_1 if there is a non-variable position $p \in Pos(l_1)$ such that there is a unifying substitution σ with $l_2\sigma \equiv l_1\sigma|_p$. (I.e. $l_2\sigma$ is the subterm of $l_1\sigma$ at p.) The situation is depicted in Figure 3.5. If for all unifying substitutions σ, $r_1\sigma$ and $l_1\sigma[r_2\sigma]$ are joinable, then R is convergent. To establish convergence for R it is sufficient to consider just one such unifying substitution for each pair of rules, the most general one.

Definition 3.21 (Most General Unifier)
Let s and t be terms with $Var(s) \cap Var(t) = \emptyset$. If there exists a substitution σ such that $s\sigma \equiv t\sigma$, then σ is called a *unifier* for s, and t, and s and t are unifiable. A substitution σ is more general than a substitution σ' if there is a substitution δ such that $\sigma' = \delta\sigma$. If σ is more general than any other substitution, then σ is the *most general unifier* $mgu(s,t)$ for s and t.

If s and t are unifiable then there is a $mgu(s,t)$ which is unique up to renaming of variables.

Example 3.22 (Most General Unifier)
For $(x \wedge y) \wedge z \to x \wedge (y \wedge z)$, and $w \wedge \neg w \to 1$ the substitution $\sigma = (x \mapsto a, y \mapsto b, z \mapsto \neg(a \wedge b), w \mapsto (a \wedge b))$ is a unifier. It is easy to see that it is the most general one.

Finding the mgu for s and t is called syntactic unification.[8] We use syntactic unification to compute most general unifiers for a set of rewrite rules. Let $l_1 \to r_1$ and $l_2 \to r_1$ be a pair of rewrite rules such that $Var(l_1) \cap Var(l_2)$ and l_2 overlaps l_1 at a non-variable position $p \in Pos(l_1)$. Let $\sigma = mgu(l_1|_p, l_2)$, then $(r_1\sigma, l_1\sigma[r_2\sigma]_p)$ is called a *critical pair*.

Now we can state the important critical pair lemma.

[8]Syntactic unification can be seen as solving equations since for a set of equations $\{s_1 = t_1, \ldots, s_n = t_n\}$ σ is a solution iff $s_1\sigma = t_1\sigma, \ldots, s_n\sigma = t_n\sigma$.

Lemma 3.23 (Critical Pair Lemma – Knuth-Bendix and Newman's Lemma)
A term rewrite system R is locally confluent iff all its critical pairs are joinable. If R is also terminating it is convergent. (see e.g. [BN98])

Example 3.24
Consider the set of identities on the left hand side of Table 3.2. Orienting these identities from left to right yields a terminating rewrite system (lexicographic path ordering with the order $1 < 0 < \neg < \vee < \wedge$ on the function symbols.) However, it is not confluent, since some critical pairs exist which can not be joined.[9]

3.2.7 Completion

Problems as above can sometimes be solved by completing the set of rules, such that critical pairs become joinable. There are a lot of variants of completion procedures [Bac91]. We only give the simplest version here.

1. Given a set of identities E, determine a reduction order $>$ such that for all identities $s = t \in E$ either $s > t$ or $t > s$ holds. If this is not possible return *fail*, otherwise orient the identities into a set of rules R such that for all rules $l \to r$ we have $l > r$. Then R is terminating.

2. Let $R' := R$. Determine all critical pairs. For all critical pairs (s, t):

   (a) Reduce s and t to s' and t' using R.

   (b) If $s' \equiv t'$, s and t are joinable, thus nothing needs to be done.

   (c) Otherwise, if s' and t' are not comparable under $>$, return *fail*.

   (d) Otherwise, if $s' > t'$, or $t' > s'$ add $s' \to t'$ or $t' \to s'$ to R', respectively.

   If $R = R'$ return *success*, otherwise let $R := R'$, and repeat step 2.

Example 3.25 (Completion)
The set of identities from the last example can be ordered from left to right and can be completed to the term rewrite system of Table 3.2

Note that the completion procedure may not terminate but can generate infinitely many new rules. The procedure fails, if some identities cannot be oriented, i.e. some terms are not comparable by the reduction order. This is particularly the case for AC-theories, i.e. for sets of identities describing associativity and commutativity. In fact there is no terminating TRS for equational theories including associativity and commutativity, even though they are important as Boolean algebra carries this property. This problem can be solved by an extension of term rewriting, called *ordered rewriting*. The idea is that a rewrite step is only admissible if it decreases the term w.r.t. a given reduction order. Let for example $R = \{x * y \to y * x\}$, then $a * b \to_R b * a$ iff $a * b >_{lpo} b * a$.[10]

[9] A confluent term rewrite system is shown on the right-hand side of Fig. 3.2.

[10] Here variables are treated as new free constants, basically meaning that there is a strict order on variables.

Table 3.2: Set of Identities and Resulting Rewrite Rules After Completion

E_n	R_n
	$x \wedge 0 \rightarrow 0$
	$0 \wedge x \rightarrow 0$
	$x \vee 1 \rightarrow 1$
	$1 \vee x \rightarrow 1$
	$1 \wedge x \rightarrow x$
$x \wedge 0 = 0$	$x \wedge 1 \rightarrow x$
$0 \wedge x = 0$	$x \vee 0 \rightarrow x$
$1 \wedge x = x$	$0 \vee x \rightarrow x$
$x \wedge 1 = x$	$\neg(\neg(x)) \rightarrow x$
$x \wedge x = x$	$x \wedge x \rightarrow x$
$x \wedge \neg(x) = 0$	$x \wedge \neg(x) \rightarrow 0$
$\neg(x) \wedge x = 0$	$\neg(1) \rightarrow 0$
$x \vee 0 = x$	$\neg(0) \rightarrow 1$
$0 \vee x = x$	$\neg(x) \wedge x \rightarrow 0$
$x \vee 1 = 1$	$x \vee x \rightarrow x$
$1 \vee x = 1$	$x \vee \neg(x) \rightarrow 1$
$x \vee x = x$	$\neg(x) \vee x \rightarrow 1$
$x \vee \neg(x) = 1$	$x \vee x \wedge y \rightarrow x$
$\neg(x) \vee x = 1$	$x \wedge y \vee x \rightarrow x$
$\neg(\neg(x)) = x$	$x \vee y \wedge x \rightarrow x$
$x \vee (x \wedge y) = x$	$x \wedge y \vee y \rightarrow y$
$(x \wedge y) \vee x = x$	$x \wedge (x \vee y) \rightarrow x$
$x \vee (y \wedge x) = x$	$(x \vee y) \wedge x \rightarrow x$
$(y \wedge x) \vee x = x$	$x \wedge (y \vee x) \rightarrow x$
$x \wedge (x \vee y) = x$	$(x \vee y) \wedge y \rightarrow y$
$(x \vee y) \wedge x = x$	$x \wedge y \wedge x \rightarrow x \wedge y$
$x \wedge (y \vee x) = x$	$x \vee y \vee x \rightarrow x \vee y$
$(y \vee x) \wedge x = x$	$x \wedge y \wedge y \rightarrow x \wedge y$
	$x \vee (x \vee y) \rightarrow x \vee y$
	$x \wedge (x \wedge y) \rightarrow x \wedge y$
	$x \wedge (y \wedge x) \rightarrow y \wedge x$
	$x \vee y \vee y \rightarrow x \vee y$
	$x \vee (y \vee x) \rightarrow y \vee x$

Further, we require that the reduction order is total on ground terms.[11] With ordered rewriting, completion can be extended to unfailing completion, because all identities can now be oriented. For the theory of Boolean rings[12] a set of rewrite rules can be derived by *ordered completion* [Bac91] which is both terminating and confluent. It rewrites terms into polynomial (normal) form, i.e. as sum ($\oplus$) of products ($*$), which is the Boolean ring counter part of CNF/DNF in Boolean algebra. However, in general the normal forms of terms in Boolean rings as well as Boolean algebras have exponential size, which can lead to an explosion of intermediate results even though the normal form of a term may be small. Therefore the application of TRSs deciding Boolean algebra is very limited in practice. Instead usually only a subset of identities is used, as for constant folding below. Term rewriting can also be combined with other decision procedures such as backtrack search.

3.3 Syntactic Preprocessing

Given terms s, t for two Boolean functions h and g, we want to decide whether $h = g$, i.e. whether $s \approx_{E_\mathbf{B}} t$. This is the case iff $E_\mathbf{B} \vdash s = t$.

If for a set of identities $E' \subseteq E_\mathbf{B}$ a syntactic procedure succeeds proving $E' \vdash s = t$ then $E_\mathbf{B} \models s = t$ follows immediately. Thus a decision procedure for $\overset{*}{\leftrightarrow}_{E'}$ is a semi decision procedure for $\approx_{E_\mathbf{B}}$. In fact this is even the case for a semi-decision procedure for $\overset{*}{\leftrightarrow}_{E'}$ such as a terminating (but not necessarily confluent) TRS R' derived from E'.[13] Thus we can use the term rewrite system R' as preprocessor for more powerful decision procedures for the Boolean equivalence problem.

Doing so is motivated by the following observation. ROBDDs can have exponential size in the number of variables. Thus reducing their number decreases their worst case size. Intermediate results in ROBDD construction may have exponential size, even though the resulting function does not. This corresponds to the number of different subterms to be considered. Thus decreasing the number of subterms may decrease the likeliness of this scenario.

The DLL SAT procedure has exponential worst case complexity in the number of variables. Again reducing their number may help. Since SAT is based on CNF and usual terms do not have this form, a CNF representation is constructed by introducing fresh variables at each subterm position. Again, the number of these variables depends on the number of different subterms, and reducing their number may help.

The idea is to reduce the complexity of these procedures by rewriting, i.e. by constructing a TRS R' such that s and t are reduced to $s \downarrow_{R'}$ and $t \downarrow_{R'}$. If $s \downarrow_{R'} \equiv t \downarrow_{R'}$ then $s \approx_{E_\mathbf{B}} t$ follows immediately. Otherwise $s \downarrow_{R'}$ and $t \downarrow_{R'}$ are simpler than s and t and $s \downarrow_{R'} \approx_{E_\mathbf{B}} t \downarrow_{R'}$ should be easier to decide than $s \approx_{E_\mathbf{B}} t$, which is equivalent. Examples for such E' and R' are given below.

Examples 3.26 (Examples for Practical Rewrite Systems)
Let in the following all terms $s, t \in T_\mathbb{B}(X)$, where $T_\mathbb{B}(X)$ is the set of Boolean terms together with the extensions made to the syntax in 2.2. We want to simplify s and t using

[11] This is the case for the lexicographic path ordering.
[12] The theory of $*$ (multiplication or $\wedge$) and $\oplus$ (exclusive-or or addition modulo 2).
[13] For two terms s and t, $s \downarrow_{R'} \equiv t \downarrow_{R'}$ implies $E_\mathbf{B} \vdash s = t$ implies $s \approx_{E_\mathbf{B}} t$.

a rewrite system R, and then decide whether $s \approx_{E_{\mathbf{B}}} t$ by comparing $s \downarrow_R$ and $t \downarrow_R$ using a decision procedure from 3.1. In cases where $s \downarrow_R$ and $t \downarrow_R$ are not identical they should at least be simpler than s and t.

1. Constant folding:

   The idea of *constant folding* is to take a set of ground identities E_G derived from a set of identities E and direct these identities into a rewrite system R.

   Consider two terms $s \equiv 0 \wedge (1 \vee 0)$ and $t \equiv 0 \vee (1 \wedge 0)$. For all combinations of $x, y \in \mathbb{B}$ and for a binary operation f we can derive either $f(x, y) = 0$ or $f(x, y) = 1$ from the Boolean identities $E_{\mathbf{B}}$, i.e.

   $$\begin{aligned} E = \{ \quad & 0 \wedge 0 = 0, 0 \wedge 1 = 0, 1 \wedge 0 = 1, 1 \wedge 1 = 1, \\ & 0 \vee 0 = 0, 0 \vee 1 = 1, 1 \vee 0 = 1, 1 \vee 1 = 1 \quad \} \end{aligned}$$

   These identities can be directed into a rewrite system R from left to right. It is easy to see by inspecting the rules R that there are no infinite descending chains of derivations, and that there is a unique normal form for each term. (Formally termination is ensured by the size order. Furthermore the left-hand side of the rules are ground and they do not overlap therefore there cannot be any critical pairs, which ensures (local) confluence.)

   Applying this set of rules onto s and t yields 0 in both cases, thus inferring $s \approx_{E_{\mathbf{B}}} t$. The rewrite system R can of course be applied to non-ground terms as well, for example $s' \equiv (0 \wedge 1) \vee x$ and $t' \equiv (0 \vee 1) \wedge x$ yields $0 \vee x$ and $1 \wedge x$. However, they cannot be joined, even though it is easy to see that $s' \approx_{E_{\mathbf{B}}} t'$. Note that this does not contradict confluence, it just tells us that R is not strong enough to decide all identities. At least the terms $0 \vee x$ and $1 \wedge x$ are simpler than s' and t'.

2. Constant propagation:

   The idea of *constant propagation* is quite similar to constant folding. Here not all but only some parts of an identity are required to be ground.[14] In the last example there is still one constant left in both $s' \downarrow_R$ and $t' \downarrow_R$ cases. We will develop a stronger rewrite system, based on constant propagation to take care of such situations. Again the idea is to derive a set of identities E' from $E_{\mathbf{B}}$ and convert them into a rewrite system. From the identities for the Boolean $\wedge$- and $\vee$-operations we receive:

   $$\begin{aligned} R' = \{ \quad & x \wedge 0 \to 0, 0 \wedge x \to 0, 1 \wedge x \to x, x \wedge 1 \to x \\ & x \vee 1 \to 1, 1 \vee x \to 1, 0 \vee x \to x, x \vee 0 \to x \quad \} \end{aligned}$$

   Now we have $0 \vee x \to_{R'} x \leftarrow_{R'} 1 \wedge x$ Again, R' is terminating since for all rules we have $l >_{sz} r$.[15] There are no overlapping rules in R'.[16] Therefore R is confluent.[17]

[14] Constant folding can be seen as a special case of constant propagation.

[15] This is also the case for $R \cup R'$, however a lot of rules in R are subsumed by R'.

[16] There are unifiable left-hand-sides, for example $x \wedge 0$ and $w \vee 1$ are unifiable with $\sigma = (x \mapsto a \vee 1, w \mapsto a)$. However, this is not an overlap, since x is a variable position.

[17] The same holds for $R \cup R'$.

3. Usual rewriting:

   The terms $(y \vee y)$ and $(x \vee y) \wedge y$ are both irreducible w.r.t. the rules R and R' above. However, obviously $(y \vee y) \approx_{E_{\mathbf{B}}} (x \vee y) \wedge y$ The rewrite system R_n in Table 3.2 rewrites these terms to y.

Reducing the number of variables in the term and reducing the number of different subterms can reduce the complexity for both SAT and ROBDDs. The number of variables is an obvious measure. The number of different subterms can be reduced by increasing sharing, i.e. by rewriting equivalent subterms such that they are isomorphic. The constant propagation rules facilitate this goal, since rule application either removes a subterm, and thus variables, or it rewrites a term to one of its subterms, which may increase sharing, but at least does not decrease it. Furthermore it potentially rewrites two distinct terms to identical normal forms. Therefore it is an appropriate preprocessor for both ROBDD and CNF-SAT based semantic approaches.

Note that stronger rewriting and thus better normalization does not necessarily lead to better sharing. For example, applying the law of associativity for $\wedge$ as rule $(a \wedge b) \wedge c \rightarrow a \wedge (b \wedge c)$ reduces the sharing in the term $(x \wedge y) \vee ((x \wedge y) \wedge z)$, where $(x \wedge y)$ occurs in two subterms, which is reduced to $(x \wedge y) \vee (x \wedge (y \wedge z))$, where $(x \wedge y)$ occurs in one subterm. Therefore, rewriting has to be applied in a sensible way.

Chapter 4

Verification of Sequential Circuits

This chapter reviews some property checking techniques for verifying functional aspects of block-sized sequential circuits. First the notions of hardware designs and their correctness are clarified. The limitations of current simulation based approaches for their functional verification are illustrated. Then symbolic model checking (SMC) and bounded model checking (BMC) are outlined. Finally, we introduce bounded interval model checking (BIMC) as variation of BMC, and discuss its advantages and drawbacks compared to the standard. Just like BMC, it is based on solving Boolean equivalence and satisfiability problems, applying techniques described in the previous chapter. The BIMC approach is extended to the register transfer level in Chapter 8.

4.1 Sequential Circuits as Finite State Machines

Informally, a sequential (digital) circuit is some piece of hardware, reading binary values from its input leads, driving binary values to its output leads, and storing data internally in latches, holding binary values. The output depends on the input and the internal state, which is constituted by the values stored in its latches. The internal state may change depending on the current state and input.

Example 4.1 (D-FF)
As an example consider a simple D-flip-flop (D-FF). Its logic symbol is depicted in Fig. 4.1. Its input signals are **d** (data) and **c** (clock), while the output signals are **q** and **q**′.

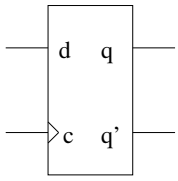


Figure 4.1: D-Flip-Flop

On a positive clock edge (a $0 \rightarrow 1$ transition of **c**), the D-flip-flop reads the value of **d** and stores it internally. The stored value is propagated to the output signal **q**. The output signal **q**′ is the inverted value of **q**.

The current output and the next state are functions of the current input and the current state. Thus the behavior of a sequential circuit can be described by sequences of inputs, outputs and states. To formally verify its behavior, a formal model is needed. Driven by this intuitive notion, *finite state machines* (FSMs), and in particular Mealy machines, are commonly used as formal models of sequential circuits.

Definition 4.2 (Mealy Machine)
A *Mealy machine* is a tuple

$$\mathcal{M} = (I, O, S, S_0, f_O, f_S)$$

where I is the *set of inputs*, O is the *set of outputs*, S is the *set of states*, $S_0 \subseteq S$ is the *set of initial states*, $f_O : I \times S \to O$ is the *output function*, and $f_S : I \times S \to S$ is the *transition function* of $\mathcal{M}$. A sequential digital circuit with n inputs, m outputs, and k latches can be represented explicitly by a Mealy machine

$$\mathcal{M} = (\mathbb{B}^n, \mathbb{B}^m, \mathbb{B}^k, S_0 \subseteq \mathbb{B}^k, f_O, f_S)$$

with $f_O : \mathbb{B}^n \times \mathbb{B}^k \to \mathbb{B}^m$, and $f_S : \mathbb{B}^n \times \mathbb{B}^k \to \mathbb{B}^k$.

Representing states, transition functions and output functions of Mealy machines explicitly by sets and maps is not very practical for computational purposes, as sequential circuit can have many thousand states. Therefore, output and transition functions of Mealy machines representing sequential circuits are usually given constructively by Boolean terms which have their usual meaning. We will call such a constructive representation of a sequential circuit a hardware design[1].

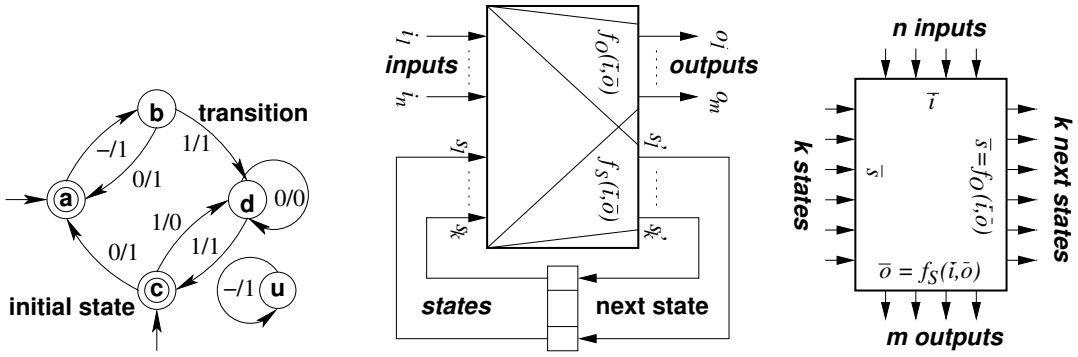
Definition 4.3 ((Digital Hardware) Design)
A constructive representation for a Mealy machine $\mathcal{M}$ as above is a tuple

$$\mathcal{D} = (\overline{\imath}, \overline{o}, \overline{s}, f_{S,0}, f_O, f_S)$$

called a *(digital hardware) design*, where:

1. $\overline{\imath} = (i_1, \ldots, i_n)$ is the *vector of (Boolean) input variables*,
2. $\overline{o} = (o_1, \ldots, o_m)$ is the *vector of (Boolean) output variables*,
3. $\overline{s} = (s_1, \ldots, s_k)$ is the *vector of (Boolean) state variables*,
4. $f_{S,0}$ is a characteristic Boolean function *describing initial states*, i.e. for $\overline{s} \in \mathbb{B}^k$, $f_{S,0}(\overline{s}) = 1$ iff $\overline{s} \in S_0$, represented by a Boolean term with the same name, over the state variables $\overline{s}$.
5. f_O is the output function of the Mealy machine, represented (component wise) by a vector of Boolean terms $(f_O^1, \ldots, f_O^m)$ over the input and state variables, i.e. $(f_O^1(\overline{\imath}, \overline{s}), \ldots f_O^m(\overline{\imath}, \overline{s})) = f_O(\overline{\imath}, \overline{s})$ for all $\overline{\imath} \in \mathbb{B}^n$, $\overline{s} \in \mathbb{B}^k$.

[1] This is consistent with common practice, since such representations are computed from HDL descriptions of a sequential circuits, also called designs.



a) State Transition Diagram b) Block Diagram c) Hardware Design

Figure 4.2: Different Representations of Sequential Circuits as FSM

6. f_S is the transition function of the Mealy machine, represented (component wise) by a vector of Boolean terms $(f_S^1, \ldots, f_S^k)$ over the input and state variables, i.e. $(f_S^1(\overline{\imath}, \overline{s}), \ldots f_S^k(\overline{\imath}, \overline{s})) = f_S(\overline{\imath}, \overline{s})$ for all $\overline{\imath} \in \mathbb{B}^n$, $\overline{s} \in \mathbb{B}^k$.

Mealy machines and hardware designs are representations of the same thing, a finite state machine (FSM) for a sequential circuit. A typical block-sized circuit comprises a few hundred up to several thousands of state variables and is specified by a few hundred lines of HDL source code.

A Mealy machine can be depicted as state transition diagram (Fig. 4.2.a), such that states are represented by nodes, while transition and output function are represented by arrows between these nodes. The arrows are annotated with an input and an output. (Initial states are only marked if $S_0 \neq \mathbb{B}^k$.) Hardware designs are usually depicted as in Fig. 4.2.b. However, we prefer to use the equivalent representation in Fig. 4.2.c.

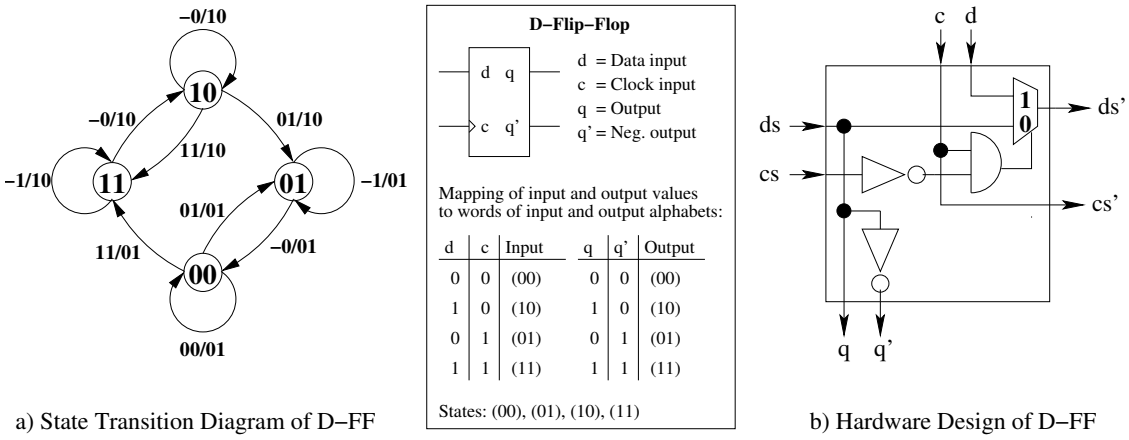
The possible combinations for the values of the n Boolean input, m Boolean output signals, and k latches of a design correspond to the $\mathbb{B}^n$ inputs, $\mathbb{B}^m$ outputs and $\mathbb{B}^k$ states of the Mealy machine, respectively. The correspondence between the values of input signals, output signals and latches of the design are described by the Boolean output and transition functions f_O, and f_S, respectively. Obviously assignments to the input, output and state variables $\overline{\imath}$, $\overline{o}$ and $\overline{s}$ of the design correspond to inputs, output and state I, O, and S of the Mealy machine, respectively. The value of the output and transition function of the Mealy machine is then a valuation of the terms for output and transition function of the design. Hence we can, and will, switch between both concepts as needed and call both variants a FSM for a sequential circuit.[2]

Example 4.4 (Finite State Machine for D-FF)
Let $\{\mathbf{d}, \mathbf{c}, \mathbf{q}, \mathbf{q}', \mathbf{ds}, \mathbf{cs}\}$ be a set of Boolean variables, denoting the input, and output signals and latches of the D-FF. The following Mealy machine is a formal model of the D-Flip-Flop:

$$\mathcal{M}_{\text{D-FF}} \;=\; (\mathbb{B}^2,\ \mathbb{B}^2,\ \mathbb{B}^2,\ \mathbb{B}^2, f_O : \mathbb{B}^2 \times \mathbb{B}^2 \to \mathbb{B}^2, f_S : \mathbb{B}^2 \times \mathbb{B}^2 \to \mathbb{B}^2)$$

It has 4 inputs, 4 outputs, and 4 states. All states are initial states. Fig. 4.3.a shows the state transition diagram of $\mathcal{M}_{\text{D-FF}}$. The correspondence between the values of input and

[2]In some sense, Mealy machines and designs are a semantic and the syntactic view on FSMs, respectively.



a) State Transition Diagram of D−FF

b) Hardware Design of D−FF

Figure 4.3: Graphical Representations of D-FF

output signals to inputs and states $I = \mathbb{B}^2$ and $S = \mathbb{B}^2$ is shown in the tables of Fig. 4.3. For example an input (01) denotes that $\mathbf{d} = 0$ and $\mathbf{c} = 1$. Initial states are not specially marked in this example, since all states are considered to be initial states. The output and transition functions f_O and f_S can be derived from this graphical representation. A constructive representation is the design

$$\mathcal{D}_{\text{D-FF}} = ((\mathbf{d}, \mathbf{c}), (\mathbf{q}, \mathbf{q}'), (\mathbf{ds}, \mathbf{cs}), 1, f_O, f_S)$$

depicted in Fig. 4.3.b, where the output and transition functions f_O and f_S are defined component wise by f_O^1, f_O^2 and f_S^1, f_S^2 below.

$$\begin{aligned} f_O^1 &\equiv \mathbf{ds} \\ f_O^2 &\equiv \neg\mathbf{ds} \\ f_S^1 &\equiv \text{ite}(\neg\mathbf{cs} \wedge \mathbf{c}, \mathbf{d}, \mathbf{ds}) \\ f_S^2 &\equiv \mathbf{c} \end{aligned}$$

4.2 Sequential Behavior

The output function of a FSM maps current inputs and current states to current outputs, while the transition function maps current inputs and current states to next states. The notion of 'current' and 'next' is based on a discrete notion of time. If t denotes the current point in time, then $\mathsf{t} + 1$ denotes the next point in time. The outputs at time t and the states at time $\mathsf{t} + 1$ depend on the inputs and states at t. Computation starts in one of the initial states S_0 at time $\mathsf{t} = 0$. In general the set of initial states S_0 is a subset of all possible states, i.e. $S_0 \subseteq \mathbb{B}^k$, but often $S_0 = \mathbb{B}^k$ holds. We will assume that a set of initial states is given. Let in the following $\mathsf{i}, \mathsf{j}, \mathsf{t} \in \mathbb{N}_0$ denote time points, and for a variable x let x_t denote the value of x at time t. Let $\overline{\imath}$, $\overline{o}$, $\overline{s}$ be the vectors of input, output and state variables of a design, then their corresponding values at time t are denoted as $\overline{\imath}_\mathsf{t}$, $\overline{o}_\mathsf{t}$, and $\overline{s}_\mathsf{t}$, respectively. Intuitively, computation sequences of designs are sequences of inputs, outputs and states respecting the output and transition function, usually starting with an initial state. Thus, the sequential behavior of circuits is characterized by the set of possible computation sequences of its FSM model.

Definition 4.5 (Computation Sequence)
Let $\mathcal{M} = (I, O, S, S_0, f_O, f_S)$ be a Mealy machine. An infinite sequence of inputs, outputs and states $(\overline{\imath}_\mathsf{t}, \overline{o}_\mathsf{t}, \overline{s}_\mathsf{t})_{\mathsf{t}\in\mathbb{N}_0}$ with $\overline{\imath}_\mathsf{t} \in I$, $\overline{o}_\mathsf{t} \in O$, $\overline{s}_\mathsf{t} \in S$ is a *computation sequence* of $\mathcal{M}$, iff

1. $\forall \mathsf{t} \in \mathbb{N}_0 : \overline{o}_\mathsf{t} = f_O(\overline{\imath}_\mathsf{t}, \overline{s}_\mathsf{t})$
2. $\forall \mathsf{t} \in \mathbb{N}_0 : \overline{s}_{\mathsf{t}+1} = f_S(\overline{\imath}_\mathsf{t}, \overline{s}_\mathsf{t})$

Consequently the same sequence $(\overline{\imath}_\mathsf{t}, \overline{o}_\mathsf{t}, \overline{s}_\mathsf{t})_{\mathsf{t}\in\mathbb{N}_0}$ is a computation sequence of the corresponding design $(\overline{\imath}, \overline{o}, \overline{s}, f_{S,0}, f_O, f_S)$. If $\overline{s}_0 \in S_0$ (resp. $f_{S,0}(\overline{s}_0) = 1$) then the computation sequence starts with an initial state.

Depending on a given state $\overline{s} \in S$ and a sequence of inputs $(\overline{\imath}_\mathsf{t})_{\mathsf{t}\in\mathbb{N}_0}$ the corresponding computation sequence of a design can be computed iteratively.

Example 4.6 (Computation Sequences of D-FF)
The set of all possible computation sequences of the D-FF is characterized by the following identities, holding for all time points $\mathsf{t} \in \mathbb{N}_0$:

$$\begin{aligned}
\mathbf{q}_\mathsf{t} &= \mathbf{ds}_\mathsf{t} \\
\mathbf{q}'_\mathsf{t} &= \neg\mathbf{ds}_\mathsf{t} \\
\mathbf{ds}_{\mathsf{t}+1} &= \mathrm{ite}(\neg\mathbf{cs}_\mathsf{t} \wedge \mathbf{c}_\mathsf{t}, \mathbf{d}_\mathsf{t}, \mathbf{ds}_\mathsf{t}) \\
\mathbf{cs}_{\mathsf{t}+1} &= \mathbf{c}_\mathsf{t}
\end{aligned}$$

The start of a computation sequence of the D-FF, starting in (00), i.e. $\mathbf{ds}_0 = 0$, $\mathbf{cs}_0 = 0$, is shown in the assignment Table 4.1.

Table 4.1: Start of a Computation Sequence of $\mathcal{D}_{\text{D-FF}}$

t	0	1	2	3	4	5	6	7	8	$\cdots$
$\mathbf{d}_\mathsf{t}$	0	1	1	0	0	0	0	0	0	$\cdots$
$\mathbf{c}_\mathsf{t}$	0	0	1	1	0	0	1	0	0	$\cdots$
$\mathbf{q}_\mathsf{t}$	0	0	0	1	1	1	1	0	0	$\cdots$
$\mathbf{q}'_\mathsf{t}$	1	1	1	0	0	0	0	1	1	$\cdots$
$\mathbf{ds}_\mathsf{t}$	0	0	0	1	1	1	1	0	0	$\cdots$
$\mathbf{cs}_\mathsf{t}$	0	0	0	1	0	0	0	1	0	$\cdots$

The set of all computation sequences starting from a single state can be depicted as computation tree (Fig. 4.4) which is a directed tree where nodes are labeled with states, and there is an edge from $\overline{s}_\mathsf{t}$ to $\overline{s}_{\mathsf{t}+1}$ if there is an input such that a transition from $\overline{s}_\mathsf{t}$ to $\overline{s}_{\mathsf{t}+1}$ will occur[3]. Given an initial states $\overline{s}_0$, not all states S may be reachable. A state $\overline{s}$ is reachable from $\overline{s}_0$ if there is a computation sequence starting with $\overline{s}_0$, containing this state, i.e. if there is a time point t such that $\overline{s} = \overline{s}_\mathsf{t}$. In the state transition graph this is the case if there is a path from a node with label $\overline{s}_0$ to $\overline{s}$.

[3]This representation abstracts away the details of inputs and outputs, only considering reachability of states, as defined below. However it is possible to construct an equivalent FSM, a so called Kripke structure, from a Mealy machine, where possible inputs and outputs are encoded into states, treating a combination of all system variables, i.e. inputs, outputs and latches, as one state of this FSM. For the D-FF this leads to $2^6 = 64$ states.

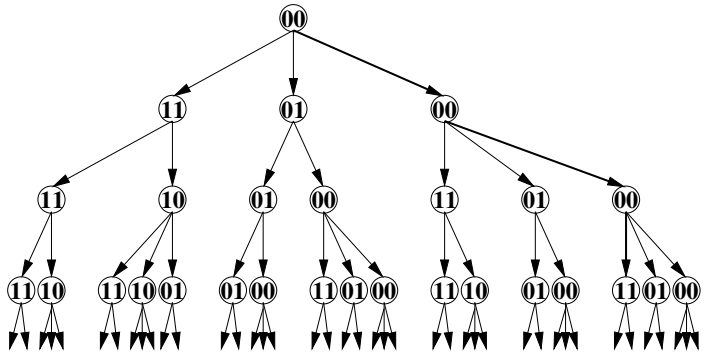


Figure 4.4: Infinite Computation Tree for D-FF

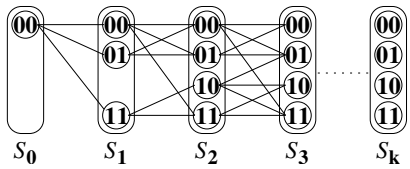


Figure 4.5: Sequence of Reachable State Sets for D-FF

Considering sets of states as possible starts of a computation sequence and all possible input assignments, this leads to sets of computation sequences. In general reachability from a set of states is defined as follows.

Definition 4.7 (Reachable State)
Let $\mathcal{M} = (I, O, S, S_0, f_O, f_S)$ be a Mealy machine. A state $\overline{s} \in S$ is *reachable from a set of states* $X \subseteq S$ iff there is a computation sequence $(\overline{\imath}_\mathsf{t}, \overline{o}_\mathsf{t}, \overline{s}_\mathsf{t})_{\mathsf{t}\in\mathbb{N}_0}$ starting in X and a time point $\mathsf{t} \in \mathbb{N}_0$ such that $\overline{s} = \overline{s}_\mathsf{t}$.

Considering sets of reachable states this leads to sequences $(S_\mathsf{t})_{\mathsf{t}\in\mathbb{N}_0}$ of states, where the set of states reachable from a given set of states is closed under reachability. This means that there are no states reachable from this set, which are not elements of this set.

Definition 4.8 (Closed Under Reachability)
Let $X \subseteq S$ be a set of states of a Mealy machine $\mathcal{M} = (I, O, S, S_0, f_O, f_S)$. The set of states reachable from X is defined as $X^* := \{\overline{s} \in S : \overline{s} \text{ is reachable from } X\}$. X is *closed under reachability* iff $X = X^*$.

The notions concerned with reachability in Mealy machines carry over to designs as usual.

Example 4.9 (Reachability)
For the D-FF all states are initial states and all states are reachable from any other state as can be observed in the state transition diagram. If we consider (00) as the only initial state, a computation tree for the D-FF is depicted in Fig. 4.4. Even though we consider only one initial state (00), all states are still reachable, leading to the following sequence of sets of states ($S_0 = \{(00)\}, S_1 = \{(00), (01), (11)\}, S_2 = \{(00), (01), (10), (11)\}, S_3 = \{(00), (01), (10), (11)\}, \ldots$), as depicted in Fig. 4.5. Here S_0 and S_1 are not closed under reachability, while S_2 and the following are.

Why is it necessary to consider sets of initial states? Mealy machines and hardware designs representing sequential circuits have sets of initial states, since in reality, after power on, the latches of a circuit have random values. Therefore, basically all states can be initial states. So why not consider all states to be initial states in general? Sometimes it is useful to consider just a subset of all these states as initial states. This is motivated by the following observation:

In practice, after power on a reset sequence is applied to the inputs of a sequential circuit in order to force it into some predefined state. This state is usually a reset state of the design.[4] After the reset sequence has been applied normal operation starts. To verify correct reset behavior, all states must be treated as initial states of the Mealy machine. To verify normal behavior after reset, it is sufficient to consider the reset state as initial state. Therefore, the set of initial states considered depends on the verification problem to be solved.

If the computation sequence of a Mealy machine, i.e. a transition from $\overline{s}_{\mathtt{t}}$ to $\overline{s}_{\mathtt{t}+1}$, is correlated to some clock, it is said to be synchronous. Then the outputs for a clock cycle depend on the inputs and states at the clock cycle before. Note that the Mealy machine of the D-Flip-Flop is not synchronous since the clock input $\mathbf{c}$ is a normal input. However it is possible to compute a synchronous Mealy machine of the D-Flip-Flop from the Mealy machine given before, where the clock input is not reflected by any input. Instead the rising edges of $\mathbf{c}$ are correlated with $\mathtt{t}$. (This is done by applying so called clocking scheme onto $\mathcal{M}_{\text{D-FF}}$, where all inputs are kept stable while the clock changes.)

Some authors prefer the use of transition relations instead of transition functions, which is more general because it allows to conveniently deal with asynchronous feedback loops in circuits and composition of modules. However, after a fixed point computation the transition relations can be transformed into functions, if a fixed point exists, which is the case for most circuits. (Otherwise their behavior would be unstable.) Therefore the use of a functional model imposes no restrictions in practice.

4.3 Correct Behavior

Intuitively a sequential circuit is correct if it behaves as expected. To verify the correct behavior the expected behavior has to be specified first. Most specification methods rely on a set of properties which have to be fulfilled by the design. First these properties are verified for the design. Then, in subsequent steps of the design process (synthesis, layout, etc.), the compliance with the specification is established by (formal) equivalence checking against the verified design.

Example 4.10 (Expected Behavior of the D-FF)
The basic function of a D-FF can be specified informally by the following properties:

1. On a positive edge of the clock signal $\mathbf{c}$, the data input $\mathbf{d}$ is propagated to the output $\mathbf{q}$.

[4] Formally a reset state of a hardware design is a state which is reachable from all states under a fixed input sequence, a reset sequence. If a design has a reset state it is resettable. The question whether a design is resettable and of computing a reset sequence is a verification problem as not all designs have this property.

2. The output $\mathbf{q}$ is stable unless a positive clock edge occurred.

3. The output $\mathbf{q}'$ is always the negation of $\mathbf{q}$.

A specification as above has to be verified to ensure the functional correctness of the design. Since its sequential behavior is characterized by the set of possible computation sequences, it behaves as expected if a given computation sequence matches an expected (specified) sequence of inputs and outputs. It is correct if it behaves as expected for all possible computation sequences. Hence the specification and a formal model for the circuits sequential behavior, i.e. a Mealy machine or a design, have to be related and checked.

4.4 Validation

Expected behavior can be specified explicitely by a set of expected computation sequences. According to common practice, only a few of all possible computation sequences are extracted from the informal specification. Note that the number of all possible sequences is far too large to be considered exhaustively, even for small designs.

Example 4.11 (An Expected Sequences of D-FF)
For the simple D-FF an expected sequence extracted from the informal specification above is shown in Table 4.2. Here '-' denotes unspecified values. The values specified above the line are inputs and states assumed to occur. The values below the line represent the expected output. As can be seen this sequence matches the computation sequence shown

Table 4.2: An Expected Computation Sequence of $\mathcal{D}_{\text{D-FF}}$

t	0	1	2	3
$\mathbf{d}_t$	0	1	1	-
$\mathbf{c}_t$	0	0	1	-
$\mathbf{ds}_t$	0	-	-	-
$\mathbf{cs}_t$	0	-	-	-
$\mathbf{q}_t$	-	-	-	1
$\mathbf{q}'_t$	-	-	-	0

in Table 4.1. However this sequence covers only a small portion of the possible behavior of the design as can be observed in the computation tree for the D-FF.

Validating expected computation sequences for designs is done practically using simulation. Simulation is a fully automated process and can therefore be used effectively by hardware designers. With simulation some input values, along with the expected output values of a design are specified. This is done for a sequence of input and output values. In each cycle a simulator calculates the values of the outputs from the specified input values and the latches. Then the calculated and the expected output values are compared, and violations are reported.

Unfortunately simulation is far from being exhaustive. The set of specified input values covers only a small portion of all assignments, even for relatively small designs. Since

simulation is incomplete some critical misbehavior might stay undetected. Therefore, the required level of correctness cannot be produced by simulation. On top of that, specifying the desired behavior by examples is cumbersome, time consuming, and error-prone in itself. Test automation tools [spe], which became popular lately, try to ease this problem by using symbolic specifications instead of examples. However, the fundamental problem of incompleteness remains.

The desired correctness of design blocks can be reached by using formal methods. (See Chapter 1 for a detailed analysis.) Formal methods are sophisticated, highly automated mathematical proof methods. They provide exhaustive, i.e. complete functional verification.

4.5 Property Checking

Instead of specifying the expected computation sequences of a design one by one, the idea is to describe the expected behavior of a design by a set of properties which must be satisfied for all possible computation sequences of the design and at all times. This can be checked automatically using model checking tools. In contrast to simulation the expected behavior of a design is formally described by a set of temporal formulas, called properties. Such properties can be checked by tools called property or model checkers. If a property does not hold, the model checker produces a refutation sequence (counter-example) driving the circuit into a state where the property does not hold. (For a thorough description see for example [Eme94].) There are two classes of properties, *safety and liveness properties*, where the first class describes that never anything bad happens, and the latter requires that something good happens again and again (infinitely often). In some sense properties are an orthogonal view on FSMs, since a sufficient set of them explicitly describes the behavior of an FSM, while the FSM itself is an implicit description of its own behavior.

For model checking usually FSMs are used as *models*. A state of the FSM represents one configuration of values for all system variables, and are annotated with elementary properties of the system, e.g. idle, start, busy. These elementary properties are called *atomic propositions*.

Example 4.12
The system variables of a hardware design are inputs, outputs and states. In case of the D-FF these are $\mathbf{d}$,$\mathbf{c}$, $\mathbf{ds}$, $\mathbf{cs}$, $\mathbf{q}$ and $\mathbf{q}'$ constituting $2^6 = 64$ states. Note that this is a different notion of FSM than the Mealy machine model, now incorporating the environment.

An elementary property of such an FSM is for example $\mathbf{q} \wedge \mathbf{q}'$, asserting that the values of $\mathbf{q}$ and $\mathbf{q}'$ are 1 in each state of the model with this label. (Note that all states with this property are unreachable from the initial state $(0, 0, 0, 0, 0, 1)$, where $\neg\mathbf{d}$, $\neg\mathbf{c}$, $\neg\mathbf{ds}$, $\neg\mathbf{cs}$, $\neg\mathbf{q}$, and $\mathbf{q}'$ hold.)

4.5.1 Temporal Logic

Properties of such models are described by *temporal logic formulas*. Temporal logic is a special kind of formal logic which is especially suited for the specification of sequential behavior of systems modeled as FSMs. Various temporal operators are used to describe how the truth values of assertions (atomic propositions of the model) change over time.

sometimes p (Fp) always p (Gp) next time p (Xp) p until q (pUq)

Figure 4.6: Intuitive Meaning of Path Formulas

They allow to specify transitions between states without explictly describing the temporal aspects, like referring to particular time points in the computation sequence, instead treating them symbolically. Temporal logic formulas are either state formulas, describing situations at a particular state within the computation tree (see Fig. 4.4), or path formulas, describing properties along particular paths through the computation tree (i.e. computation sequences) of the model considered.

The basic elements of temporal logic are atomic propositions of the model. Combining them using Boolean operations yield atomic propositions. There are two kinds of formulas, namely *state formulas* and *path formulas*. A state formula is either true or false in a state. Each atomic proposition is a state formula. The change of the truth values of a state formula along computation sequences is described by path formulas.

If p is an atomic proposition, then Fp (sometimes/eventually p), Gp (always p), Xp (next time p) and, if q is another atomic proposition, pUq (p until q) are path formulas, and F, G, X and U are called *temporal operators*. Their intended meaning is illustrated in Fig. 4.6 where p is an atomic proposition (or in general a state formula). The application of Boolean connectives and temporal operations to path formulas yield path formulas.

The validity of path formulas can be restricted by path quantifiers, namely A, the universal path quantifier, and E, the existential path quantifier. The universal path quantifier describes that the following path formula holds for all paths in the computation tree, while the existential path quantifier states that there is a path in the computation tree, where the path formula holds.

Applying a path quantifier to a path formula yields a state formula. E.g. if p is an atomic proposition then $AG(p)$ (p holds for all states along all traces), $AF(p)$ (p holds for some state along all traces), $EG(p)$ (p holds at all states along some trace) and $EF(p)$ (p holds at some state in some trace) are state formulas, with the intended meaning shown in Fig. 4.7. Boolean connection of state formulas yield state formulas. (E.g. $AG(p \Rightarrow AF(q)) :=$ whenever p holds, q will hold at some time.) A state formula is true for a design, iff it is true for all initial states of the FSM model.

The unrestricted combination of linear time operators and path quantifiers and their Boolean connection yields a temporal logic called CTL^*. (I.e. atomic propositions are state formulas. Applying the operations F, G, X, U to state formulas yields path formulas. Applying path quantifiers (A, E) to path formulas yields state formulas. Respective Boolean combinations of atomic propositions, state or path formulas are atomic propositions, state or path formulas, respectively.)

Interesting subsets of this logic are

1. *Propositional linear time temporal logic* (PLTL), or *LTL* for short, where temporal formulas are build from atomic propositions, temporal operators and Boolean connectives such that an LTL formula is a universally quantified path formula. An LTL formula holds for a state iff it is true for all computation sequences starting in this state. It is true for a design iff it is true for all initial states.

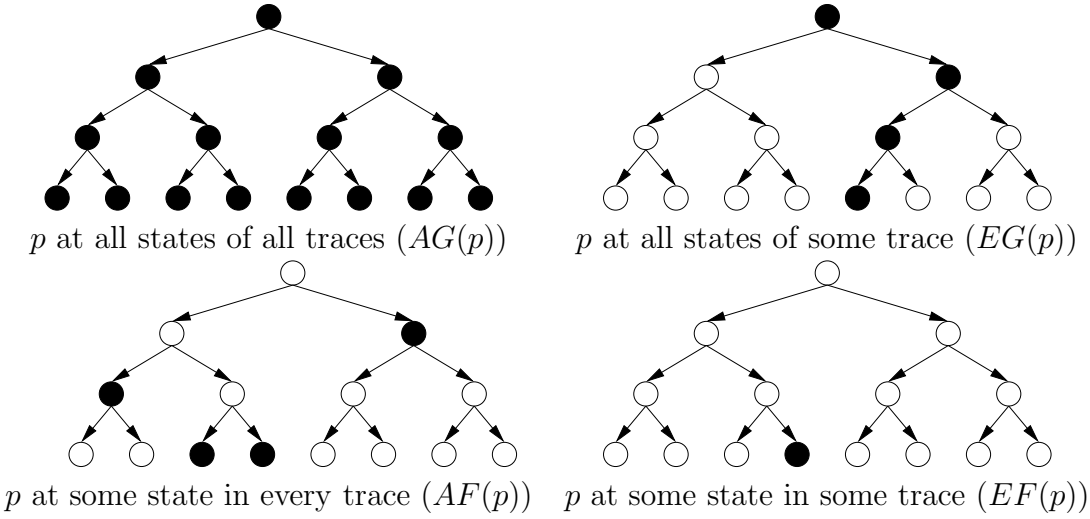


Figure 4.7: Intuitive Meaning of Some State Formulas

2. *Computation tree logic* (*CTL*) where quantification is restricted to immediate application to temporal operators (constituting the CTL operations AG, AF, AX, AU, EG, EF, EX and EU).

3. ACTL which is CTL in positive normal form, i.e. where only universal path quantification is allowed.

Example 4.13
The three properties of the D-FF design can be expressed by the following temporal logic formulas:

1. $AG((\neg\mathbf{c} \wedge X\mathbf{c}) \Rightarrow X((\mathbf{d} \wedge X\mathbf{q}) \vee (\neg\mathbf{d} \wedge X\neg\mathbf{q})))$, stating that on a positive clock edged the data appears on the output in the next step,

2. $AG((\mathbf{c} \vee X\neg\mathbf{c}) \Rightarrow X(\mathbf{q} = X\mathbf{q}))$, stating that the output $\mathbf{q}$ is stable unless a positive clock edge occurred before, and

3. $AG((\mathbf{q} \wedge \neg\mathbf{q}') \vee (\neg\mathbf{q} \wedge \mathbf{q}'))$, stating that $\mathbf{q}$ and $\mathbf{q}'$ are mutual exclusive.

LTL and CTL represent two possible views regarding the underlying nature of time, namely linear time, where at each moment there is only one possible future, and branching time, where at each moment, time may split into different possible futures. Each of them is important to model checking of hardware designs in its own regard and comprises different computational power. Note that for the model checking approaches based on CTL*, (LTL and CTL) the considered computation sequences always start at a specific time point, mostly 0, which is called the anchor. Hence CTL and LTL are anchored temporal logics. (We will consider a non-anchored version of LTL, interval temporal logic (ITL), later.)

4.5.2 Model Checking Procedures

Checking whether a temporal logic formula is true for a given model can be done fully automatic using a *model checker*. In our case this means that a property holds for a

design. In case that a property does not hold, the model checker computes a refutation sequence, i.e. a path through the computation tree, for which the property is violated. (Here only CTL model checking is described.)

Explicit Model Checking

Given a state transition diagram of an FSM, labeled with atomic propositions, and a CTL formula f to be checked, a naive approach to model checking is based on induction over the structure of the formula and propagation of formula labels within the state transition diagram.

1. Starting from an innermost state formula, which is not atomic, e.g. AFp, where p is an atomic proposition do the following:
2. For all states s labeled with p do:
   (a) Label s with AFp. (Obviously AFp holds in s whenever p holds.)
   (b) Label all states from which s is reachable with AFp.
3. All states not labeled with AFp are labeled with $\neg AFp$. (For the other operators the algorithm is similar.)
4. Take a part of the formula for which all sub formulas are already treated and repeat the procedure until the whole formula is treated, i.e. states are labeled either with f or $\neg f$.

Obviously the above algorithm involves computing sets of forward and backward (with respect to the transition relation) reachable states. If Z is a set of states the set of states reachable in one step from Z under the transition relation R is the image of Z under R, while all states from which Z is reachable in one step is the pre-image or inverse image of Z, i.e.

$$Img(Z, R) := \{s' : \exists s \in Z : (s, s') \in R\}$$

$$Img^{-1}(Z, R) := \{s' : \exists s \in Z : (s', s) \in R\}$$

Note that if P is a set of states labeled with an atomic proposition p, then $Img^{-1}(P, R)$ is the set of states where EXp holds (which is usually denoted as $EX\ P$). The set of states reachable from a given set of states is then computed as follows. Let $Z_0 := Z$, and $Z_{i+1} := Z_i \bigcup Img(Z_i, R)$ Z_n is the set of states reachable from Z iff $Z_n = Z_{n+1}$, i.e. we reached a fixed point. The model checking algorithm above relies on an explicit representation of the FSM as state transition diagram, which has exponential size in the number of system variables making this approach almost impractical.

Symbolic Model Checking

The idea of *symbolic model checking* (SMC) is to represent the transition relation R, and characteristic functions for sets of states as BDDs. Then computation of images, sets of reachable states can be done by BDD operations (logic And, and Boolean quantification) on the characteristic functions. Furthermore, CTL operations have a fixed point

characterization, allowing to iteratively compute sets of states satisfying a CTL formula using BDDs. (Note that this approach strongly relies on the cheap test, whether two BDDs represent the same function.) Given BDDs for the characteristic functions f_p and f_q for sets of states labeled with atomic propositions p, and q, the sets of states labeled with $\neg p$, $p \wedge q$, and $p \vee q$ can be computed directly using BDD operations on f_p and f_q, e.g. $f_{\neg p} := \neg f_p$. For EXp and $AXp = \neg EX \neg p$ the pre-image of f_P can be used, i.e. $f_{EXp} := Img^{-1}(f_p, f_R)$. All other operations can be characterized by greatest or least fixed points of monotonic functions, e.g. $EFp = p \vee EX\ EF\ p$

Thus an explicit representation of states and state transitions can be avoided. The method is called symbolic since the characteristic functions for sets of states are represented symbolically, i.e. by BDDs. Using SMC much larger state sets can be handled than using explicit MC. However, the algorithm needs potentially an exponential number of iterations in the number of state variables. The complexity of the image computation is worst case exponential, and representing characteristic functions by BDDs may require an exponential amount of memory (also in the number of state variables). Unfortunately these worst case scenario do occur frequently in practice and lead to unpredictable runtime behavior of the model checker.

Bounded Model Checking

To avoid the unpredictable memory consumption of SMC and potentially use other function representations than BDDs the idea of bounded model checking (BMC) is to bound the number of fixed point iterations by a fixed limit k. If a fixed point is reached or a violation is found within this bound, the procedure is equivalent to SMC. Otherwise no decision can be made. (However increasing the bound and restarting the model checker is an option.)

Given a symbolic representation of a FSM, a LTL formula Gp to be checked this corresponds to finding a witness (counter-example) satisfying $F\neg p$. In BMC path quantifiers are used to indicate whether a path formula is expected to hold along all paths (A) or only some paths (E). Given a user supplied time bound k, a propositional formula f' is constructed which is satisfiable if a path formula f holds along some computation sequence of the FSM (i.e. if Ef holds). The satisfiability of f' can be checked for example using SAT or BDDs. The methods relies on the observation that if a witness exists it can be characterized by a finite sequence of states. Let $f_{S,0}(\overline{s})$, and $f_R(\overline{s}, \overline{s}')$ be characteristic functions for the set of initial states and the transition relation of the FSM (not necessarily given as BDDs), where $\overline{s}$ denotes current states and $\overline{s}'$ denotes next states. (In case that the transition relation is a function as required for our Mealy machines, just take $\overline{s}' = f_S(\overline{s})$.) The transition relation is unrolled as follows:

$$f_{R,k} := f_{S,0}(\overline{s}_0) \wedge \bigwedge_{i=0}^{k} f_R(\overline{s}_\mathsf{i}, \overline{s}_{\mathsf{i}+1})$$

The property f is also unrolled. Let for example $f = EFp$ and $k = 2$ then $f_2 = (p(\overline{s}_0) \vee p(\overline{s}_1) \vee p(\overline{s}_2)$. If f_k represents the unrolled property, then the $f' = f_{R,k} \wedge f_k$. The construction above allows to check safety properties (AGp) if the bound k is large enough. Successively increasing the bound one yields a semi-decision procedure for safety properties. For liveness (AFp) the translation is similar. A formula f' is constructed for

a given bound k such that if f' is valid then AFp holds. Again there is a k such that if AFp holds then f' is valid. The validity of f' can be checked by BDD or SAT techniques. Subsequently increasing k yields a semi-decision procedure for liveness properties. The combination of both semi-decision procedures will terminate eventually since either AGp or $EG\neg p$ must hold.

4.6 Bounded Interval Model Checking

Here bounded interval model checking is proposed as formal method for functional verification on the block level. The verification procedure is described without justifying its correctness.

First bounded interval temporal properties are introduced. To check if such a property holds for a design, they have to be related with the possible computation sequences of the design. This is done using temporal formulas, characterizing the set of possible computation sequences, and substituting them into a property recursively. If the resulting Boolean term represents the constant function 1 then the property holds for the design.

4.6.1 Bounded Interval Properties

First bounded interval properties are introduced here as formal specification of expected behavior.

Let $\mathcal{D} = (\overline{\imath}, \overline{o}, \overline{s}, f_{S,0}, f_O, f_S)$ be a design. Now let t be a symbol denoting an arbitrary time point. Let $\mathsf{n} \in \mathbb{N}$ be fixed, and let $\overline{\imath}_{\mathsf{t}+0}, \ldots, \overline{\imath}_{\mathsf{t}+\mathsf{n}}$, $\overline{o}_{\mathsf{t}+0}, \ldots, \overline{o}_{\mathsf{t}+\mathsf{n}}$, and $\overline{s}_{\mathsf{t}+0}, \ldots, \overline{s}_{\mathsf{t}+\mathsf{n}}$ be variables denoting values of $\overline{\imath}$, $\overline{o}$, and $\overline{s}$ within the finite discrete time interval $[\mathsf{t}, \mathsf{t}+\mathsf{n}]$ of length $\mathsf{n}+1$, respectively. Note that the length of this interval is independent of the choice of t.

An expected computation sequence of a design can be described by relating the values of input, output and state variables within such a finite time interval. We call the interval $[\mathsf{t}, \mathsf{t}+\mathsf{n}]$ *observation window*, and t the start of the observation.

Example 4.14 (Semi-Formal Specification of D-FF)
The expected computation sequences for the D-FF can be described as follows. For any given point in time $\mathsf{t} \in \mathbb{N}_0$ the following requirements must be satisfied:

1. Assume $\mathbf{c}_\mathsf{t} = 0$, and $\mathbf{c}_{\mathsf{t}+1} = 1$, then $\mathbf{q}_{\mathsf{t}+2} = \mathbf{d}_{\mathsf{t}+1}$.

2. Assume $\mathbf{c}_\mathsf{t} = 1$ or $\mathbf{c}_{\mathsf{t}+1} = 0$ then $\mathbf{q}_{\mathsf{t}+2} = \mathbf{q}_{\mathsf{t}+1}$

3. Always $\mathbf{q}'_\mathsf{t} = \neg\mathbf{q}_\mathsf{t}$.

The observation windows are $[\mathsf{t}, \mathsf{t}+2]$, and $[\mathsf{t}, \mathsf{t}]$, respectively. The first requirement states that after a rising edge of the clock input $\mathbf{c}$, the data output $\mathbf{q}$ will show the same value as the data input $\mathbf{d}$ at the time of the rising clock edge. The second requirement states that the output $\mathbf{q}'$ is always the negation of the output $\mathbf{q}$. Note, that in contrast to the computation sequence specified earlier in Table 4.2, here the values of inputs, outputs and states are variable.

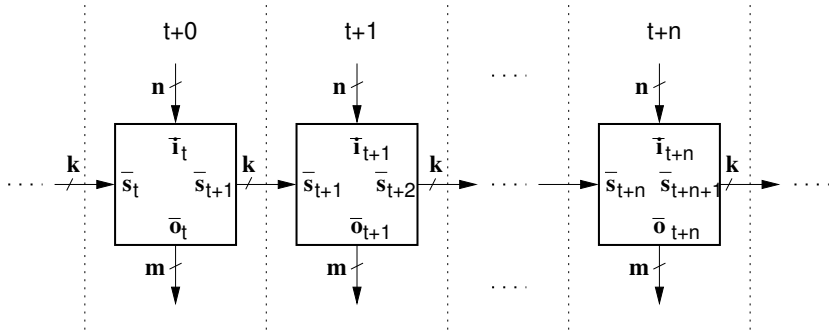


Figure 4.8: Unrolled a Design for a Finite Observation Window

Such relations can be compactly described by Boolean terms over input, output and state variables within the observation window. They are called bounded interval properties.

Definition 4.15 (Bounded Interval Property)
Let $\mathcal{D} = (\overline{\imath}, \overline{o}, \overline{s}, f_O, f_S, f_{S,0})$ be a design. Let t be a symbol for an arbitrary time point, and let $n \in \mathbb{N}$. Then a Boolean term p over the timed input, output and state variables within the interval $[\mathsf{t}, \mathsf{t}+\mathsf{n}]$ is called *bounded interval property for* $\mathcal{D}$.

In the following, bounded interval properties will be simply called properties.

Example 4.16 (Specification of D-FF by Properties)
The following properties describe the expected behavior of the D-FF:

1. $p_1 \equiv (\mathbf{c}_\mathsf{t} = 0 \wedge \mathbf{c}_{\mathsf{t}+1} = 1) \Rightarrow (\mathbf{q}_{\mathsf{t}+2} = \mathbf{d}_{\mathsf{t}+1})$
2. $p_2 \equiv \mathbf{q}'_\mathsf{t} = \neg \mathbf{q}_\mathsf{t}$

A property holds for a design iff it holds for all computation sequences of the design an for all possible starting points $\mathsf{t} \in \mathbb{N}_0$ of the observation window. (The intuition is that the formula holds along all traces of the computation tree and at all nodes.)

4.6.2 Constructing Bounded Interval Model Checking Problems

To prove that a property holds for a design, a Boolean term is constructed from the property and the Boolean terms for output and transition function of the design. (These terms characterize the possible behavior of the design.) If this term represents the constant function 1 the property holds for the design.

The idea of this construction is based on the following observation. Within a finite window of time $[\mathsf{t}, \mathsf{t}+\mathsf{n}]$, the outputs at an intermediate time point $\mathsf{t}+\mathsf{i}$ and the states at $\mathsf{t}+\mathsf{i}+1$ depend on the states and inputs at $\mathsf{t}+\mathsf{i}$. Therefore the computations of the design within a finite observation window $[\mathsf{t}, \mathsf{t}+\mathsf{n}]$ can be thought of as the design, unrolled n-times over its next-states, as depicted in Fig. 4.8.

Properties can be thought of as combinational circuits as shown in Fig. 4.9. Given a property with an observation window $[\mathsf{t}, \mathsf{t}+\mathsf{n}]$, the identities characterizing the computation sequences of a design are substituted recursively into this property. Then the

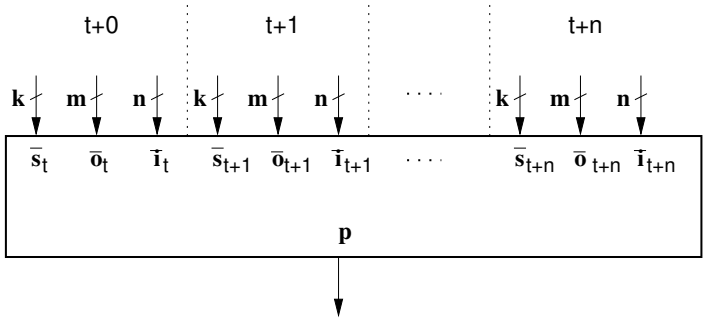


Figure 4.9: A Property as Combinational Circuit

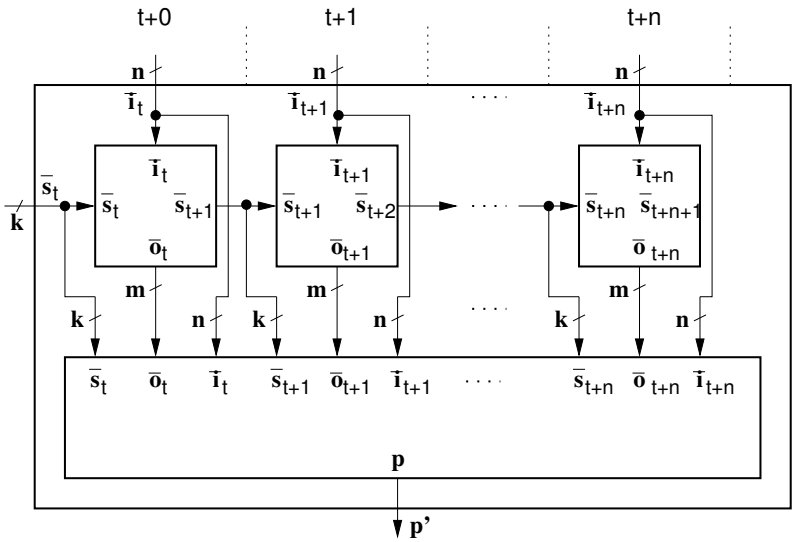


Figure 4.10: A Bounded Model Checking Problem as Combinational Circuit

resulting term has only variables $\overline{\imath}_{\mathsf{t}}, \overline{\imath}_{\mathsf{t+1}}, \ldots, \overline{\imath}_{\mathsf{t+n}}$, and $\overline{s}_{\mathsf{t}}$. Thus the represented function only depends on the inputs over the observation window and the states at the start of the observation window. If we allow all possible assignments for the state variables at the start of the observation window, all previous behavior is covered. If we allow all possible assignments to the input variables within the observation window, all possible input behavior is covered. Thus all possible computation sequences of the design are treated at once.

The recursive substitution can be thought of as plugging the inputs, outputs, and states of the design within the observation window into the property circuit, as shown in Fig. 4.10. The resulting circuit represents the bounded model checking problem. If the output p' is equal to 1 for all possible inputs of the circuit, then the property holds for the design.

Now the procedure for the construction of a bounded model checking problem is explained in a more formal way.

Let $\mathcal{D} = (\overline{\imath}, \overline{o}, \overline{s}, f_{S,0}, f_O, f_S)$ be a design. Let $f_O = (f_O^1, f_O^2, \ldots, f_O^m)$ and $f_S = (f_S^1, f_S^2, \ldots, f_S^k)$ be the vectors of output and transition terms of the design $\mathcal{D}$.

Let $\mathsf{t} + \mathsf{i} \in \mathbb{N}_0$ be a time point. The vectors of output and transition terms $f_{O,\mathsf{t+i}}$

and $f_{S,\mathsf{t+i}}$ at time $\mathsf{t+i}$ are defined as follows. The vector of output terms at $\mathsf{t+i}$ is $f_{O,\mathsf{t+i}} = (f^1_{O,\mathsf{t+i}}, f^2_{O,\mathsf{t+i}}, \ldots, f^m_{O,\mathsf{t+i}})$, with $f^l_{O,\mathsf{t+i}} := f^l_O[\overline{\imath}/\overline{\imath}_{\mathsf{t+i}}][\overline{s}/\overline{s}_{\mathsf{t+i}}]$. The vector of transition terms at $\mathsf{t+i}$ is $f_{S,\mathsf{t+i}} := (f^1_{S,\mathsf{t+i}}, f^2_{S,\mathsf{t+i}}, \ldots, f^k_{S,\mathsf{t+i}})$, with $f^j_{S,\mathsf{t+i}} := f^j_S[\overline{\imath}/\overline{\imath}_{\mathsf{t+i}}][\overline{s}/\overline{s}_{\mathsf{t+i}}]$.

Let p be a property with observation window $[\mathsf{t}, \mathsf{t+n}]$. Let $p_{\mathsf{t+n}} := p$. Starting with $\mathsf{i} = \mathsf{n}$ and ending with $\mathsf{i} = 1$ the output variables at time $\mathsf{t+i}$ and state variables at $\mathsf{t+i+1}$ occurring in p_{t+i}, are replaced with the corresponding output and transition terms at $\mathsf{t+i}$, respectively.

Using this approach, the term p_{t} is constructed from $p_{\mathsf{t+n}}$ as follows: For $\mathsf{i} \in \mathbb{N}_{\mathsf{n}}$, let $p_{\mathsf{t+i-1}} := p_{\mathsf{t+i}}[\overline{o}_{\mathsf{t+i}}/f_{O,\mathsf{t+i}}][\overline{s}_{\mathsf{t+i}}/f_{S,\mathsf{t+i-1}}]$. Then the term p_{t} with $p_{\mathsf{t}} = p_{\mathsf{t+0}}$ contains only state variables at t ($\overline{s}_{\mathsf{t}}$) and input variables within $[\mathsf{t}, \mathsf{t+n}]$, $(\overline{\imath}_{\mathsf{t}}, \ldots, \overline{\imath}_{\mathsf{t+n}})$. The remaining variables are unconstrained. It can be shown formally that if the resulting term is constantly 1, the property holds for all possible computation sequences of the design.

In the following, the term p_{t} constructed above is called *bounded model checking problem.*

Example 4.17
The possible computation sequences of the D-FF are characterized by the following identities, holding for all time points $\mathsf{t+i} \in \mathbb{N}_0$.

$$\begin{aligned}
\mathbf{q}_{\mathsf{t+i}} &= \mathbf{ds}_{\mathsf{t+i}} \\
\mathbf{q}'_{\mathsf{t+i}} &= \neg\mathbf{ds}_{\mathsf{t+i}} \\
\mathbf{ds}_{\mathsf{t+i+1}} &= \mathrm{ite}(\neg\mathbf{cs}_{\mathsf{t+i}} \wedge \mathbf{c}_{\mathsf{t+i}}, \mathbf{d}_{\mathsf{t+i}}, \mathbf{ds}_{\mathsf{t+i}}) \\
\mathbf{cs}_{\mathsf{t+i+1}} &= \mathbf{c}_{\mathsf{t+i}}
\end{aligned}$$

Consider the property p_1 for the D-FF in Example 4.16. The observation window is $[\mathsf{t}, \mathsf{t}+2]$. The bounded model checking problem p_{t} is constructed from the property p_1 as follows.

First we set $p_{\mathsf{t+2}} := p_1 \equiv (\mathbf{c}_{\mathsf{t}} = 0 \wedge \mathbf{c}_{\mathsf{t+1}} = 1) \Rightarrow (\mathbf{q}_{\mathsf{t+2}} = \mathbf{d}_{\mathsf{t+1}})$. Then we iterate through $\mathsf{i} = 2$ down to $\mathsf{i} = 1$, and construct the intermediate results $p_{\mathsf{t+1}}$ and $p_{\mathsf{t+0}}$. The term $p_{\mathsf{t+0}}$ is the final result, the bounded model checking problem.

1. Let $\mathsf{i} = 2$. Then $p_{\mathsf{t+1}} = p_{\mathsf{t+2}}[\overline{o}_2/f_{O,2}][\overline{s}_2/f_{S,2}]$. The only output variable with index $t+2$ is $\mathbf{q}_{\mathsf{t+2}}$. We already know that $\mathbf{q}_{\mathsf{t+2}} = \mathbf{ds}_{\mathsf{t+2}}$. Then we have

   $$p_{\mathsf{t+2}}[\overline{o}_2/f_{O,2}] \equiv (\mathbf{c}_{\mathsf{t}} = 0 \wedge \mathbf{c}_{\mathsf{t+1}} = 1) \Rightarrow (\mathbf{ds}_{\mathsf{t+2}} = \mathbf{d}_{\mathsf{t+1}})$$

   There is one state variable with index $\mathsf{t+2}$, $\mathbf{ds}_{\mathsf{t+2}}$, and we know that $\mathbf{ds}_{\mathsf{t+2}} = \mathrm{ite}(\neg\mathbf{cs}_{\mathsf{t+1}} \wedge \mathbf{c}_{\mathsf{t+1}}, \mathbf{d}_{\mathsf{t+1}}, \mathbf{ds}_{\mathsf{t+1}})$ Then $p_{\mathsf{t+1}} = p_{\mathsf{t+2}}[\overline{o}_2/f_{O,2}][\overline{s}_2/f_{S,2}]$ is given by:

   $$p_{\mathsf{t+1}} \equiv (\mathbf{c}_{\mathsf{t}} = 0 \wedge \mathbf{c}_{\mathsf{t+1}} = 1) \Rightarrow (\mathrm{ite}(\neg\mathbf{cs}_{\mathsf{t+1}} \wedge \mathbf{c}_{\mathsf{t+1}}, \mathbf{d}_{\mathsf{t+1}}, \mathbf{ds}_{\mathsf{t+1}}) = \mathbf{d}_{\mathsf{t+1}})$$

2. Let $\mathsf{i} = 1$. There are no output variables with index $\mathsf{t+1}$, but the state variables $\mathbf{cs}_{\mathsf{t+1}}$, and $\mathbf{ds}_{\mathsf{t+1}}$ with index $\mathsf{t+1}$ in $p_{\mathsf{t+1}}$. Then

   $$\begin{aligned} p_{\mathsf{t+0}} \equiv \quad & (\mathbf{c}_{\mathsf{t}} = 0 \wedge \mathbf{c}_{\mathsf{t+1}} = 1) \Rightarrow \\ & (\mathrm{ite}(\neg\mathbf{c}_{\mathsf{t}} \wedge \mathbf{c}_{\mathsf{t+1}}, \mathbf{d}_{\mathsf{t+1}}, \mathrm{ite}(\neg\mathbf{cs}_{\mathsf{t}} \wedge \mathbf{c}_{\mathsf{t}}, \mathbf{d}_{\mathsf{t}}, \mathbf{ds}_{\mathsf{t}}))) = \mathbf{d}_{\mathsf{t+1}}) \end{aligned}$$

Now p_{t} with $p_{\mathsf{t}} = p_{\mathsf{t+0}}$ is the bounded model checking problem. If the formula $p_{\mathsf{t}} = 1$ is valid in the Boolean algebra $\mathcal{A}_\beta$, then the property $p_1 \equiv (\mathbf{c}_{\mathsf{t}} = 0 \wedge \mathbf{c}_{\mathsf{t+1}} = 1) \Rightarrow (\mathbf{q}_{\mathsf{t+2}} = \mathbf{d}_{\mathsf{t+1}})$ holds for the D-FF design. (The converse holds as well, as explained below.)

If the property p' does not hold, there is an assignment to the variables in the observation window, such that $p' = 0$. Such an assignment is called a counter-example for the property p. There are several possible reasons for counter-examples.

1. The property does not describe the correct behavior. Then it should be changed.

2. The property describes the correct behavior, but the design is incorrect. Then the design should be changed.

3. The counter-example is unreachable.

The first two cases are the usual cases encountered during verification of a design block using bounded model checking. The third case requires more explanation, which is given below.

4.6.3 Reachability of Counter-Examples

If bounded model checking problems are constructed as above, a property may hold for a design, even though the bounded model checking problem is invalid. This is the case because the set of states considered at the beginning of the observation window is the set of all states. However, it may be the case that not all states are reachable from the initial states of the design. Then the set of all states is only an approximation of the set of reachable states. This approximation may be too weak. Therefore the bounded model checking problem may be invalid for unreachable states.

This problem can be tackled as follows. If there is a formula describing the set of reachable states, it can be added to the bounded model checking problem in the following way. Let p_R be a formula, describing the set of reachable states for a design $\mathcal{D}$, let p be a property and let p' be the bounded model checking problem, then the property p holds for the design $\mathcal{D}$, iff the formula $p_R \Rightarrow p'$ holds. The problem with this approach is that such a formula p_R is in general hard to find.

If such a formula cannot be computed with reasonable effort, an approximation of the set of reachable states can be used instead. Let $p_{R'}$ be a formula which describes a good approximation of the set of reachable states, which is closed under reachability (See Definition 4.8.). Then if $p_{R'} \Rightarrow p'$ holds, the property p holds for the design. If the set of states described is a good approximation of the set of reachable states, then unreachable counter-examples are unlikely.

If $p_{R'}$ describes the set of all states then it is equivalent to the formula 1. In this case the approach is equivalent to the one described before, and p' is checked for validity.

4.6.4 Solving Bounded Interval Model Checking Problems

To prove that a property p holds for a design we have to show that $\mathcal{T}_{\mathbf{B}}(X) \models p' = 1$ which is a Boolean equivalence problem in normal form as in 3.1.1. It can be solved semantically as in 3.1, syntactically as in 3.2 or as combination thereof as in 3.3, possibly employing preprocessing techniques. In the next section we will develop a new preprocessing technique for the Boolean equivalence problem which is especially well suited for verification of properties of regular designs.

4.6.5 Remarks on Bounded Interval vs. LTL BMC

Compared to LTL bounded model checking, interval temporal logic has some advantages.

1. The operations of an ITL property are bound and allow to express bounded until, bounded finally and bounded globally, but their validity is unbound. Therfore it is possible to verify unbounded safety properties, since ITL is not anchored at the initial state of the design.

2. The implicit computation of the cone of influence of a property avoids the construction of so called symmetric parts of the search space by leaving out unreferenced parts of the output and transition function as well as inputs and states during the construction of the formula to be checked. (This approach can possibly be applied to LTL based BMC as well.)

3. From the author's perspective ITL properties are much easier to grasp by average hardware designers usually unfamiliar with the concepts of infinite sequences and fixed points, which are required for understanding and writing LTL properties. There is no intrinsic use of temporal operators and reachable state sets in ITL properties. (However, it would be possible to add bounded versions of them as syntactic sugar.)

4. Unbounded liveness properties can be shown by induction on the meta level without the need for computing fixed points for reachable state sets.

The biggest disadvantage of this approach is the possibility for false negatives, i.e. unjustified counter examples due to unreachable states. However, this problem can be attacked using approximations of the reachable state set as additional constraints as described before.

Chapter 5

Symmetry Reduction on the Bit-Level

In this chapter, the basic idea for symmetry reduction for Boolean equivalence problems is explained. First, we define equivalent values, and show how they can be exploited for preprocessing when proving some Boolean function to be constantly 1. Semi-decision procedures for the Boolean equivalence problem can be used for finding such equivalent values. Then equivalence of values is extended to symmetry, leading to the permutation equivalence problem of Boolean terms, which has to be solved for finding symmetrical values. It is shown, how symmetrical values can be exploited for preprocessing in conjunction with decision procedures for the Boolean equivalence problem. This approach is extended to bitvector terms in Chapter 10.

5.1 Equivalent Values

Let $f \in \mathbb{B}^{\mathbb{B}^n}$ be an n-ary Boolean function. The values 0 and 1 are equivalent for x_i in f iff the cofactors $f|_{x_i=0}$ and $f|_{x_i=1}$, with

$$f|_{x_i=0} := f(x_1, \ldots, x_{i-1}, 0, x_{i+1}, \ldots, x_n)$$

$$f|_{x_i=1} := f(x_1, \ldots, x_{i-1}, 1, x_{i+1}, \ldots, x_n)$$

are identical. (We use sometimes use f_x ($f_{\bar{x}}$) instead of $f|_{x=1}$ ($f|_{x=0}$) to ease readability.)

Example 5.1 (Equivalent Values)
As an example consider the function g:

$$\begin{aligned} g : \mathbb{B} \times \mathbb{B} &\rightarrow \mathbb{B} \\ g(x, y) &\mapsto (x \wedge \neg y) \vee (\neg x \wedge \neg y) \end{aligned}$$

Here 0 and 1 are equivalent values for x in g, since $g(0, y) = g(1, y)$ holds for all y, i.e. $g(0, 0) = g(1, 0)$ and $g(0, 1) = g(1, 1)$, i.e. $g_x = g_{\bar{x}}$, as shown below. (Note that the values 0 and 1 are not equivalent for y in g.)

$$\begin{aligned} g_x = &\ (1 \wedge \neg y) \vee (\neg 1 \wedge \neg y) &= \neg y \\ g_{\bar{x}} = &\ (0 \wedge \neg y) \vee (\neg 0 \wedge \neg y) &= \neg y \end{aligned}$$

The values 0 and 1 for a variable x in some function f over variables X correspond to a partial assignment to the variables X, namely $(x \mapsto 0)$ and $(x \mapsto 1)$, which are considered to be equivalent iff 0 and 1 are equivalent values for x in f.

If, instead of value pairs for one variable, sets of variables (and the corresponding combinations of values) are considered, this leads to a more general notion of equivalence of values:

Definition 5.2 (Equivalent Value Vectors)
Let $f : \mathbb{B}^n \to \mathbb{B}$ be a Boolean function in n variables X. Let $X_k \subseteq X$ with $X_k = \{x_1, \ldots, x_k\}$. Let $a_1, \ldots, a_k, b_1, \ldots, b_k \in \mathbb{B}$ The vectors $(a_1, \ldots, a_k), (b_1, \ldots, b_k) \in \mathbb{B}^k$ are called *equivalent value vectors* for the vector of variables $(x_1, \ldots, x_k)$ w.r.t. f, iff

$$f|_{x_1=a_1,\ldots,x_k=a_k} = f|_{x_1=b_1,\ldots,x_k=b_k}$$

Here $f|_{x_1=a_1,\ldots,x_k=a_k}$ and $f|_{x_1=b_1,\ldots,x_k=b_k}$ are *k-th order cofactors*[1] *of* f. By the definition above, $\varphi, \varphi' : X^k \to \mathbb{B}$ are considered to be *equivalent assignments* for X_k w.r.t. f iff $\varphi(x_i) = a_i$ and $\varphi'(x_i) = b_i$. Equivalent values for some variables in some function, if known, can be used for preprocessing when deciding the Boolean equivalence problem, as shown below.

5.2 Cofactor Expansion

Let in the following $f : \mathbb{B} \times \cdots \times \mathbb{B} \to \mathbb{B}$ be an n-ary Boolean function in n variables $X = \{x_1, \ldots, x_n\}$. Consider the question whether $f = 1$ holds. Obviously, this is the case iff both cofactors of f w.r.t. to some variable x are constantly 1, i.e. iff $f_x = 1$ and $f_{\bar{x}} = 1$. Hence we have:

$$\begin{aligned} & f = 1 \\ \Leftrightarrow \quad & f_x = 1 \ \wedge \ f_{\bar{x}} = 1 \\ \Leftrightarrow \quad & f_x = 1 \ \wedge \ f_x = f_{\bar{x}} \\ \Leftrightarrow \quad & f_{\bar{x}} = 1 \ \wedge \ f_x = f_{\bar{x}} \end{aligned}$$

If the values 0 and 1 are equivalent for x in f, then $f_x = f_{\bar{x}}$ follows. If this is the case it is sufficient to consider either $f_x = 1$ or $f_{\bar{x}} = 1$ in order to decide whether $f = 1$ holds.

Example 5.3
In Example 5.1 the values 0 and 1 for x in g, and thus the cofactors g_x and $g_{\bar{x}}$, are the same. Then only one cofactor of g w.r.t. x has to considered, to decide whether $g = 1$ holds. Consider for example $g_x \equiv (1 \wedge \neg y) \vee (\neg 1 \wedge \neg y) = \neg y$. Obviously, $\neg y = 1$ is invalid since it is false for $y = 1$. Therefore, $g = 1$ does not hold.

If it is not known, whether two cofactors are identical, $f_x = 1$ and $f_{\bar{x}} = 1$ can each be expanded again into cofactors w.r.t. another variables, say y. Then the following propositions are equivalent:

[1] See Shannon expansion Section 3.1.3.

$$
\begin{aligned}
& f = 1 \\
\Leftrightarrow\ & f_{xy} = 1 \ \wedge\ f_{\bar{x}y} = 1 \ \wedge\ f_{x\bar{y}} = 1 \ \wedge\ f_{\bar{x}\bar{y}} = 1 \\
\Leftrightarrow\ & f_{xy} = 1 \ \wedge\ f_{xy} = f_{\bar{x}y} = f_{x\bar{y}} = f_{\bar{x}\bar{y}} \\
\Leftrightarrow\ & f_{x\bar{y}} = 1 \ \wedge\ f_{xy} = f_{\bar{x}y} = f_{x\bar{y}} = f_{\bar{x}\bar{y}} \\
\Leftrightarrow\ & f_{\bar{x}y} = 1 \ \wedge\ f_{xy} = f_{\bar{x}y} = f_{x\bar{y}} = f_{\bar{x}\bar{y}} \\
\Leftrightarrow\ & f_{\bar{x}\bar{y}} = 1 \ \wedge\ f_{xy} = f_{\bar{x}y} = f_{x\bar{y}} = f_{\bar{x}\bar{y}}
\end{aligned}
$$

In general, let $F_{X'}$ be the set of all k-th order cofactors of f w.r.t a set of variables $X' \subseteq X$ with $|x| = k$. (Let for example $X' = \{x_1, x_2\}$, then $F_{X'} := \{f_{x_1x_2}, f_{\bar{x}_1x_2}, f_{x_1\bar{x}_2}, f_{\bar{x}_1\bar{x}_2}\}$.) Then $f = 1$ holds, iff all cofactors in $F_{X'}$ are constantly 1, or, respectively, if they are all pairwise identical, and either of them is constantly one, as stated formally below.

$$
\begin{aligned}
& f = 1 \\
\Leftrightarrow\ & \forall f' \in F_{X'} : f' = 1 \\
\Leftrightarrow\ & \forall f', g' \in F_{X'} : f' = g' \ \wedge\ \exists h' \in F_{X'} : h' = 1
\end{aligned}
$$

Repeating this construction for all n variables leads to 2^n n-th order cofactors, where each of them is trivially identical to 0 or 1, since all variables have been removed. (This amounts to enumerating all possible assignments for the n variables.)

If all cofactors $f', g' \in F_{X'}$ are pairwise identical, then f is 1, iff some cofactor $f' = 1$. Conversely, if there are two cofactors which are not equivalent, it follows immediately that f is not constant. Hence $f = 1$ does not hold, as formally stated below.

$$
\begin{aligned}
(\forall g, h \in F_{X'} : g = h) \ &\Rightarrow\ (f = 1 \Leftrightarrow \exists f' \in F_{X'} : f' = 1) \\
(\exists g, h \in F_{X'} : g \neq h) \ &\Rightarrow\ (f \neq 1)
\end{aligned}
$$

The general idea to exploit this fact is the following. If, during the construction above, all cofactors are shown to be the same for some X', it is sufficient to subsequently consider (and to expand further) only one cofactor f'. In fact, this is possible for every X', since identity of Boolean functions can be decided either semantically or syntactically, as shown in Section 3. Unfortunately, the decision procedures provided there have exponential worst-case space or runtime requirements, which can make this approach prohibitively expensive. However, there are some syntactic criteria which imply the equivalence of terms. We will therefore first consider a syntactic characterization of the results above and then show how simple syntactic criteria can be applied.

(Note, the expansion above is only comparable with the Shannon expansion, in that it uses cofactors to split on. However, for some function it only preserves the 'core' of its codomain (i.e. the ability to produce the same values), but it is not concerned with representing f by cofactors.)

5.3 Equivalence of Cofactor Terms

Let $f : \mathbb{B} \times \cdots \times \mathbb{B} \to \mathbb{B}$ be an n-ary Boolean function, represented by some Boolean term $t \in T_{\mathbb{B}}(X)$ in n Boolean variables X. Then the positive and negative cofactors of f

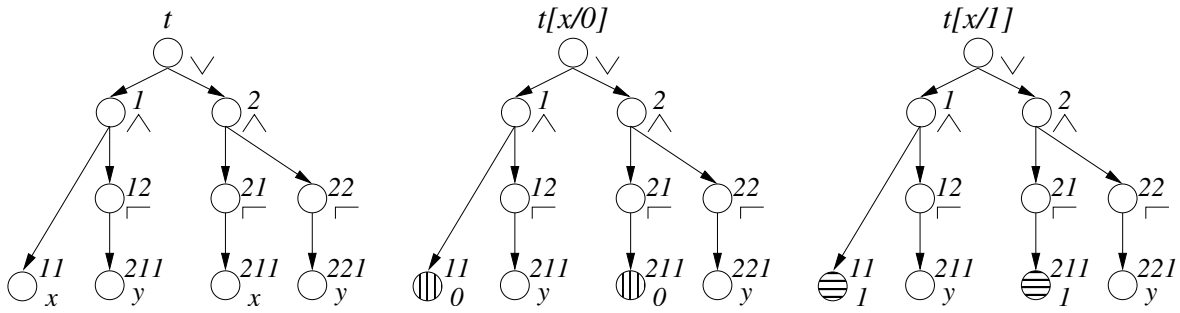


Figure 5.1: Graphical Representation of a Boolean Term and its Cofactors w.r.t. x

w.r.t. some variable $x \in X$, f_x and $f_{\bar{x}}$ are represented by the Boolean terms $t[x/1]$ and $t[x/0]$, where x is replaced with the constant symbols 1 and 0, respectively. The question whether $f = 1$ holds, is then equivalent to $\mathcal{T}_{\mathbf{B}}(X) \models t = 1$, and $f_x = f_{\bar{x}}$ is equivalent to $\mathcal{T}_{\mathbf{B}}(X) \models t[x/1] = t[x/0]$.

Example 5.4
Let the function $g : \mathbb{B} \times \mathbb{B} \to \mathbb{B}$ from Example 5.1 be represented by the Boolean term $t \equiv (x \wedge \neg y) \vee (\neg x \wedge \neg y)$ over the variables x and y. Then the cofactors g_x and $g_{\bar{x}}$ are represented by the terms $t[x/1] \equiv (1 \wedge \neg y) \vee (\neg 1 \wedge \neg y)$ and $t[x/0] \equiv (0 \wedge \neg y) \vee (\neg 0 \wedge \neg y)$, respectively. The term t, and the cofactor terms $t[x/1]$ and $t[x/0]$, are shown graphically in Fig. 5.1.

Let the set of cofactor terms of t w.r.t. a set of variables $X' \subseteq X$ be denoted by $T_{X'}$. The cofactor expansion described before is then equivalent to:

$$\begin{array}{ll} & \mathcal{T}_{\mathbf{B}}(X) \models t = 1 \\ \Leftrightarrow & \forall t' \in T_{X'} : \mathcal{T}_{\mathbf{B}}(X) \models t' = 1 \\ \Leftrightarrow & (\forall s', t' \in T_{X'} : \mathcal{T}_{\mathbf{B}}(X) \models s' = t') \ \wedge \ (\exists t' \in T_{X'} : \mathcal{T}_{\mathbf{B}}(X) \models t' = 1) \end{array}$$

Again it follows immediately that:

$$\begin{array}{lll} (\forall s', t' \in T_{X'} : \mathcal{T}_{\mathbf{B}}(X) \models s' = t') & \Rightarrow & (\mathcal{T}_{\mathbf{B}}(X) \models t = 1 \Leftrightarrow \exists t' \in T_{X'} : \mathcal{T}_{\mathbf{B}}(X) \models t' = 1) \\ (\exists s', t' \in T_{X'} : \mathcal{T}_{\mathbf{B}}(X) \not\models s' = t') & \Rightarrow & (\mathcal{T}_{\mathbf{B}}(X) \not\models t = 1) \end{array}$$

The idea is now to employ a semi-decision procedure for the Boolean equivalence problem during cofactor expansion, in order to (semi) decide whether they are equivalent, and thus to reduce their number. A semi-decision procedure for the Boolean equivalence problem $\mathcal{T}_{\mathbf{B}}(X) \models s = t$ is a procedure '`Boolean equiv(Term s, Term t)`', for which '`equiv(Term s, Term t)= true`' implies $\mathcal{T}_{\mathbf{B}}(X) \models s = t$, but not vice versa. Then from the equations above it follows:

$$(\forall s', t' \in T_{X'} : \texttt{equiv}(s', t')\texttt{=true}) \Rightarrow (\mathcal{T}_{\mathbf{B}}(X) \models t = 1 \Leftrightarrow \exists t' \in T_{X'} : \mathcal{T}_{\mathbf{B}}(X) \models t' = 1)$$

Example 5.5 (Semi-Decision of Boolean Equivalence)
Consider the examples for semi-decision procedures:

1. If the cofactor terms $t[x/0]$ and $t[x/1]$ of t are syntactically identical, then they are obviously equivalent, i.e. $t[x/0] \equiv t[x/1]$ implies $\mathcal{T}_{\mathbf{B}}(X) \models t[x/1] = t[x/0]$. However, this criterion is always too weak, as $t[x/1] \equiv t[x/0]$ can never be identical, because they differ in the positions of x. Let for example t, $t[x/1]$ and $t[x/0]$ be as in Example 5.4. Then $t[x/0]$ and $t[x/1]$ differ in the positions 11 and 211, as shown graphically in Fig. 5.1. Hence, we have to employ some of the semantics of $\mathcal{T}_{\mathbf{B}}(X)$, as shown below.

2. Let $\mathcal{T}_{\mathbf{B}}(X) \models E$ for some set of identities E. Then $E \vdash t[x/1] = t[x/0]$ implies $\mathcal{T}_{\mathbf{B}}(X) \models t[x/1] = t[x/0]$. Let for example E_n and R_n be as in Fig. 3.2, Pg. 36. Then R_n is a terminating and confluent rewrite system, deciding whether $E_n \vdash t[x/1] = t[x/0]$. Since E_n is derived from a subset of Boolean identities $E_{\mathbf{B}}$, we have $\mathcal{T}_{\mathbf{B}}(X) \models E_n$, Thus the TRS R_n (Tab. 3.2, Pg. 36) defines a semi-decision procedure for the Boolean equivalence problem. For example we obtain $t[x/1] \downarrow_{R_n} \equiv \neg y \equiv t[x/0] \downarrow_{R_n}$ for $t[x/0]$ and $t[x/1]$ from above.

If the number of generated cofactor terms is limited by some sensible bound, an exponential blowup can be avoided as follows. If all cofactor terms can be shown to be equivalent, their number can be reduced to one. Otherwise, they are expanded. If their number reaches the given bound, the expansion stops, and a full-decision procedure is employed.

The approach leads to Algorithm 1 '`taut-cf`'. Let '`all-equivalent`' be a semi-decision procedure for the Boolean equivalence problem of a set of Boolean terms `T`, returning `true` if all `s` and `t` in `T` are shown to be pairwise equivalent, and `false` otherwise. Furthermore, let '`taut(t)`' be a full decision procedure returning `true` iff for a Boolean term `t` $\mathcal{T}_{\mathbf{B}}(X) \models \mathtt{t} = 1$. Further, let $\mathtt{k} \leq n$ be some natural. Algorithm 1 uses equivalent values to decide whether $\mathcal{T}_{\mathbf{B}}(X) \models \mathtt{t} = 1$. For `k = 0` the algorithm is equivalent to directly calling the decision procedure `taut`. Otherwise, it tries to prove the equivalence of some cofactor terms until it reaches the maximum number of iterations `k`. Then the number of cofactor terms in `T` is bound by $2^{\mathtt{k}}$. Suppose that `all-equivalent(T)` is relatively cheap. (Obviously `taut(t)` must be expensive (unless Co-NP=P).) Then this procedure can be used as preprocessor for tautology checking of (first-order) Boolean formulas. The approach and its limitations are illustrated in the following example.

Example 5.6 (Property Checking for a Register File)
Consider the following hardware design:

$$
\begin{array}{rrll}
D = (& \overline{\imath} = & (w, a, b, i), & \text{input variables} \\
 & \overline{o} = & (o), & \text{output variables} \\
 & \overline{s} = & (s, t, u, v), & \text{state variables} \\
f_{S_0} : \mathbb{B}^4 \to \mathbb{B}, & f_{S_0}(\overline{s}) = & 1, & \text{initial states function} \\
f_O : \mathbb{B}^8 \to \mathbb{B}^4, & f_O(\overline{\imath}, \overline{s}) = & (\mathrm{ite}(b, \mathrm{ite}(a, v, u), \mathrm{ite}(a, t, s))), & \text{output function} \\
f_S : \mathbb{B}^8 \to \mathbb{B}^4, & f_S(\overline{\imath}, \overline{s}) = & (\mathrm{ite}(w \wedge \neg a \wedge \neg b, i, s), & \text{transition function} \\
 & & \mathrm{ite}(w \wedge a \wedge \neg b, i, t), & \\
 & & \mathrm{ite}(w \wedge \neg a \wedge b, i, u), & \\
 & & \mathrm{ite}(w \wedge a \wedge b, i, v)) \ ) &
\end{array}
$$

It describes a simple register file with 4 Boolean entries s, t, u, v, which are addressed by a combination of the inputs a and b. If $w = 1$, then the input i is stored as register entry

Algorithm 1 Tautology Checking with Equivalent Value Vectors

```
Boolean taut-cf(Term t, Natural k)
{
  Natural i := 0;                     // Current number of iterations.
  Set<Term> T := { t };               // Cofactor terms considered.
  foreach (Variable x in Var(t)) {    // For each variable x in t:
    if (i >= k)                       // If the max number of
      break;                          // iterations is not reached:
    i++;                              // Increment iteration count.
    Set<Term> T';                     // Set of cofactors w.r.t. x.
    foreach Term t' in T {            // Foreach t' in T:
      T'.insert(substitute(t',x,0)); // Compute both cofactors
      T'.insert(substitute(t',x,1)); // and store them in T'.
    }
    if (all-equivalent(T',k)) {       // If all cofactors are
      T := { T'.choose() };           // equivalent choose one
      i := 0;                         // and reset iteration count.
    } else {                          // Otherwise consider T'
      T:= T';                         // in the following.
    }
  }                                   // Stop cofactor term generation.
  foreach t' in T' {                  // Consider all remaining
   if (! taut(t'))                    // cofactor terms. If one is
     return false;                    // false then t=1 doesn't hold.
  }                                   // Otherwise it does.
  return true;
}
```

corresponding to a and b. If $w = 0$, then all entries remain unchanged. The output o always shows the current value of the register addressed by a and b.

To verify one aspect of its behavior we formulate the following property p, stating that $w = 1$ at $\mathtt{t}$ results in the register entry indexed by a and b changed to i at $\mathtt{t}$, at the following time point $\mathtt{t}+1$.

$$p \equiv \begin{array}{lll} & (w_\mathtt{t} \wedge \neg a_\mathtt{t} \wedge \neg b_\mathtt{t} & \Rightarrow s_{\mathtt{t}+1} = i_\mathtt{t}) \\ \wedge & (w_\mathtt{t} \wedge a_\mathtt{t} \wedge \neg b_\mathtt{t} & \Rightarrow t_{\mathtt{t}+1} = i_\mathtt{t}) \\ \wedge & (w_\mathtt{t} \wedge \neg a_\mathtt{t} \wedge b_\mathtt{t} & \Rightarrow u_{\mathtt{t}+1} = i_\mathtt{t}) \\ \wedge & (w_\mathtt{t} \wedge a_\mathtt{t} \wedge b_\mathtt{t} & \Rightarrow v_{\mathtt{t}+1} = i_\mathtt{t}) \end{array}$$

The Boolean formula p' to prove that this property holds is obtained as usual:

$$p' \equiv \begin{array}{lll} & (w_\mathtt{t} \wedge \neg a_\mathtt{t} \wedge \neg b_\mathtt{t} & \Rightarrow \mathrm{ite}(w_\mathtt{t} \wedge \neg a_\mathtt{t} \wedge \neg b_\mathtt{t}, i_\mathtt{t}, s_\mathtt{t}) = i_\mathtt{t}) \\ \wedge & (w_\mathtt{t} \wedge a_\mathtt{t} \wedge \neg b_\mathtt{t} & \Rightarrow \mathrm{ite}(w_\mathtt{t} \wedge a_\mathtt{t} \wedge \neg b_\mathtt{t}, i_\mathtt{t}, t_\mathtt{t}) = i_\mathtt{t}) \\ \wedge & (w_\mathtt{t} \wedge \neg a_\mathtt{t} \wedge b_\mathtt{t} & \Rightarrow \mathrm{ite}(w_\mathtt{t} \wedge \neg a_\mathtt{t} \wedge b_\mathtt{t}, i_\mathtt{t}, u_\mathtt{t}) = i_\mathtt{t}) \\ \wedge & (w_\mathtt{t} \wedge a_\mathtt{t} \wedge b_\mathtt{t} & \Rightarrow \mathrm{ite}(w_\mathtt{t} \wedge a_\mathtt{t} \wedge b_\mathtt{t}, i_\mathtt{t}, v_\mathtt{t}) = i_\mathtt{t}) \end{array}$$

The positive and negative cofactor terms of p' w.r.t. to $a_\mathtt{t}$ are then $p'[a_\mathtt{t}/1]$ and $p'[a_\mathtt{t}/0]$, s.t.:

$$p'[a_\mathtt{t}/1] \equiv \begin{array}{lll} & (w_\mathtt{t} \wedge \neg 1 \wedge \neg b_\mathtt{t} & \Rightarrow \mathrm{ite}(w_\mathtt{t} \wedge \neg 1 \wedge \neg b_\mathtt{t}, i_\mathtt{t}, s_\mathtt{t}) = i_\mathtt{t}) \\ \wedge & (w_\mathtt{t} \wedge 1 \wedge \neg b_\mathtt{t} & \Rightarrow \mathrm{ite}(w_\mathtt{t} \wedge 1 \wedge \neg b_\mathtt{t}, i_\mathtt{t}, t_\mathtt{t}) = i_\mathtt{t}) \\ \wedge & (w_\mathtt{t} \wedge \neg 1 \wedge b_\mathtt{t} & \Rightarrow \mathrm{ite}(w_\mathtt{t} \wedge \neg 1 \wedge b_\mathtt{t}, i_\mathtt{t}, u_\mathtt{t}) = i_\mathtt{t}) \\ \wedge & (w_\mathtt{t} \wedge 1 \wedge b_\mathtt{t} & \Rightarrow \mathrm{ite}(w_\mathtt{t} \wedge 1 \wedge b_\mathtt{t}, i_\mathtt{t}, v_\mathtt{t}) = i_\mathtt{t}) \end{array}$$

$$p'[a_\mathtt{t}/0] \equiv \begin{array}{lll} & (w_\mathtt{t} \wedge \neg 0 \wedge \neg b_\mathtt{t} & \Rightarrow \mathrm{ite}(w_\mathtt{t} \wedge \neg 0 \wedge \neg b_\mathtt{t}, i_\mathtt{t}, s_\mathtt{t}) = i_\mathtt{t}) \\ \wedge & (w_\mathtt{t} \wedge 0 \wedge \neg b_\mathtt{t} & \Rightarrow \mathrm{ite}(w_\mathtt{t} \wedge 0 \wedge \neg b_\mathtt{t}, i_\mathtt{t}, t_\mathtt{t}) = i_\mathtt{t}) \\ \wedge & (w_\mathtt{t} \wedge \neg 0 \wedge b_\mathtt{t} & \Rightarrow \mathrm{ite}(w_\mathtt{t} \wedge \neg 0 \wedge b_\mathtt{t}, i_\mathtt{t}, u_\mathtt{t}) = i_\mathtt{t}) \\ \wedge & (w_\mathtt{t} \wedge 0 \wedge b_\mathtt{t} & \Rightarrow \mathrm{ite}(w_\mathtt{t} \wedge 0 \wedge b_\mathtt{t}, i_\mathtt{t}, v_\mathtt{t}) = i_\mathtt{t}) \end{array}$$

Rewriting the cofactor terms $p'[a_\mathtt{t}/1]$ and $p'[a_\mathtt{t}/0]$ using R_n (Tab. 3.2, Pg. 36) yields $p''_{a_\mathtt{t}} \equiv p'[a_\mathtt{t}/1] \downarrow_{R_n}$ and $p''_{\bar{a}_\mathtt{t}} \equiv p'[a_\mathtt{t}/0] \downarrow_{R_n}$, as below:

$$p''_{\bar{a}_\mathtt{t}} \equiv \begin{array}{lll} & (w_\mathtt{t} \wedge \neg b_\mathtt{t} & \Rightarrow \mathrm{ite}(w_\mathtt{t} \wedge \neg b_\mathtt{t}, i_\mathtt{t}, s_\mathtt{t}) = i_\mathtt{t}) \\ \wedge & (w_\mathtt{t} \wedge b_\mathtt{t} & \Rightarrow \mathrm{ite}(w_\mathtt{t} \wedge b_\mathtt{t}, i_\mathtt{t}, u_\mathtt{t}) = i_\mathtt{t}) \end{array}$$

$$p''_{a_\mathtt{t}} \equiv \begin{array}{lll} & (w_\mathtt{t} \wedge \neg b_\mathtt{t} & \Rightarrow \mathrm{ite}(w_\mathtt{t} \wedge \neg b_\mathtt{t}, i_\mathtt{t}, t_\mathtt{t}) = i_\mathtt{t}) \\ \wedge & (w_\mathtt{t} \wedge b_\mathtt{t} & \Rightarrow \mathrm{ite}(w_\mathtt{t} \wedge b_\mathtt{t}, i_\mathtt{t}, v_\mathtt{t}) = i_\mathtt{t}) \end{array}$$

Note, that the terms $p''_{a_\mathtt{t}}$ and $p''_{\bar{a}_\mathtt{t}}$ are almost identical. Repeating the process, computing the cofactor terms of $p''_{a_\mathtt{t}}$ and $p''_{\bar{a}_\mathtt{t}}$ w.r.t. $b_\mathtt{t}$, and rewriting them with R_n, yields $p''_{a_\mathtt{t} b_\mathtt{t}}$, $p''_{a_\mathtt{t} \bar{b}_\mathtt{t}}$, $p''_{\bar{a}_\mathtt{t} b_\mathtt{t}}$, and $p''_{\bar{a}_\mathtt{t} \bar{b}_\mathtt{t}}$, as below:

$$\begin{array}{ll} p''_{\bar{a}_\mathtt{t} \bar{b}_\mathtt{t}} \equiv & (w_\mathtt{t} \Rightarrow \mathrm{ite}(w_\mathtt{t}, i_\mathtt{t}, s_\mathtt{t}) = i_\mathtt{t}) \\ p''_{a_\mathtt{t} \bar{b}_\mathtt{t}} \equiv & (w_\mathtt{t} \Rightarrow \mathrm{ite}(w_\mathtt{t}, i_\mathtt{t}, t_\mathtt{t}) = i_\mathtt{t}) \\ p''_{\bar{a}_\mathtt{t} b_\mathtt{t}} \equiv & (w_\mathtt{t} \Rightarrow \mathrm{ite}(w_\mathtt{t}, i_\mathtt{t}, u_\mathtt{t}) = i_\mathtt{t}) \\ p''_{a_\mathtt{t} b_\mathtt{t}} \equiv & (w_\mathtt{t} \Rightarrow \mathrm{ite}(w_\mathtt{t}, i_\mathtt{t}, v_\mathtt{t}) = i_\mathtt{t}) \end{array}$$

Again, the cofactor terms are almost identical, but to decide whether $p' = 1$ holds, the expansion continues with $k = 2$, and we have to check each of the 4 cofactor terms for constantness. Although the terms above look very similar, the method, developed so far, is not able to take advantage of this situation, which illustrates its limitations.

Note that the cofactor terms of each iteration in the example above look very similar. They only differ in variables names. It is indeed possible to reason that it is sufficient to consider just one of them further (for each iteration) by proving that they are permutation equivalent, i.e. symmetric, as shown in the next section.

Further note, that algorithm 1 could be improved by some backtracking, taking back the cofactor term generation for some variables and choosing others instead. Furthermore, `all-equivalent` could be changed such that it returns an equivalence relation on the set of cofactor terms `T` in case that they are not all pairwise equivalent. Then only one representative of each class would have to be considered. These ideas are used and treated in more detail for bitvector terms in Chapter 10.

5.4 Symmetrical Values

Let $f, g \in \mathbb{B}^{\mathbb{B}^n}$ be an n-ary Boolean functions over variables $X = \{x_1, \ldots, x_n\}$. The Boolean functions f, g are *symmetrical* (permutation equivalent) iff there is a permutation[2] $\pi \in Sym(X)$ of the variables X such that $f = \pi(g)$, i.e. for all $x_1, \ldots, x_n \in \mathbb{B}$, $f(x_1, \ldots, x_n) = g(\pi(x_1), \ldots, \pi(x_n))$. Such a permutation is called a *variable renaming*.

Example 5.7 (Symmetrical Functions)
As an example consider the functions f and g:

$$\begin{array}{rcl} f : \{0,1\} \times \{0,1\} & \rightarrow & \{0,1\} \\ f(x,y) & \mapsto & (x \wedge \neg y) \vee (\neg x \wedge \neg y) \\ g : \{0,1\} \times \{0,1\} & \rightarrow & \{0,1\} \\ g(x,y) & \mapsto & (y \wedge \neg x) \vee (\neg y \wedge \neg x) \end{array}$$

They are identical after renaming x with y, and y with x in either function, i.e. for $\pi = (x \mapsto y, y \mapsto x)$ we have $f(x,y) = g(\pi(x), \pi(y))$. Note, that the functions f' and g' below are also identical under the same π.

$$\begin{array}{rcl} f' : \{0,1\} \times \{0,1\} & \rightarrow & \{0,1\} \\ f'(x,y) & \mapsto & \neg y \\ g' : \{0,1\} \times \{0,1\} & \rightarrow & \{0,1\} \\ g'(x,y) & \mapsto & \neg x \end{array}$$

The functions f and f' are actually the same. Therefore they are also symmetrical. (The same holds for g and g'.)

If a function is constant, renaming variables does not change this fact. For Boolean functions we have in particular $f = 1$ iff $\pi(f) = 1$. Now consider the cofactors f_x, and $f_{\bar{x}}$ of a Boolean function f w.r.t. some variable x. We have already seen that $f = 1$ iff $f_x = 1$, and $f_{\bar{x}} = 1$. Then $f = 1$ iff for some variable renaming π, $f_x = 1$, and $\pi(f_{\bar{x}}) = 1$. Finally we conclude, $f = 1$ iff $f_x = 1$, and there exists some π such that $f_x = \pi(f_{\bar{x}})$. If there is such a variable renaming π we call 0 and 1 *symmetrical values for x in f*. (Then $(x \mapsto 0)$ and $(x \mapsto 1)$ are symmetrical partial assignments.) In general, i.e. for more than one variable, we define symmetrical value vectors as follows.

[2]The set of all permutations of X, as well as the symmetric group on X, are denoted as $Sym(X)$.

Definition 5.8 (Symmetrical Value Vectors)
Let $f : \mathbb{B}^n \to \mathbb{B}$ be a Boolean function in n variables X. Let $X_k \subseteq X$ with $X_k = \{x_1, \ldots, x_k\}$. The vectors $(a_1, \ldots, a_k), (b_1, \ldots, b_k) \in \mathbb{B}^k$ are called *symmetrical value vectors* for the vector of variables $(x_1, \ldots, x_k)$ w.r.t. f, iff

$$\exists \pi \in Sym(X \backslash X_k) : f|_{x_1=a_1,\ldots,x_k=a_k} = \pi(f|_{x_1=b_1,\ldots,x_k=b_k})$$

Then, by the definition above, $\varphi, \varphi' : X^k \to \mathbb{B}$ are *symmetrical assignments* for X_k w.r.t. f if $\varphi(x_i) = a_i$ and $\varphi'(x_i) = b_i$. Symmetrical value vectors for variables in some function, if known, can be used for preprocessing in the same way as equivalent value vectors, when deciding the Boolean equivalence problem.

As before, let $F_{X'}$ be the set of all k-th order cofactors of f w.r.t a set $X' \subseteq X$ with $|X| = k$. Then $f = 1$ holds, iff all cofactors in $F_{X'}$ are constantly 1, or, respectively, if they are all pairwise symmetrical, and either of them is constantly one.

$$\begin{aligned} & f = 1 \\ \Leftrightarrow \quad & \forall f', g' \in F_{X'} \exists \pi \in Sym(X \backslash X') : f' = \pi(g') \ \wedge \ \exists h' \in F_{X'} : h' = 1 \end{aligned}$$

The problem of finding π such that for two functions f and g, $f = \pi(g)$ holds is known as the permutation equivalence problem. For Boolean functions, it is also known as Boolean isomorphism problem [AT96].

5.5 Permutation Equivalence of Cofactor Terms

In this section, the concept of symmetrical values, and its application to reduction, is translated from Boolean functions to Boolean terms representing them. Consider the cofactor expansion of a Boolean term t w.r.t. a variable x, occurring in t again. Then we have:

$$\begin{aligned} & \mathcal{T}_{\mathbf{B}}(X) \models t = 1 \\ \Leftrightarrow \quad & \mathcal{T}_{\mathbf{B}}(X) \models t[x/1] = 1 \ \wedge \ \mathcal{T}_{\mathbf{B}}(X) \models t[x/0] = 1 \\ \Leftrightarrow \quad & \mathcal{T}_{\mathbf{B}}(X) \models t[x/1] = t[x/0] \ \wedge \ \mathcal{T}_{\mathbf{B}}(X) \models t[x/1] = 1 \end{aligned}$$

Obviously, renaming variables in a term does not change the represented function. We will show that the Boolean equivalence problem for cofactor terms $t[x/1]$, and $t[x/0]$ above, can be relaxed to so called Boolean isomorphism problem, i.e. to the permutation equivalence problem for Boolean terms modulo Boolean axioms.

Two Boolean terms $s, t \in T_{\mathbb{B}}(X)$ are permutation equivalent, if there is a variable renaming, a bijection on X, such that they are equivalent under this renaming, i.e. iff $\exists \pi \in Sym(X) : \mathcal{T}_{\mathbf{B}}(X) \models s = t\pi$. (Note that since we usually require X to be finite, a bijection on X is a permutation of X, hence the name permutation equivalence. Furthermore, π is a substitution, naturally extending to a homomorphism on terms. Finally, it is sufficient to consider permutations of $Var(s) \cup Var(t) \subseteq X$.)

Let $\pi_0, \pi_1 \in Sym(X \backslash \{x\})$ be some variable permutations. Let $\pi := \pi_1^{-1}\pi_0$, then

$$\begin{array}{ll} & \mathcal{T}_{\mathbf{B}}(X) \models t = 1 \\ \Leftrightarrow & \mathcal{T}_{\mathbf{B}}(X) \models t[x/1] = 1 \ \wedge \ \mathcal{T}_{\mathbf{B}}(X) \models t[x/0] = 1 \\ \Leftrightarrow & \mathcal{T}_{\mathbf{B}}(X) \models t[x/1]\pi_1 = 1 \ \wedge \ \mathcal{T}_{\mathbf{B}}(X) \models t[x/0]\pi_0 = 1 \\ \Leftrightarrow & \mathcal{T}_{\mathbf{B}}(X) \models t[x/1]\pi_1 = t[x/0]\pi_0 \ \wedge \ \mathcal{T}_{\mathbf{B}}(X) \models t[x/0]\pi_0 = 1 \\ \Leftrightarrow & \mathcal{T}_{\mathbf{B}}(X) \models t[x/1] = t[x/0]\pi_1^{-1}\pi_0 \ \wedge \ \mathcal{T}_{\mathbf{B}}(X) \models t[x/0] = 1 \\ \Leftrightarrow & \mathcal{T}_{\mathbf{B}}(X) \models t[x/1] = t[x/0]\pi \ \wedge \ \mathcal{T}_{\mathbf{B}}(X) \models t[x/0] = 1 \end{array}$$

It follows immediately that for a Boolean term t, $\mathcal{T}_{\mathbf{B}}(X) \models t = 1$, if the cofactor terms $t[x/0]$ and $t[x/1]$ are permutation equivalent, and either of which is constantly true, which is formally stated below.

$$\begin{array}{ll} & (\exists \pi \in Sym(X \backslash \{x\}) : \mathcal{T}_{\mathbf{B}}(X) \models t[x/1] = t[x/0]\pi) \\ \Rightarrow & (\mathcal{T}_{\mathbf{B}}(X) \models t = 1 \Leftrightarrow \mathcal{T}_{\mathbf{B}}(X) \models t[x/0] = 1) \end{array}$$

Considering a set of identities E, such that $\mathcal{T}_{\mathbf{B}}(X) \models E$, leads to the following result:

Theorem 5.9
Let E be a set of identities, such that $\mathcal{T}_{\mathbf{B}}(X) \models E$. Let $t \in T_{\mathbb{B}}(X)$ be a Boolean term, and let $x \in X$ be a variable, then:

$$\begin{array}{ll} & (\exists \pi \in Sym(X \backslash \{x\}) : E \vdash t[x/0] = t[x/1]\pi) \\ \Rightarrow & (\mathcal{T}_{\mathbf{B}}(X) \models t = 1 \Leftrightarrow \mathcal{T}_{\mathbf{B}}(X) \models t[x/1] = 1) \end{array}$$

Proof: Since $\mathcal{T}_{\mathbf{B}}(X) \models E$, $E \vdash t[x/0] = t[x/1]\pi$ implies $\mathcal{T}_{\mathbf{B}}(X) \models t[x/0] = t[x/1]\pi$ which implies $(\mathcal{T}_{\mathbf{B}}(X) \models t = 1 \Leftrightarrow \mathcal{T}_{\mathbf{B}}(X) \models t[x/1] = 1)$. ■

Corollary 5.10
Let E be a set of identities, such that $\mathcal{T}_{\mathbf{B}}(X) \models E$. Let $t \in T_{\mathbb{B}}(X)$ be a Boolean term, and let $x \in X$ be a variable, then:

$$\begin{array}{ll} & (E \vdash s_0 = t[x/0] \ \wedge \ E \vdash s_1 = t[x/1] \ \wedge \ \exists \pi \in Sym(X \backslash \{x\}) : s_0 \equiv s_1\pi) \\ \Rightarrow & (\mathcal{T}_{\mathbf{B}}(X) \models t = 1 \Leftrightarrow \mathcal{T}_{\mathbf{B}}(X) \models t[x/1] = 1) \end{array}$$

This corollary states that it is possible to solve two problems independently, first finding s_0 and s_1, which are equivalent to $t[x/0]$ and $t[x/1]$ modulo some set of identities E, and second, finding a variable permutation π, such that $s_0 \equiv s_1\pi$. Then $\mathcal{T}_{\mathbf{B}}(X) \models t = 1$ follows from $\mathcal{T}_{\mathbf{B}}(X) \models t[x/0] = 1$.

Let for example R be a convergent rewrite system derived from E, let $s_0 \equiv t[x/0] \downarrow_R$ and $s_1 \equiv t[x/1] \downarrow_R$. Then the remaining problem of finding π, can then solved using isomorphisms of directed acyclic graphs, as shown later (in more generality) in the context permutation equivalence of bitvector terms.

Finally, the results above can be extended to sets of cofactor terms, as below.

Corollary 5.11
Let $X' \subseteq X$ be a set of Boolean variables, and let $T_{X'}$ be the set of cofactor terms of a Boolean term t over X w.r.t. X'. Let E be a set of identities, such that $\mathcal{T}_{\mathbf{B}}(X) \models E$, then:

$$\begin{array}{ll} & (\forall s', t' \in T_{X'} \exists \pi \in Sym(X \backslash X') : E \vdash s' = t'\pi) \\ \Rightarrow & (\mathcal{T}_{\mathbf{B}}(X) \models t = 1 \ \Leftrightarrow \ \exists t' \in T_{X'} : \mathcal{T}_{\mathbf{B}}(X) \models t' = 1) \end{array}$$

Let '`all-symmetrical`' be an algorithm returning `true` if all `s` and `t` of a set of terms `T` pairwise permutation equivalent and `false` otherwise. (It is a semi-decision procedure for the permutational equivalence problem for `T`.) Then '`all-symmetrical`' can be used instead of '`all-equivalent`' in Algorithm 1. The latter remains complete, and is more powerful than before, as shown in the example below.

Example 5.12 (Property Checking for a Register File (continued))
Consider the normalized cofactor terms $p''_{a_\mathtt{t}}$ and $p''_{\bar{a}_\mathtt{t}}$ of p' of the first iteration in Example 5.6 again.

$$\begin{array}{rcll} p''_{\bar{a}_\mathtt{t}} \equiv & & (w_\mathtt{t} \wedge \neg b_\mathtt{t} & \Rightarrow \mathrm{ite}(w_\mathtt{t} \wedge \neg b_\mathtt{t}, i_\mathtt{t}, s_\mathtt{t}) = i_\mathtt{t}) \\ & \wedge & (w_\mathtt{t} \wedge b_\mathtt{t} & \Rightarrow \mathrm{ite}(w_\mathtt{t} \wedge b_\mathtt{t}, i_\mathtt{t}, u_\mathtt{t}) = i_\mathtt{t}) \\ p''_{a_\mathtt{t}} \equiv & & (w_\mathtt{t} \wedge \neg b_\mathtt{t} & \Rightarrow \mathrm{ite}(w_\mathtt{t} \wedge \neg b_\mathtt{t}, i_\mathtt{t}, t_\mathtt{t}) = i_\mathtt{t}) \\ & \wedge & (w_\mathtt{t} \wedge b_\mathtt{t} & \Rightarrow \mathrm{ite}(w_\mathtt{t} \wedge b_\mathtt{t}, i_\mathtt{t}, v_\mathtt{t}) = i_\mathtt{t}) \end{array}$$

The terms $p''_{a_\mathtt{t}}$ and $p''_{\bar{a}_\mathtt{t}}$ are identical under the variable permutation $\pi = (t_\mathtt{t}\ s_\mathtt{t})(v_\mathtt{t}\ u_\mathtt{t})$ (written in disjoint cycle notation), which implies that p' holds if $p''_{a_\mathtt{t}}$ holds. Hence, it is sufficient to subsequently consider (and further expand) only one of them, say $p''_{a_\mathtt{t}}$. Repeating the process, computing the cofactor terms of $p''_{a_\mathtt{t}}$ w.r.t. $b_\mathtt{t}$, and rewriting them with R_n, yields $p''_{a_\mathtt{t} b_\mathtt{t}}$ and $p''_{a_\mathtt{t} \bar{b}_\mathtt{t}}$ below:

$$\begin{array}{rl} p''_{a_\mathtt{t} b_\mathtt{t}} \equiv & (w_\mathtt{t} \Rightarrow \mathrm{ite}(w_\mathtt{t}, i_\mathtt{t}, v_\mathtt{t}) = i_\mathtt{t}) \\ p''_{a_\mathtt{t} \bar{b}_\mathtt{t}} \equiv & (w_\mathtt{t} \Rightarrow \mathrm{ite}(w_\mathtt{t}, i_\mathtt{t}, t_\mathtt{t}) = i_\mathtt{t}) \end{array}$$

Again, there is a permutation $\pi_2 = (v_\mathtt{t}\ t_\mathtt{t})$, such that $p''_{a_\mathtt{t} b_\mathtt{t}} \equiv p''_{a_\mathtt{t} \bar{b}_\mathtt{t}} \pi_2$, allowing to conclude that $p' = 1$ if $p''_{a_\mathtt{t} b_\mathtt{t}} = 1$, which is significantly easier to prove, as the variables $a_\mathtt{t}$, $b_\mathtt{t}$, $s_\mathtt{t}$, $t_\mathtt{t}$, and $u_\mathtt{t}$ have been removed.

5.6 Symmetrical Variables and Equivalent Values

The intuitive notion of symmetry in a function is that of symmetrical variables. An n-ary Boolean function f is symmetrical in some variables x and y, if f is invariant under permutation of these variables, i.e. f is invariant under $\pi = (x\ y)$ with $\pi(f) = f(\pi x_1, \ldots, \pi x_n) = f(x_1, \ldots, x_n)$. For example, the variables x and y in the function $f : \mathbb{B} \times \mathbb{B} \times \mathbb{B} \to \mathbb{B}$ with $f(x, y, z) \mapsto (y \wedge x) \vee z$ are symmetrical, since $f(x, y, z) = (x \wedge y) \vee z = (y \wedge x) \vee z = (\pi(x) \wedge \pi(y)) \vee \pi(z) = f(\pi(x), \pi(y), \pi(z)) = \pi(f(x, y, z))$. In general, a symmetry of f is an arbitrary permutation of its variables under which f is invariant. The relation between the classical notion of symmetry in variables above and equivalent value vectors, as introduced in this thesis, is described below.

If x_i, x_j are symmetrical variables for $f(x_1, \ldots, x_n)$ then $(0, 1)$ and and $(1, 0)$ are equivalent value vectors for (x_i, x_j) w.r.t. f (see Def. 5.2). The converse holds as well. However, if for example $(1, 1)$ and $(0, 0)$ are symmetrical value vectors for (x_i, x_j) w.r.t. f, then this fact can not be expressed by variable permutation. The situation becomes even more obvious considering sets of symmetries (variable permutations) vs. sets of equivalent value vector pairs, as in the following example.

Example 5.13
Let $f : \mathbb{B}^n \to \mathbb{B}$ with $n \geq 4$ be fully symmetrical in the variables x_1, x_2, x_3, x_4, i.e. for all 24 permutations $\pi \in Sym(\{x_1, \ldots, x_4\})$ we have $f = \pi(f)$. Consider the 16 possible ways of assigning $x_1, \ldots, x_4$ with Boolean values. The full variable symmetry above implies the following equivalence relation between the 16 value vectors for $(x_1, \ldots, x_4)$ (written as a partition P of $\mathbb{B}^4$).

$$
\begin{array}{rcl}
P & = & \{ \ \{(0,0,0,0)\}, \\
 & & \ \{(0,0,0,1),(0,0,1,0),(0,1,0,0),(1,0,0,0)\}, \\
 & & \ \{(0,0,1,1),(0,1,1,0),(1,1,0,0),(0,1,0,1),(1,0,1,0),(1,0,0,1)\}, \\
 & & \ \{(1,1,1,0),(1,1,0,1),(1,0,1,1),(0,1,1,1)\}, \\
 & & \ \{(1,1,1,1)\} \qquad \}
\end{array}
$$

The number of occurrences of 1s and 0s in the value vectors is decisive for the resulting equivalence relation. Again, if for example $(1,1,1,0)$ and $(0,0,0,1)$ are equivalent value vectors, this fact can not be expressed by variable permutation.

Equivalent value vectors provide a more concise way of expressing symmetry. However, results for symmetry of variables, such as techniques for detecting symmetry, may be applicable to equivalent value vectors and the intended reduction. Conversely, techniques described here for equivalent value vectors are useful for symmetry in variables as well.

Symmetry of variables has been studied extensively. The application of symmetry reduction by symmetry breaking predicates, along the lines of [CGLR96]), has been limited to only a few practical cases. Many of the results reported show reduction of artificial examples such as the pigeon hole problem, which rarely occur in real life hardware designs.

In contrast, the application of equivalent and symmetrical value vectors for preprocessing can speed up verification significantly in relevant industrial cases. Finding and exploiting equivalent and symmetrical value vectors for BMC is best done on the register transfer level, minimizing the effort needed later on the Boolean level. (Equivalent and symmetrical value vectors correspond to equivalent and symmetrical bitvector values.) The approach is described in detail in the following, second part, of this thesis.

Part II

Register Transfer Level Methods

Chapter 6

Algebraic Framework for Bitvector Functions

On the register transfer level, the content of registers and the I/O values of hardware designs are bitvectors. Transfer of data between registers of different sizes can be described by bitvector functions. Bitvector functions can be handled conveniently in a many-sorted algebras, which provide the algebraic framework for their representation and related decision procedures.

6.1 Bitvector Functions

Bitvectors are, as the name suggests, finite width vectors of bits. For example (0), $(0, 1, 0, 0, 1)$ and $(0, 0, 0, 0, 0, 0, 0)$ are bitvectors. Bitvectors are characterized by their width. The *set of bitvectors of width* n is $\mathbb{B}^n$. The bit elements of a bitvector are written from right to left, and are numbered by their position, starting with 0.

Example 6.1
Let $(1, 0, 1, 0, 1, 0, 1, 0)$ be a bitvector. Its width is 8. The bit at position 0 is the rightmost bit, the bit at position 7 is the leftmost bit.

Bitvectors are denoted symbolically by letters in bold face, and their width is indicated in square brackets as subscript. For example $\mathbf{a}_{[4]}$ denotes a bitvector named $\mathbf{a}$ of width 4. The width may be omitted if clear from the context or irrelevant. A bitvector $\mathbf{a}_{[n]}$ equals the vector of its components, i.e. as $\mathbf{a}_{[n]} = (a_{n-1}, \ldots, a_0)$.

Often, bitvectors are identified with natural numbers in binary representation. If $(a_{n-1}, \ldots, a_0)$ is an n-bit vector, it is identified with $\sum_{i=0}^{n-1} 2^i a_i$. For example the bitvector $(1, 0, 1, 0, 1, 0, 1, 0)$ is identified with the natural 170. Conversely, given a width n, a natural number a can be identified with a bitvector $(a_{n-1}, \ldots, a_0)$ of width n such that for $(a_{n-1}, \ldots, a_0)$, $\sum_{i=0}^{n-1} 2^i a_i \equiv a \bmod 2^n$. Naturals that are interpreted as bitvectors are also written in bold face. For example $\mathbf{170}_{[8]}$ denotes $(1, 0, 1, 0, 1, 0, 1, 0)$.[1]

[1] Other interpretations of bitvectors include integer numbers (e.g. as two's complement), fixed point and floating point numbers.

Definition 6.2
A k-ary *bitvector function of width* n is a function taking arguments in $\mathbb{B}^{n_1}, \ldots \mathbb{B}^{n_k}$ and delivering values in $\mathbb{B}^n$, i.e.

$$f : \mathbb{B}^{n_1} \times \cdots \times \mathbb{B}^{n_k} \to \mathbb{B}^n$$

Examples 6.3 (Bitvector Functions)

1. Boolean operations:

   The Boolean operations $\{\wedge, \vee, \neg, 0, 1\}$ on $\mathbb{B}$ are bitvector functions, e.g.:

   $$\wedge : \mathbb{B}^1 \times \mathbb{B}^1 \to \mathbb{B}^1$$

2. Bit projection:

   Let $(a_5, \ldots, a_0) \in \mathbb{B}^6$ be a bitvector of width 6. Let $[4]_{[6]} : \mathbb{B}^6 \to \mathbb{B}$ be the 4-th bit projection of $\mathbb{B}^6$ onto $\mathbb{B}$. Then $(a_5, \ldots, a_0)[4]_{[6]}$ denotes the 4-th bit of $(a_5, \ldots, a_0)$, i.e. a_4. In general, the set of all bit projections from $\mathbb{B}^k$ onto $\mathbb{B}$ is the set of unary bitvector functions of width 1, such that for $i \in \{0, \ldots, k-1\}$:

   $$\begin{aligned} [i]_{[k]} : \mathbb{B}^k &\to \mathbb{B} \\ (a_{k-1}, \ldots, a_0)[i]_{[k]} &\mapsto a_i \end{aligned}$$

   If the width of the argument of a projection is clear from the context, the subscript is omitted. In this example we would write $(a_5, \ldots, a_0)[4]$ instead of $(a_5, \ldots, a_0)[4]_{[6]}$.

3. Concatenation:

   The concatenation of two bitvectors $\mathbf{a} = (a_{i-1}, \ldots, a_0)$ and $\mathbf{b} = (b_{j-1}, \ldots, b_0)$, of widths i and j respectively, is the bitvector function $\otimes_{[i],[j]}$ which 'glues' them together. For $i, j \in \mathbb{N}$ the set of all concatenation functions is the set of binary bitvector functions of width $i + j$, defined below:

   $$\begin{aligned} \otimes_{[i],[j]} : \mathbb{B}^i \times \mathbb{B}^j &\to \mathbb{B}^{i+j} \\ (a_{i-1}, \ldots, a_0) \otimes_{[i],[j]} (b_{j-1}, \ldots, b_0) &\mapsto (a_{i-1}, \ldots, a_0, b_{j-1}, \ldots, b_0) \end{aligned}$$

   For example $\otimes_{[2],[1]}$ is a bitvector function concatenating bitvectors of widths 2 and 1, yielding a bitvector of width 3. If the widths of the arguments are clear from the context, we usually write $\otimes$ instead of $\otimes_{[i],[j]}$.

4. Componentwise Boolean operations:

   For $k \in \mathbb{N}$ the componentwise extension of Boolean operations on $\mathbb{B}$ to k-tuples

   $$\begin{aligned} \wedge_{[k]} : \mathbb{B}^k \times \mathbb{B}^k &\to \mathbb{B}^k \\ \vee_{[k]} : \mathbb{B}^k \times \mathbb{B}^k &\to \mathbb{B}^k \\ \neg_{[k]} : \mathbb{B}^k &\to \mathbb{B}^k \\ 0^k : &\to \mathbb{B}^k \\ 1^k : &\to \mathbb{B}^k \end{aligned}$$

are defined as usual[2], e.g. $(a_{k-1}, \ldots, a_0) \vee_{[k]} (b_{k-1}, \ldots, b_0) := (a_{k-1} \vee b_{k-1}, \ldots, a_0 \vee b_0)$. For example $\wedge_{[3]}$ is a bitvector function of width 3, usually written as $\wedge$ if the width of its arguments is clear.

6.2 Many-Sorted Algebras

Many-sorted algebras provide the algebraic framework for representing bitvector functions. They are an extension of universal algebras, where sorts partition the carrier of a many-sorted algebra into disjoint subsets, and its operations are defined over the partitioned carrier. However, ignoring anomalities such as empty carrier sets, all results for universal algebras carry over to many-sorted algebras, as illustrated below.[3]

Definition 6.4 (Sort, (Many-Sorted) Carrier, (Many-Sorted) Operation)
Given a finite non-empty set $\mathcal{S}$, the elements of $\mathcal{S}$ are called *sorts* (sort symbols). Sort symbols will be denoted by Greek letters (α, β, γ, ...). For each sort $\alpha \in \mathcal{S}$ let A_α be a set, such that for different sorts α and β, A_α and A_β are disjoint. Let $A = \dot{\bigcup}_{\alpha \in \mathcal{S}} A_\alpha$, then A is called an *$\mathcal{S}$-sorted carrier*. An *n-ary operation of sort α* on A is function

$$f : A_{\alpha_1} \times \cdots \times A_{\alpha_n} \to A_\alpha$$

Examples 6.5 (Operations on Bitvectors)
Bitvectors are sorted by their widths. For $k \in \mathbb{N}$ let $\mathcal{S} = \{1, \ldots, k\}$ be the set of bitvector widths (sorts) considered. For $i, j \leq k, i \neq j$ the sets of bitvectors of widths i and j, $\mathbb{B}^i$ and $\mathbb{B}^j$, are disjoint. For $n \in \mathbb{N}$, let ${}^n\mathbb{B}$ be the set of all bitvectors of width equal or less to n, i.e. ${}^n\mathbb{B} := \dot{\bigcup}_{i=1}^{k} \mathbb{B}^k$.

1. The Boolean operations on $\mathbb{B}$ are operations of width 1 on ${}^n\mathbb{B}$.

2. For i, k with $0 \leq i < k \leq n$ the bit projections $[i]_{[k]} : \mathbb{B}^k \to \mathbb{B}$ are operations on ${}^n\mathbb{B}$ of width (sort) 1.

3. For $i + j \leq n \in \mathbb{N}$ the concatenations $\otimes_{[i],[j]} : \mathbb{B}^i \times \mathbb{B}^j \to \mathbb{B}^{[i+j]}$ are operations of width (sort) $i + j$ on ${}^n\mathbb{B}$.

4. For $i \leq n$ the Boolean operations $\wedge_{[i]}$, $\vee_{[i]}$, $\neg_{[i]}$, $(0 \cdots 0)_{[i]}$, and $(1 \cdots 1)_{[i]}$ on $\mathbb{B}^i$ are all operations of width (sort) i on ${}^n\mathbb{B}$.

Definition 6.6 (Signature of an Operation)
Given a set of sorts $\mathcal{S}$, and an $\mathcal{S}$-sorted carrier A, the *signature* of an m-ary operation $f : A_{\alpha_1} \times \cdots \times A_{\alpha_m} \to A_\alpha$ (of sort α) on A, is the string $\alpha_1 \cdots \alpha_m \alpha \in \mathcal{S}^{m+1}$, usually written as $f : \alpha_1 \times \cdots \times \alpha_m \to \alpha$, if $m > 0$, and as $f : \alpha$, if $m = 0$ (i.e. if f is a 0-ary operation).

[2]Note that for each k these operations define a Boolean algebra on $\mathbb{B}^k$ where the all zero and all one vectors $0^k := (0, \ldots, 0)_{[k]}$ and $1^k := (1, \ldots, 1)_{[k]}$ serve as 0 and 1. (see Example 2.2.)

[3]For a similar notation of many-sorted algebras cf. e.g. [Ave95], using families of carriers and variables instead of partitions.

Example 6.7 (Signatures of Bitvector Operations)
The bit projections $[i]_{[k]}$ for $i < k \leq n$, the concatenations $\otimes_{[i],[j]}$ for $i + j \leq n$, and the Boolean operations $\wedge$, $\vee$, $\neg$, 0, and 1 carry the following signatures:

$$\begin{array}{rl} [i]_{[k]}: & k \to 1 \\ \otimes_{[i],[j]}: & i \times j \to i + j \\ \wedge: & 1 \times 1 \to 1 \\ \vee: & 1 \times 1 \to 1 \\ \neg: & 1 \to 1 \\ 0: & 1 \\ 1: & 1 \end{array}$$

Definition 6.8 (Many-Sorted Algebra)
Let $\mathcal{S}$ be a set of sort symbols. Let $A = \bigcup_{\alpha \in \mathcal{S}} A_\alpha$ be an $\mathcal{S}$-sorted carrier, and let $F = \{f_1, \ldots, f_i, \ldots, f_m\}$ be m n_i-ary operations on A, then $\mathcal{A} = (A, F)$ is called a *many-sorted algebra*. The *signature* of the many-sorted algebra $\mathcal{A}$ is an m-tuple $(s_1, \ldots, s_i, \ldots, s_m)$ of non-empty strings over $\mathcal{S}$, such that for each i, with $1 \leq i \leq m$, s_i is the signature of the corresponding operation f_i.

Examples 6.9 (Bitvector Algebras)

1. (The) Bitvector algebra:

   Let $F = \{\wedge, \vee, \neg, 0, 1\}$ be a set of Boolean operations on $\mathbb{B}$ (such that $(\mathbb{B}, F)$ is a Boolean algebra). For $n \in \mathbb{N}$ let ${}^n\mathbb{B} := \dot{\bigcup}_{i \leq n} \mathbb{B}^i$ denote the set of all bitvectors of width n or less, and let

   $$\begin{array}{rcl} {}^nF_\otimes & := & \{\cdot \otimes_{[i][j]} \cdot : \mathbb{B}^i \times \mathbb{B}^j \to \mathbb{B}^{i+j} : i + j \leq n\} \\ {}^nF_{[\,]} & := & \{\cdot[i]_{[k]} : \mathbb{B}^k \to \mathbb{B} : 0 \leq i < k \leq n\} \\ {}^nF & := & F \cup {}^nF_\otimes \cup {}^nF_{[\,]} \end{array}$$

   be sets of all *concatenation* and *bit projection operations* on ${}^n\mathbb{B}$. Then the many-sorted algebra ${}^n\mathbf{B} := ({}^n\mathbb{B}, {}^nF)$ is the smallest *bitvector algebra* (of width n), and the operations ${}^nF$ are called bitvector operations. Note that ${}^nF$ contains $n^2 + 5$ operations ($n \cdot (n-1)/2$ concatenations, $n \cdot (n+1)/2$ bit projections and 5 Boolean operations).

2. Algebra of bitvector functions: Let $({}^n\mathbb{B})^{({}^n\mathbb{B})^m}$ be the *set of all bitvector functions* of width n (or less) in $n * m$ (or less) arguments, m (or less) of each width $k \leq n$ (at most $n * m$ arguments). Let ${}^nF$ be the set of bitvector operations on ${}^n\mathbb{B}$ as above. Then the bitvector operations on $({}^n\mathbb{B})^{({}^n\mathbb{B})^m}$, also denoted by ${}^nF$, are defined pointwise by the operations ${}^nF$ on ${}^n\mathbb{B}$ as follows.

   (a) For each two bitvector functions of width 1, $f_{[1]} : \mathbb{B}^{\alpha_1} \times \cdots \times \mathbb{B}^{\alpha_l} \to \mathbb{B}$, $g_{[1]} : \mathbb{B}^{\alpha_1} \times \cdots \times \mathbb{B}^{\alpha_l} \to \mathbb{B}$ ($\in \mathbb{B}^{({}^n\mathbb{B})^m}$), let $(f_{[1]} \vee g_{[1]}) : \mathbb{B}^{\alpha_1} \times \cdots \times \mathbb{B}^{\alpha_l} \to \mathbb{B}$, such that

   $$(f_{[1]} \vee g_{[1]})(\mathbf{x}_1, \ldots, \mathbf{x}_l) := f_{[1]}(\mathbf{x}_1, \ldots, \mathbf{x}_l) \vee g_{[1]}(\mathbf{x}_1, \ldots, \mathbf{x}_l)$$

   (We define $(f_{[1]} \wedge g_{[1]})$, and $(\neg f_1)$ accordingly.)

(b) For each two bitvector functions of widths i and j with $i + j \leq n$, in $l \leq n * m$ arguments, $f_{[i]} : \mathbb{B}^{\alpha_1} \times \cdots \times \mathbb{B}^{\alpha_l} \to \mathbb{B}^i$ $(\in (\mathbb{B}^i)^{({}^n\mathbb{B})^m})$, and $g_{[j]} : \mathbb{B}^{\alpha_1} \times \cdots \times \mathbb{B}^{\alpha_l} \to \mathbb{B}^j$ $(\in (\mathbb{B}^j)^{({}^n\mathbb{B})^m})$, we define $(f_{[i]} \otimes_{[i][j]} g_{[j]}) : \mathbb{B}^{\alpha_1} \times \cdots \times \mathbb{B}^{\alpha_l} \to \mathbb{B}^j$ $(\in (\mathbb{B}^{i+j})^{({}^n\mathbb{B})^m})$, such that

$$(f_{[i]} \otimes_{[i][j]} g_{[j]})(\mathbf{x}_1, \ldots, \mathbf{x}_l) := f_{[i]}(\mathbf{x}_1, \ldots, \mathbf{x}_l) \otimes_{[i][j]} g_{[j]}(\mathbf{x}_1, \ldots, \mathbf{x}_l)$$

(c) For each bitvector function of width $k \leq n$ in $l \leq n * m$ arguments, $f_{[k]} : \mathbb{B}^{\alpha_1} \times \cdots \times \mathbb{B}^{\alpha_l} \to \mathbb{B}^k$ $(\in (\mathbb{B}^k)^{({}^n\mathbb{B})^m})$, and $0 \leq i < k$ the function $f_{[k]}[i]_{[k]} : \mathbb{B}^{\alpha_1} \times \cdots \times \mathbb{B}^{\alpha_l} \to \mathbb{B}$, is defined such that

$$(f_{[k]}[i]_{[k]})(\mathbf{x}_1, \ldots, \mathbf{x}_l) := (f_{[k]}(\mathbf{x}_1, \ldots, \mathbf{x}_l))[i]_{[k]}$$

(Note, that for $l = 1$ and $k = n$, we have $f_{[k]}[i]_{[k]} = \pi_i$, with π_i as in Example 2.9, Pg. 17.)

Then $({}^n\mathbf{B})^{({}^n\mathbf{B})^m} := (\ ({}^n\mathbb{B})^{({}^n\mathbb{B})^m}\ ,\ {}^nF\ )$ is called *algebra of bitvector functions* (of width n, or less, in at most $n * m$ arguments of width n, or less). As will be shown later, every bitvector function is composed of projections (bitvector variables), Boolean operations (a), concatenations (b), and bit projections (c).

3. For all $i \in \mathbb{N}$, let $F_i = \{\wedge_{[i]}, \wedge_{[i]}, \neg_{[i]}, 0^i, 1^i\}$ be sets of bitwise Boolean operations on $\mathbb{B}^i$, and for $n \in \mathbb{N}$ let $F' = \dot{\bigcup}_{i \leq n} F_i$. Then $({}^n\mathbb{B}, F')$ is a many-sorted algebra collecting all the Boolean algebras over $\mathbb{B}^i$.

As for universal algebras, many-sorted algebras sharing some common aspects can be collected in a class. For example, many-sorted algebras with the same signature can be grouped in a class.

Example 6.10
The bitvector algebra, the algebra of bitvector functions, and the bitvector term algebra of width less or equal to n (defined above) have the same signature.

Classes of algebras, sharing the same signature, can be specified by signature specifications, as below.

Definition 6.11 (Signature Specification)
Let F be a set of names for operations, called *function symbols*. A function $\Sigma : F \to \mathcal{S}^+$ associating each function symbol with a signature, is called *signature specification*.

Given a set of function symbols, a set of sort symbols, and a signature specification, many-sorted algebras can be derived by associating the function symbols with operations of the same signature, the *interpretations* of these function symbols. To formally distinguish the realms of function symbols (syntax) and operations in some algebra (semantics) the operation associated with a function symbol f in an algebra $\mathcal{A}$ is denoted as $[\![f]\!]^{\mathcal{A}}$, and called *interpretation of f in $\mathcal{A}$*. However, this is not necessary in most cases since the respective algebra is clear from the context. Therefore we often write f instead of $[\![f]\!]^{\mathcal{A}}$. As for universal algebras, a many-sorted algebra $\mathcal{A}$, with a signature specified by $\Sigma : F \to \mathcal{S}^+$ and with operations $F^{\mathcal{A}}$, is called a *(many-sorted) Σ-algebra*.

6.2.1 Many-Sorted Terms and Term Algebras

Given a signature specification $\Sigma : F \to \mathcal{S}^+$ and an $\mathcal{S}$ sorted set of variables we can construct many-sorted terms as defined below, which are then used as carrier of a many-sorted term algebra.

Definition 6.12 (Σ-Terms)
Let $\Sigma : F \to \mathcal{S}^+$ be a signature specification. For each $\alpha \in \mathcal{S}$ let X_α be a set of variables, and let $X = \dot{\bigcup} X_\alpha$ be the set of all variables. (Let $F \cap X = \emptyset$.) A function symbol $f \in F$ with $\Sigma(f) = \alpha_1\alpha_2 \ldots \alpha_n\alpha$ is called a function symbol of sort α. If $n = 0$, then f is called a *constant symbol of sort* α. The set of all Σ-*terms of sort* α *over* X, $T^\alpha_\Sigma(X)$, is defined inductively as:

1. Each α-variable is an α-term, i.e. $\forall \alpha \in \mathcal{S} : X_\alpha \subseteq T^\alpha_\Sigma(X)$.
2. Let $f \in F$ be a function symbol of sort α.
   (a) Each constant symbol of sort α is an α-term, i.e. if $\Sigma(f) = \alpha$ then $f \in T^\alpha_\Sigma(X)$.
   (b) Application of a function symbol of sort α to terms yields an α-term, i.e. if $\Sigma(f) = \alpha_1\alpha_2 \cdots \alpha_n\alpha$, and if $t_1, t_2, \ldots, t_n$ are terms with $t_i \in T^{\alpha_i}_\Sigma(X)$, then $f(t_1, t_2, \ldots, t_n) \in T^\alpha_\Sigma(X)$.

The set of all Σ-*terms over* X is the set of all terms with arbitrary sort, i.e. $T_\Sigma(X) := \dot{\bigcup}_{\alpha \in \mathcal{S}} T^\alpha_\Sigma(X)$.

Note that for $\alpha \neq \beta$, $T^\alpha_\Sigma(X) \cap T^\beta_\Sigma(X) = \emptyset$. We can now build Σ-term algebras as usual.

Example 6.13 (Bitvector Terms and Bitvector Term Algebra)
Bitvector terms are build over *bitvector variables*, which can take values according to their width. They are denoted in bold face and their widths are indicated in square brackets as subscript. For example, $\mathbf{x}_{[2]}$ can take the values $(0, 0)$, $(0, 1)$, $(1, 0)$, and $(1, 1)$. The width may be omitted, if clear from the context or irrelevant. For $n, m \in \mathbb{N}$ let $X := \dot{\bigcup}_{i \leq n} X_i$ be a set of *bitvector variables*, where for each $i \leq n$, X_i are sets of m bitvector variables of width i ($|X_i| = m$). Let ${}^nF$ be the set of bitvector function symbols, then the set of bitvector terms over X of width k, $T^k_{\mathcal{BV}}(X)$, is defined recursively according to the definition above, such that:

1. The constants 0 and 1 are bitvector terms of width 1.
2. If $s_{[1]}$ and $t_{[1]}$ are bitvector terms of width 1, so are $s_{[1]} \wedge t_{[1]}$, $s_{[1]} \vee t_{[1]}$, and $\neg s_{[1]}$.
3. For $k \leq n$ each bitvector variable $\mathbf{x}_{[k]} \in X_k$ of width k is a bitvector term of width k.
4. If $s_{[i]}$ and $s_{[j]}$ are bitvector terms of widths i and j, respectively, such that $i + j = k$, then $s_{[i]} \otimes_{[i],[j]} s_{[j]}$ is a bitvector term of width k.
5. If $s_{[k]}$ is a bitvector term of width k then $s_{[k]}[i]_{[k]}$ is a bitvector term of width 1.

The set of all bitvector terms is ${}^nT_{\mathcal{BV}}(X) := \dot{\bigcup}_{k\leq n} T^k_{\mathcal{BV}}(X)$. For example

$$(((\mathbf{x}_{[2]})[1]_{[2]}) \wedge ((\mathbf{y}_{[3]})[0]_{[3]})) \otimes_{[1][2]} (((\mathbf{y}_{[3]})[1]_{[3]}) \otimes_{[1][1]} ((\mathbf{x}_{[2]})[0]_{[2]}))$$

is a bitvector term of width 3. Here $\mathbf{x}_{[2]}$ and $\mathbf{y}_{[3]}$ are bitvector variables of width 2 and 3, respectively, and $((\mathbf{x}_{[2]})[1]_{[2]}) \wedge ((\mathbf{y}_{[3]})[0]_{[3]})$ denotes the conjunction of bit 1 of $\mathbf{x}$ and bit 0 of $\mathbf{y}$. The whole term denotes the concatenation of this conjunction with bit 1 of $\mathbf{y}$ and bit 0 of $\mathbf{x}$. If the widths of the subterms and operations is clear from the context we usually drop the sort indices and write:

$$(\mathbf{x}[1] \wedge \mathbf{y}[0]) \otimes \mathbf{y}[1] \otimes \mathbf{x}[0]$$

The *bitvector term algebra* has the bitvector terms ${}^nT_{\mathcal{BV}}(X)$ as carrier and the operations ${}^nF$ on ${}^nT_{\mathcal{BV}}(X)$ are defined such that (formal) application of an m-ary operation f of width k on m terms $t_1, \ldots, t_m$ (of compatible widths) yields the respective term $f(t_1, \ldots, t_m)$ of width k. For example, let $s_{[i]}$, and $s_{[j]}$ be bitvector terms of width i and j, respectively, such that $i + j = k \leq n$, then applying $\otimes_{[i],[j]}$ onto $s_{[i]}$, and $s_{[j]}$) yields the bitvector term $s_{[i]} \otimes_{[i],[j]} s_{[j]}$. The bitvector term algebra $({}^nT_{\mathcal{BV}}(X), {}^nF_{\mathcal{BV}})$ is denoted as ${}^n\mathcal{T}_{\mathcal{BV}}(X)$.

The structure of many-sorted terms is defined exactly as in the single-sorted case (Def. 3.2.1). Additionally for each position p in a term t the subterm of t at position p, $t|_p$ is equipped with a sort ($sort(t|_p)$) according to the signature of the function symbol at this position. The *sort of a term* is then the sort of the function symbol at its root position ϵ. Operations on terms, such as *replacement* and *substitution*, carry over from the single-sorted case, iff they respect the sorts:

Let s, t be terms, and let p be a position in t. If $sort(s) = sort(t|_p)$, we denote t *replaced with* s *at position* p by $t[s]_p$, as usual. (Otherwise, i.e. if $sort(s) \neq sort(t|_p)$, a replacement is not permitted.)

For $X = \dot{\bigcup}_{\alpha\in\mathcal{S}} X_\alpha$, and terms $T_\Sigma(X) = \dot{\bigcup}_{\alpha\in\mathcal{S}} T^\alpha_\Sigma(X)$, a *substitution* is a function $\sigma : X \to T_\Sigma(X)$ with $\sigma := \dot{\bigcup}_{\alpha\in\mathcal{S}} \sigma_\alpha$, such that $\sigma_\alpha : X_\alpha \to T^\alpha_\Sigma(X)$. As in the single sorted case, substitution naturally extends to a mapping from terms to terms. We write $t\sigma$ instead of $\sigma(t)$. If x is a variable and s is a term such that $sort(x) = sort(s)$, then x substituted with s in t is denoted as $t[x/s]$. (We usually require that the sets of variables X_α are finite and write a substitution σ as $\sigma = (x_1 \mapsto t_1, \ldots, x_n \mapsto t_n)$. If we consider countably infinite sets of variables we require that for only finitely many x, $\sigma(x) \neq x$.)

Example 6.14
Let $t \equiv (\mathbf{x}[1] \wedge \mathbf{y}[0]) \otimes (\mathbf{y}[1] \otimes \mathbf{x}[0])$ be the bitvector term from the last example. The term t is depicted graphically as labeled directed tree in Fig. 6.1. Then 111, 121, and 21 are positions in t, where 111 and 121 are variable positions, $sort(t|_{111}) = 2$, $sort(t|_{121}) = 3$, and $sort(t|_{21}) = 1$ (since $t|_{111} = \mathbf{x}$, $t|_{121} = \mathbf{y}$, and $t|_{21} = \mathbf{y}[1]$). Replacing t at position 21 with $s \equiv \mathbf{x}[1] \wedge \mathbf{y}[1]$ yields $t' \equiv (\mathbf{x}[1] \wedge \mathbf{y}[0]) \otimes ((\mathbf{x}[1] \wedge \mathbf{y}[1]) \otimes \mathbf{x}[0])$, and substituting $\mathbf{y}$ with $\mathbf{x}[1] \otimes \mathbf{x}$ using $\sigma = (\mathbf{y} \mapsto \mathbf{x}[1] \otimes \mathbf{x})$ yields $t'' = t\sigma \equiv (\mathbf{x}[1] \wedge (\mathbf{x}[1] \otimes \mathbf{x})[0]) \otimes ((\mathbf{x}[1] \otimes \mathbf{x})[1] \otimes \mathbf{x}[0])$.

6.2.2 Homomorphisms

Many-sorted-algebras with the same signature can be related by homomorphisms.

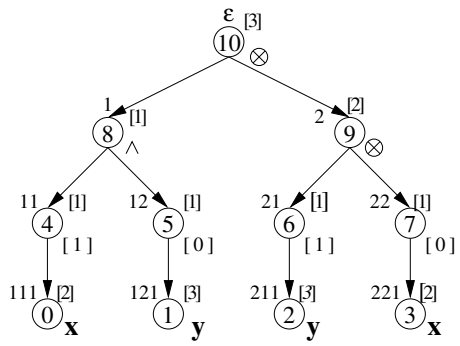


Figure 6.1: Graphical Representation of the Bitvector Term $(\mathbf{x}[1] \wedge \mathbf{y}[0]) \otimes (\mathbf{y}[1] \otimes \mathbf{x}[0])$

Definition 6.15 (Homomorphism)
Let $\mathcal{S}$ be a set of sort symbols and let $A = \dot{\bigcup}_{\alpha \in \mathcal{S}} A_\alpha$, $B = \dot{\bigcup}_{\alpha \in \mathcal{S}} B_\alpha$ be $\mathcal{S}$-sorted carriers. Given two many-sorted algebras $\mathcal{A} = (A, F^{\mathcal{A}})$ and $\mathcal{B} = (B, F^{\mathcal{B}})$ with the same signature, a function $\phi : A \to B$ with $\phi := \dot{\bigcup}_{\alpha \in \mathcal{S}} \phi_\alpha$, such that $\phi_\alpha : A_\alpha \to B_\alpha$, is called a *homomorphism* from $\mathcal{A}$ into $\mathcal{B}$ iff for all n-ary operations $f^{\mathcal{A}}$ on A, and arguments $a_1, \ldots, a_n \in A$ of compatible sorts, there is an n-ary operation $f^{\mathcal{B}}$ on B with the same signature as $f^{\mathcal{A}}$, such that $\phi(f^{\mathcal{A}}(a_1, \ldots, a_n)) = f^{\mathcal{B}}(\phi(a_1), \ldots, \phi(a_n))$.

Examples 6.16 (Homomorphisms)

1. For $i \leq n$ let X_i be pairwise disjoint finite sets of bitvector variables of width i, respectively, such that for each i, $|X_i| = m$. Let $X := \dot{\bigcup}_{i \leq n} X_j$ be the set of all $n \cdot m$ bitvector variables. Let $\varphi : X \to {}^n\mathbb{B}$ be a function mapping each variable to a bitvector, where $\varphi := \dot{\bigcup}_{i \leq n} \varphi_i$, such that $\varphi_i : X_i \to \mathbb{B}^i$. Then φ can be extended to a homomorphism $\hat{\varphi} : {}^nT(X) \to {}^n\mathbf{B}$ such that for each variable $\mathbf{x} \in X$, $\hat{\varphi}(\mathbf{x}) := \varphi(\mathbf{x})$, for the constant symbols 0 and 1, $\hat{\varphi}(0) := 0$, and $\hat{\varphi}(1) := 1$, and for all other terms $t \equiv f(t_1, \ldots, t_m)$, $\hat{\varphi}(t) := [\![f]\!](\hat{\varphi}(t_1), \ldots, \hat{\varphi}(t_m))$. (Note that $[\![f]\!]$ denotes the interpretation of the function symbol f in ${}^n\mathbf{B}$. Let for example t_1 and t_2 be terms with $sort(t_1) = 2$ and $sort(t_2) = 3$, then $\hat{\varphi}(t_1 \otimes_{[2][3]} t_2) = [\![\otimes_{[2][3]}]\!](\hat{\varphi}(t_1), \hat{\varphi}(t_2)) = \hat{\varphi}(t_1) \otimes_{[2][3]} \hat{\varphi}(t_2)$.) By this construction, $\hat{\varphi}$ is a homomorphism sending any bitvector term of width k to a bitvector of width k. As in the single-sorted case, φ is called *variable assignment* and $\hat{\varphi}(t)$ is the *valuation of* t *under* φ in the many-sorted algebra ${}^n\mathbf{B}$. (We usually write φ instead of $\hat{\varphi}$.)

2. Let X be the set of bitvector variables from above. Let, as before, $({}^n\mathbb{B})^{({}^n\mathbb{B})^m}$ denote the set of all bitvector functions of width n or less, in m or less arguments of width $k \leq n$ each (at most $n * m$ arguments). For $1 \leq i \leq m$ and $1 \leq j \leq n$ let $\pi_{i,j} : ({}^n\mathbb{B})^m \to \mathbb{B}^j$ be the $i \cdot j$'th projection of $({}^n\mathbb{B})^m$ onto $\mathbb{B}^j$, i.e.

$$\pi_{i,j}(\mathbf{x}_{1,[1]}, \ldots, \mathbf{x}_{m,[1]}, \ldots, \mathbf{x}_{1,[n]}, \ldots, \mathbf{x}_{m,[n]}) := \mathbf{x}_{i,[j]}$$

The mapping $\phi : X \to ({}^n\mathbb{B})^{({}^n\mathbb{B})^m}$ with $\phi = \dot{\bigcup}_{1 \leq j \leq n} \phi_j$, such that $\phi_j : X_j \to (\mathbb{B}^j)^{({}^n\mathbb{B})^m}$ and $\phi_j(\mathbf{x}_{i,[j]}) = \pi_{i,j}$ extends to a surjective homomorphism (also denoted as ϕ) from ${}^n\mathcal{T}_{\mathcal{BV}}(X)$ onto $({}^n\mathbb{B})^{({}^n\mathbb{B})^m}$. (E.g. $\phi_{i+j}(t_1 \otimes_{[i][j]} t_2) = \phi_i(t_1) \otimes_{[i][j]} \phi_j(t_2)$.) Hence, every bitvector term denotes a bitvector function.

6.2.3 Congruences Generated by Identities

As for universal algebras, congruences on the carrier of a many-sorted algebra can be derived from a set of identities.

Let $A = \dot{\bigcup}_{\alpha \in \mathcal{S}} A_\alpha$. Then an *identity* is a pair $(a, b) \in A^2$, such that $sort(a) = sort(b)$. Let $\mathcal{A} = (A, F)$ be a many-sorted algebra. An equivalence relation $\approx$ on A respecting the sorts (i.e. with $a \approx b \Rightarrow sort(a) = sort(b)$) is called a (many-sorted) *congruence on* A, iff for all n-ary operations f on A, with $f : \alpha_1 \times \alpha_n \to \alpha$, $a_1 \approx b_1, \ldots, a_n \approx b_n$, and $sort(a_1) = \alpha_1, \ldots, sort(a_n) = \alpha_n$, we have $f(a_1, \ldots, a_n) = f(b_1, \ldots, b_n)$. For each $\alpha \in \mathcal{S}$, the congruence $\approx$ partitions the elements of A_α into congruence classes $[a_\alpha]_\approx := \{b_\alpha \in A_\alpha : a_\alpha \approx b_\alpha\}$. The set of all congruence classes of sort α is then $A/_{\approx,\alpha} := \{[a_\alpha]_\approx : a_\alpha \in A_\alpha\}$, and the set of all congruence classes is $A/_\approx := \dot{\bigcup}_{\alpha \in \mathcal{S}} A/_{\approx,\alpha}$.

Given a signature specification $\Sigma : \mathcal{S} \to F^+$, and a Σ-algebra $\mathcal{A} = (A, F)$, the identities holding in $\mathcal{A}$, and the models of a set of identities E, are defined as in the single sorted case (see Def. 2.6). Again, given a set of identities E, the congruence $\approx_E$ induced by E is the smallest (w.r.t. set inclusion) congruence on A, comprising E. Again, we write $E \models s = t$, if $s = t$ is a semantic consequence of E, i.e. if $s = t$ holds in all models of E.

The quotient algebra $\mathcal{A}/_\approx$ (of $\mathcal{A}$ modulo $\approx$) is defined as for universal algebras. Let $A = \dot{\bigcup}_{\alpha \in \mathcal{S}} A_\alpha$, let $\mathcal{A} = (A, F)$ be a many-sorted algebra, and let $\approx$ be a congruence on A. Then $\mathcal{A}/_\approx$ denotes the associated quotient algebra. (Note that $\approx$ respects the sorts, i.e. for each $a_\alpha \in A_\alpha$ the elements of $[a_\alpha]_\approx$ all have sort α.)

For a signature specification Σ, let $T_\Sigma(X)$ be the set of terms over a set of variables X. As in the single sorted case, the congruence $\approx_E$ induced by a set of identities E, defines a quotient algebra $\mathcal{T}_\Sigma(X)/_{\approx_E}$ of the term algebra $\mathcal{T}_\Sigma(X)$.

Example 6.17 (Bitvector-Algebra modulo Bitvector-Identities)
Let ${}^n\mathcal{T}_{\mathcal{BV}}(X)$ be the bitvector term algebra of width less or equal to n. Let $E_{\mathcal{BV}}$ be the set of bitvector identities defined below, together with the Boolean identities for terms of width 1 (cf. Example 2.7, Pg. 16).

(A1) $\mathbf{a}_{[1]}[0]_{[1]} = \mathbf{a}_{[1]}$

(A2) $\mathbf{a}_{[k]} = \mathbf{a}_{[k]}[k-1]_{[k]} \otimes_{[1][k-1]} (\mathbf{a}_{[k]}[k-2]_{[k]} \otimes_{[1][k-2]} \cdots \otimes_{[1][1]} \mathbf{a}_{[k]}[0]_{[k]})$ with $(0 \leq k \leq n)$

(A3) $(\mathbf{a}_{[i]} \otimes_{[i][j]} \mathbf{b}_{[j]}) \otimes_{[i+j][k]} \mathbf{c}_{[k]} = \mathbf{a}_{[i]} \otimes_{[i][j+k]} (\mathbf{b}_{[j]} \otimes_{[j][k]} \mathbf{c}_{[k]})$ with $(0 < i, 0 < j, 0 < k, i + j + k \leq n)$

(A4) $(\mathbf{a}_{[i]} \otimes_{[i][j]} \mathbf{b}_{[j]})[k]_{[i+j]} = \mathbf{b}_{[j]}[k]_{[j]}$ with $(k < j, 0 < i, 0 < j, 0 \leq k \leq i + j \leq n)$

(A5) $(\mathbf{a}_{[i]} \otimes_{[i][j]} \mathbf{b}_{[j]})[k]_{[i+j]} = \mathbf{a}_{[i]}[k-j]_{[i]}$ with $(k \geq j, 0 < i, 0 < j, 0 \leq k \leq i + j \leq n)$

The first identity **(A1)** states that bit projection of a bit is the bit itself. The identities **(A2)** require a bitvector to be the concatenation of its components, while **(A3)** describe the pseudo[4] associativity of the concatenation operations. The identities **(A4)** and **(A5)**

[4]We say pseudo associativity because, after dropping the type indices the identities read as a single identity $(\mathbf{a} \otimes \mathbf{b}) \otimes \mathbf{c} = \mathbf{a} \otimes (\mathbf{b} \otimes \mathbf{c})$. However, the concatenations are in fact different operations.

describe how bit projections distribute over concatenations. Using these identities we can derive for example:

1. $(\mathbf{x}_{[1]} \wedge 1)[0]_{[1]} = \mathbf{x}_{[1]} \wedge 1 = \mathbf{x}_{[1]}$, or short $(\mathbf{x} \wedge 1)[0] = \mathbf{x} \wedge 1 = \mathbf{x}$.

2. $(\mathbf{y}_{[3]}[2]_{[3]} \otimes_{[1][1]} \mathbf{y}_{[3]}[1]_{[3]}) \otimes_{[2][1]} \mathbf{y}_{[3]}[0]_{[3]} = \mathbf{y}_{[3]}[2]_{[3]} \otimes_{[1][2]} (\mathbf{y}_{[3]}[1]_{[3]} \otimes_{[1][1]} \mathbf{y}_{[3]}[0]_{[3]}) = \mathbf{y}_{[3]}$, or short $(\mathbf{y}[2] \otimes \mathbf{y}[1]) \otimes \mathbf{y}[0] = \mathbf{y}[2] \otimes (\mathbf{y}[1] \otimes \mathbf{y}[0]) = \mathbf{y}$.

3. $(\mathbf{y}_{[3]} \otimes_{[3][1]} \mathbf{x}_{[1]})[2]_{[4]} = \mathbf{y}_{[3]}[0]_{[3]}$, or short $(\mathbf{y}_{[3]} \otimes \mathbf{x}_{[1]})[2] = \mathbf{y}[0]$.

Let $E_{\mathcal{BV}}$ be the set of bitvector identities above, then the bitvector term algebra modulo $\approx_{E_{\mathcal{BV}}}$, ${}^n\mathcal{T}_{\mathcal{BV}}(X)/_{\approx_{E_{\mathcal{BV}}}}$, the bitvector algebra ${}^n\mathbf{B}$, and the algebra of bitvector functions $({}^n\mathbb{B})^{({}^n\mathbb{B})^m}$ are models of $E_{\mathcal{BV}}$. We will see later that ${}^n\mathcal{T}_{\mathcal{BV}}(X)/_{\approx_{E_{\mathcal{BV}}}}$ (the bitvector term algebra modulo bitvector identities) is isomorphic to the algebra of bitvector functions $({}^n\mathbb{B})^{({}^n\mathbb{B})^m}$.

6.2.4 Free Algebras

Definition 6.18 (Subalgebra)
Let C be a class of many-sorted algebras with the same signature. Let $\mathcal{A} = (A, F^{\mathcal{A}})$ and $\mathcal{B} = (B, F^{\mathcal{B}})$ be algebras of C with $A = \dot{\bigcup}_{\alpha \in \mathcal{S}} A_\alpha$ and $B = \dot{\bigcup}_{\alpha \in \mathcal{S}} B_\alpha$ such that $B \subseteq A$, i.e. for all $\alpha \in \mathcal{S}$, $B_\alpha \subseteq A_\alpha$. $\mathcal{B}$ is called *subalgebra* of $\mathcal{A}$ iff for all $f : \alpha_1 \times \cdots \times \alpha_n \to \alpha$ and for all $b_1 \in B_{\alpha_1}, \ldots, b_n \in B_{\alpha_n}$ we have $f^{\mathcal{A}}(b_1, \ldots, b_n) = f^{\mathcal{B}}(b_1, \ldots, b_n)$.

Using the definition above the notions of the *subalgebra of $\mathcal{A}$ generated by $X \subseteq A$* and that of a *free algebra in the class C with generating set X* carry over literally from universal algebra (Def. 2.8) to many-sorted algebras.

Examples 6.19 (Free Algebras)

1. The algebra of bitvector terms ${}^n\mathcal{T}_{\mathcal{BV}}(X)$ is free in the class of algebras with the bitvector signature with generating set X. The quotient algebra ${}^n\mathcal{T}_{\mathcal{BV}}(X)/_{\approx_{E_{\mathcal{BV}}}}$ is free with generating set X in the class of algebras in which $E_{\mathcal{BV}}$ hold, since $E_{\mathcal{BV}}$ does not contain any trivial identities (i.e. no identities of the form $x = y$ with $x, y \in X$ for distinct x and y). The algebras ${}^n\mathcal{T}_{\mathcal{BV}}(\emptyset)$, and ${}^n\mathcal{T}_{\mathcal{BV}}(\emptyset)/_{\approx_{E_{\mathcal{BV}}}}$ are free with empty generating set. For example the bitvector $(0, 1, 0)$ is represented by the congruence class of the bitvector term $0 \otimes (1 \otimes 0)$ modulo $\approx_{E_{\mathcal{BV}}}$. (Here ${}^n\mathcal{T}_{\mathcal{BV}}(\emptyset)/_{\approx_{E_{\mathcal{BV}}}}$ is the initial algebra in the class of bitvector algebras.)

2. The algebra of bitvector functions is free in the class of algebras with bitvector signature with generating set $\{\pi_{i,j} : 1 \leq i \leq m, 1 \leq j \leq n\}$ where $\pi_{i,j}$ is the $i \cdots j$'th projection of $({}^b\mathbb{B})^m$ onto $\mathbb{B}^j$ such that $\pi_{i,j}(\mathbf{x}_{1,[1]}, \ldots, \mathbf{x}_{m,[1]}, \ldots, \mathbf{x}_{1,[n]}, \ldots, \mathbf{x}_{m,[n]}) := \mathbf{x}_{i,[j]}$ (as defined as in 2 of the last example). (The construction of an isomorphism between ${}^n\mathcal{T}_{\mathcal{BV}}(X)/_{\approx_{E_{\mathcal{BV}}}}$ and $({}^n\mathbb{B})^{({}^n\mathbb{B})^m}$ will be treated next.)

Note, that every homomorphism from ${}^n\mathcal{T}_{\mathcal{BV}}(X)$ into ${}^n\mathbf{B}$ is uniquely defined by the images of X since ${}^n\mathcal{T}_{\mathcal{BV}}(X)$ is free in its class with generating set X. By choosing φ, $\varphi : {}^nT(X) \to {}^n\mathbb{B}$ is unique for a fixed interpretation of the function symbols in ${}^n\mathbf{B}$, and the set of all extensions $\{\varphi : {}^nT(X) \to {}^n\mathbb{B}$ for $\varphi : X \to {}^n\mathbb{B}\}$ is the set of all homomorphisms. Hence, for two bitvector terms s, t we have ${}^n\mathcal{T}(X)/_{\approx_{E_{\mathcal{BV}}}} \models s = t$ iff $\forall \varphi : \varphi(s) = \varphi(t)$.

6.2.5 Syntactic Consequences

As before, a congruence $\approx_E$ on terms can also be derived from a set of identities $E \subseteq T_\Sigma(X) \times T_\Sigma(X)$, by closing the relation E under reflexivity, symmetry, transitivity and substitutions. The theoretical results from Section 3.2 carry over to the many-sorted case, allowing to reason either syntactically or semantically.

To show that it is safe to consider bitvector terms modulo the set of bitvector axioms instead of bitvector functions, i.e. that this is an appropriate representation and set of axioms it has to be shown that the algebra of bitvector functions and the algebra of bitvector terms modulo bitvector axioms are isomorphic.

We already know, that each bitvector term uniquely denotes a bitvector function, as we constructed a surjective homomorphism $\phi : {}^n\mathcal{T}_{\mathcal{BV}}(X) \to ({}^n\mathbb{B})^{({}^n\mathbb{B})^m}$ in Example 6.16. To show that ${}^n\mathcal{T}_{\mathcal{BV}}(X)/_{\approx_{E_{\mathcal{BV}}}}$ and $({}^n\mathbb{B})^{({}^n\mathbb{B})^m}$ are isomorphic the following remains to be shown.

1. Each bitvector function $f : \mathbb{B}^{n_1} \times \cdots \times \mathbb{B}^{n_k} \to \mathbb{B}^{n_f} \in ({}^n\mathbb{B})^{({}^n\mathbb{B})^m}$ can be expressed as a bitvector term of sort n_f over k bitvector variables of width n_1 through n_k.

2. Two bitvector terms are in the same equivalence class modulo $E_{\mathcal{BV}}$ iff they represent the same function.

First we define a normal form for bitvector terms, to be used as representative of their respective equivalence classes modulo $E_{\mathcal{BV}}$. It is shown that each bitvector term can be (syntactically) mapped to its normal form, and thus to its equivalence class, by application of bitvector axioms. Then we show that each bitvector function can be expressed as bitvector term in (CDNF). Finally, the completeness of bitvector axioms ensures that terms, representing the same function are in the same congruence class.

A bitvector term t of width n is in *concatenation disjunctive normal form* (CDNF) iff it is a concatenation of conjunctions of disjunctions of literals over bitvector variables of width 1 or bit-projections of bitvector variables of width greater 1, i.e. t has the form $t \equiv (d_{n-1} \otimes_{[1][n-1]} (d_{n-2} \otimes_{[1][n-1]} \cdots d_0))$, with $d_i \equiv \bigvee_j c_{j_i}$, $c_{j_i} \equiv \bigwedge_k l_{k_{j_i}}$, and $l_{k_{j_i}}$ are of the from $\mathbf{x}_{[1]}$, $\neg\mathbf{x}_{[1]}$, $\mathbf{x}_{[m]}[o]$, or $\neg(\mathbf{x}_{[m]}[o])$ (with $m > 1$, $o < m$ and some i, j_i, and k_{j_i}). We define the *concatenation conjunctive normal form* (CCNF) of a bitvector term accordingly.

By construction, the mapping $\phi_{\approx,\alpha} : A_\alpha \to A/_\approx$, with $a_\alpha \mapsto [a_\alpha]_\approx$, is surjective and $\phi_\approx := \dot{\bigcup}_{\alpha\in\mathcal{S}} \phi_{\approx,\alpha}$ is a homomorphism from $\mathcal{A}$ into $\mathcal{A}/_\approx$. ($\phi_{\approx,\alpha}$ is called the *canonical homomorphism.*) In particular, the function $\phi_{\approx_{E_{\mathcal{BV}}}} : {}^nT(X) \to {}^nT(X)/_{\approx_{E_{\mathcal{BV}}}}$, mapping bitvector terms to their congruence classes modulo the set of bitvector identities $E_{\mathcal{BV}}$, is a homomorphism from the algebra of bitvector terms ${}^n\mathcal{T}(X)$ into ${}^n\mathcal{T}(X)/_{\approx_{E_{\mathcal{BV}}}}$. The function $\phi_{\approx_{E_{\mathcal{BV}}}}$ is implicitly defined by the construction algorithm for CDNF below.

Lemma 6.20
Every bitvector term t can be written in CDNF using the bitvector identities as follows: Apply the bitvector identity **(A2)** once to t and all variables occurring in t (as rule directed from left to right). Subsequently apply the bitvector identities **(A1)**, **(A3)**, **(A4)**, and **(A5)** and the Boolean identities $\neg(\mathbf{a} \wedge \mathbf{b}) = \mathbf{a} \vee \mathbf{b}$, $\neg(\mathbf{a} \vee \mathbf{b}) = \mathbf{a} \wedge \mathbf{b}$, $\neg\neg(\mathbf{a}) = \mathbf{a}$, $(\mathbf{a} \vee \mathbf{b}) \wedge \mathbf{c} = (\mathbf{a} \wedge \mathbf{c}) \vee (\mathbf{b} \wedge \mathbf{c})$, and $\mathbf{a} \wedge (\mathbf{b} \vee \mathbf{c}) = (\mathbf{a} \wedge \mathbf{b}) \vee (\mathbf{a} \wedge \mathbf{c})$ (as rules directed from left to right) until no more identities are applicable this way. Then the resulting term is in CDNF.

Proof: If $t \equiv \mathbf{x}_{[1]}$, the process yields $\mathbf{x}_{[1]}$. If $t \equiv \mathbf{x}_{[k]}$ for $k > 1$, the process yields $\mathbf{x}_{[k]}[k-1] \otimes (\mathbf{x}_{[k]}[k-2] \otimes \cdots \mathbf{x}_{[k]}[0])$, hence variables are in normal form. If t is not a variable, after the first rule application it has a concatenation on the toplevel if it is not of width 1. For all terms t of width 1, $t[0]$ is changed into t, For all non Boolean terms t of the form $t_1 \otimes t_2$ no rule is applicable. If t has the form $(t_1 \otimes t_2)[k]$ it is changed into a term of the form $t_1[l]$ or $t_2[k]$. This way all intermediate concatenations of terms and bit-projections are removed. Bit-projections occur directly at non-Boolean variables, and concatenations occur at the top of the term only. The intermediate term is solely made up by Boolean connectives which are written in DNF. ■

The same holds for CCNF if the rules $(\mathbf{a} \vee \mathbf{b}) \wedge \mathbf{c}) = (\mathbf{a} \wedge \mathbf{c}) \vee (\mathbf{b} \wedge \mathbf{c})$, and $\mathbf{a} \wedge (\mathbf{b} \vee \mathbf{c}) = (\mathbf{a} \wedge \mathbf{b}) \vee (\mathbf{a} \wedge \mathbf{c})$ are replaced with their duals, i.e. $(\mathbf{a} \wedge \mathbf{b}) \vee \mathbf{c} = (\mathbf{a} \vee \mathbf{c}) \wedge (\mathbf{b} \vee \mathbf{c})$, and $\mathbf{a} \vee (\mathbf{b} \wedge \mathbf{c}) = (\mathbf{a} \vee \mathbf{b}) \wedge (\mathbf{a} \vee \mathbf{c})$.

Remark 6.21 (Minterm Normal Form for Bitvector Terms)
Since every Boolean term can be written as sum (disjunction) of minterms, every bitvector term can be written as concatenation of disjunctions of minterms over bit projections. (Just write it in CDNF and then use the Boolean identities to derive the minterms.) Disjunctions of minterms can be written in a unique form by imposing a total order on them, and minterms can be written in a unique form by imposing a total order on variables. This gives rise to a unique normal form for bitvector terms, allowing to compare them syntactically. If s and t are terms and s' and t' denote their normal forms, we have $E_{\mathcal{BV}} \vdash s = t$ iff $s' \equiv t'$. Note that this is just a theoretical concept and is most likely useless for practical application since s' and t' have exponential size.

Lemma 6.22 (Functional Completeness of Bitvector Terms)
Let $f : \mathbb{B}^{n_1} \times \cdots \times \mathbb{B}^{n_k} \to \mathbb{B}^{n_f} \in ({}^{n}\mathbb{B})^{({}^{n}\mathbb{B})^{m}}$. Then f can be represented by a term $t \in \mathcal{T}^{n_f}_{\mathcal{BV}}(X)$ with $X = \{\mathbf{x}_1, \ldots, \mathbf{x}_k\}$ and for all $i \leq k$, $sort(\mathbf{x}_i) = n_i$.

Proof: It can be shown that any bitvector function can be expressed as concatenation of disjunctions (sums) of minterms over bit projections of bitvector variables. (see Appendix A.1 for the full proof.) ■

The algebra of bitvector functions is functionally complete, i.e. any bitvector function can be expressed as bitvector term.

Examples 6.23

1. In the proof above let $\mathbf{x}$ and $\mathbf{y}$ be bitvector variables of width 2 and 3 respectively. Let $m = 2$, $n_1 = 2$, $n_2 = 3$ and $\mathbf{a} = ((0,1),(1,1,0))$. Then $f^1_{\mathbf{a}} = \neg\mathbf{x}[1] \wedge \mathbf{x}[0] \wedge \mathbf{y}[2] \wedge \mathbf{y}[1] \wedge \neg\mathbf{y}[0]$.

2. Let $\wedge_{[2]} : \mathbb{B}^2 \times \mathbb{B}^2 \to \mathbb{B}^2$ be the bitwise conjunction for vectors of width 2. Let $\mathbf{x}_{[2]}$ and $\mathbf{y}_{[2]}$ be bitvector variables of width 2. Let x_i and y_i denote $\mathbf{x}_{[2]}[i]_{[2]}$ and $\mathbf{y}_{[2]}[i]_{[2]}$, respectively. Further let's drop the $\wedge$ within minterms. Then

$$\begin{aligned} \wedge_{[2]}(\mathbf{x}_{[2]}, \mathbf{y}_{[2]}) &= (x_1x_0y_1y_0 \vee x_1x_0y_1\neg y_0 \vee x_1\neg x_0y_1y_0 \vee x_1\neg x_0y_1\neg y_0) \\ &\otimes (x_1x_0y_1y_0 \vee x_1x_0\neg y_1y_0 \vee \neg x_1x_0y_1y_0 \vee \neg x_1x_0\neg y_1y_0) \end{aligned}$$

$$\begin{tikzcd}
 & {}^{n}\mathcal{T}_{\mathcal{BV}}(X) \arrow[dl, "\phi_{\approx_{E_{\mathcal{BV}}}}"'] \arrow[dr, "\phi"] & \\
{}^{n}\mathcal{T}_{\mathcal{BV}}(X)/_{\approx_{E_{\mathcal{BV}}}} \arrow[rr, shift left, "\phi_{\mathcal{BV}}"] & & ({}^{n}\mathbb{B})^{({}^{n}\mathbb{B})^{m}} \arrow[ll, shift left, "\phi^{-1}_{\mathcal{BV}}"]
\end{tikzcd}$$

Figure 6.2: Isomorphic Bitvector Algebras

Since for any bitvector function a bitvector term can be constructed like this, the set of bitvector terms is functionally complete. Note that this is also the case taking any other Boolean base instead of the Boolean operations F for its construction.

Lemma 6.24 (Completeness of Bitvector Identities)
Two bitvector terms s and t are in the same equivalence class modulo $\approx_{E_{\mathcal{BV}}}$ iff they represent the same bitvector function, i.e.

$$[s]_{\approx_{E_{\mathcal{BV}}}} = [t]_{\approx_{E_{\mathcal{BV}}}} \Leftrightarrow \phi(s) = \phi(t)$$

Proof: See Appendix A.2. ■

The lemma above implies that the class of all algebras formed by taking subalgebras, homomorphic images and products of it, and iterating this process is an equational class, given by the equations holding in the algebra of bitvector functions.

Corollary 6.25
The algebra of bitvector functions $({}^{n}\mathbf{B})^{({}^{n}\mathbf{B})^{m}}$ and the algebra of bitvector terms modulo the bitvector identities ${}^{n}\mathcal{T}_{\mathcal{BV}}(X)/_{\approx_{E_{\mathcal{BV}}}}$ are isomorphic, and $\phi_{\mathcal{BV}} : {}^{n}T(X)/_{\approx_{E_{\mathcal{BV}}}} \rightarrow ({}^{n}\mathbb{B})^{({}^{n}\mathbb{B})^{m}}$ with $\phi_{\mathcal{BV}}([t]_{\approx_{E_{\mathcal{BV}}}}) := \phi(\phi_{\approx_{E_{\mathcal{BV}}}}(t))$ is an isomorphism.

(Note, that the equivalence classes modulo $\approx_{E_{\mathcal{BV}}}$ are represented by terms (in CDNF). The construction is depicted in Fig. 6.2.)

Bitvector algebras along with decision procedures have been proposed before (cf. [CRM97, MR98, Joh01]). However, neither of them was able to deal with the great variety of operations nicely. Often they are left functionally incomplete to allow for efficient decision procedures. We show in the following how our bitvector algebras can be extended to incorporate arbitrary HDL-operations on fixed-sized bitvectors. Decision procedures are discussed in Chapter 7.

6.3 Extended Bitvector Syntax

Given an algebra, the set of operations is often extended for convenience by defining operations, which are basically shortcuts for more complex applications of different basic operations. This is of special importance in the context of bitvectors since hardware description languages use a very rich set of operations, including (basic) bitvector operations, but also arithmetic, memory accesses, equality, and multiplexers. They can however

all be reduced to subsequent application of bitvector operations and are thus just abbreviations and do not add more algebraic content. It is however useful to work with these abbreviations, since we can describe their interrelation by identities derived from basic identities about basic operations.

Example 6.26 (Defined Operation)
Let ${}^n\mathbf{B} = ({}^n\mathbb{B}, {}^n F)$ be the bitvector algebra. Then for $k \leq n$ we can define the additional operation $\wedge_{[k]} : \mathbb{B}^k \times \mathbb{B}^k \to \mathbb{B}^k$ using the operations ${}^nF$ as follows.

$$\mathbf{x}_{[k]} \wedge_{[k]} \mathbf{y}_{[k]} \quad := \quad (\mathbf{x}_{[k]}[k-1] \wedge \mathbf{y}_{[k]}[k-1]) \otimes \cdots \otimes (\mathbf{x}_{[k]}[0] \wedge \mathbf{y}_{[k]}[0])$$

The expected algebraic properties of this operation, e.g. commutativity and associativity, can be derived from the basic bitvector identities.

1. Commutativity:

$$\begin{aligned} \mathbf{x} \wedge_{[k]} \mathbf{y} &= (\mathbf{x}[k-1] \wedge \mathbf{y}[k-1]) \otimes \cdots \otimes (\mathbf{x}[0] \wedge \mathbf{y}[0]) \\ &= (\mathbf{y}[k-1] \wedge \mathbf{x}[k-1]) \otimes \cdots \otimes (\mathbf{y}[0] \wedge \mathbf{x}[0]) \\ &= \mathbf{y} \wedge_{[k]} \mathbf{x} \end{aligned}$$

2. Associativity:

$$\begin{aligned} & \mathbf{x} \wedge_{[k]} (\mathbf{y} \wedge_{[k]} \mathbf{z}) \\ = \; & \mathbf{x} \wedge_{[k]} ((\mathbf{y}[k-1] \wedge \mathbf{z}[k-1]) \otimes \cdots \otimes (\mathbf{y}[0] \wedge \mathbf{z}[0])) \\ = \; & (\mathbf{x}[k-1] \wedge ((\mathbf{y}[k-1] \wedge \mathbf{z}[k-1]) \otimes \cdots \otimes (\mathbf{y}[0] \wedge \mathbf{z}[0]))[k-1]) \\ & \otimes \cdots \otimes (\mathbf{x}[0] \wedge ((\mathbf{y}[k-1] \wedge \mathbf{z}[k-1]) \otimes \cdots \otimes (\mathbf{y}[0] \wedge \mathbf{z}[0]))[0]) \\ = \; & (\mathbf{x}[k-1] \wedge (\mathbf{y}[k-1] \wedge \mathbf{z}[k-1])) \otimes \ldots \otimes (\mathbf{x}[0] \wedge (\mathbf{y}[0] \wedge \mathbf{z}[0])) \\ = \; & ((\mathbf{x}[k-1] \wedge \mathbf{y}[k-1]) \wedge \mathbf{z}[k-1]) \otimes \ldots \otimes ((\mathbf{x}[0] \wedge \mathbf{y}[0] \wedge) \mathbf{z}[0]) \\ = \; & (((\mathbf{x}[k-1] \wedge \mathbf{y}[k-1]) \otimes \cdots \otimes (\mathbf{x}[0] \wedge \mathbf{y}[0]))[k-1] \wedge \mathbf{z}[k-1]) \\ & \otimes \cdots \otimes (((\mathbf{x}[k-1] \wedge \mathbf{y}[k-1]) \otimes \cdots \otimes (\mathbf{x}[0] \wedge \mathbf{y}[0]))[0] \wedge \mathbf{z}[0]) \\ = \; & ((\mathbf{x}[k-1] \wedge \mathbf{y}[k-1]) \otimes \cdots \otimes (\mathbf{x}[0] \wedge \mathbf{y}[0])) \wedge_{[k]} \mathbf{z} \\ = \; & (\mathbf{x} \wedge_{[k]} \mathbf{y}) \wedge_{[k]} \mathbf{z} \end{aligned}$$

Given further bitwise Boolean operations 0^k, 1^k, $\vee_{[k]}$ and $\neg_{[k]}$ way we can show in the same way the other Boolean identities for each k.

In general we can distinguish the set of operations into generators (generating the elements of the carrier of the algebra) and defined operations, called operators, which are defined algorithmically. However, these sets may be different for two algebras with the same signature. For example the carrier ${}^n\mathbb{B}$ of ${}^n\mathbf{B}$ is generated by the operations 0, 1 and the concatenations $\otimes_{[1][k]}$ $(k < n)$, while the set of all bitvector functions (which is the carrier of the algebra of bitvector functions) is generated by 0, 1, all bit-projections $[i]_{[j]}$ $(0 \leq i < j \leq n)$, and the concatenations $\otimes_{[1][k]}$ $(k < n)$.

We will see that input, output and state variables of hardware designs can be represented by bitvector variables, and that the output and transition functions are bitvector functions, which are represented as bitvector terms. We extend the set of bitvector operations to a subset of Verilog-HDL [ver95]. The extended bitvector signature specification along with the usual notation of the operations used in this thesis is given in Table B.1, where each line defines the signature for a whole set of operators.

Table 6.1: Typical Bitvector Terms

BV term	Abbreviation	description
$(0001101)_{[7]}$	**13**	constant
$\mathbf{a}_{[n]} = \mathbf{b}_{[n]}$	$\mathbf{a} = \mathbf{b}$	equality
$\mathbf{x}_{[2]} \otimes_{[2][4][6]} \mathbf{b}_{[4]}$	$\mathbf{x}_{[2]} \otimes \mathbf{y}_{[4]}$	concatenation
$\mathbf{a}_{[4]}[\mathbf{y}_{[2]}]_{[4][2][2]}$	$\mathbf{a}_{[4]}[\mathbf{y}_{[2]}]_{[2]}$	extraction
$\mathbf{a}_{[4]} \gg_{[4][2][4]} \mathbf{y}_{[2]}$	$\mathbf{a}_{[4]} \gg \mathbf{y}_{[2]}$	logic shift right
$\mathbf{a}_{[4]} *_{[4][4][4]} \mathbf{b}_{[4]}$	$\mathbf{a}_{[4]} * \mathbf{b}_{[4]}$	multiplication
$\mathbf{a}_{[4]} \wedge_{[4][4][4]} \mathbf{b}_{[4]}$	$\mathbf{a} \wedge \mathbf{b}$	bitwise conjunction

Definition 6.27 (Extended Bitvector Signature Specification $\Sigma_{\mathcal{BV}}$)
The signature specification $\Sigma_{\mathcal{BV}} : F_{\mathcal{BV}} \to \mathbb{N}^+$, as shown in Table B.1 Pg. 182, is called *(extended) bitvector signature specification.* (In this table $m * [\alpha]$ denotes $\underbrace{[\alpha] \cdots [\alpha]}_{m}$.)

Bitvector terms are build over bitvector function symbols and bitvector variables, using the signature $\Sigma_{\mathcal{BV}}$ defined above in the usual way. Typical examples for bitvector terms are given in Table 6.1 on page Pg. 89. As before, the signatures of the operations, formally indicated as subscript in the operators, are usually omitted for readability. They are usually obvious from the context, i.e. from the sorts of the arguments. This is especially important for complex bitvector terms, e.g. $\text{ite}_{[1][4][4][4]}(\mathbf{x}_{[2]} =_{[2][2][1]} \mathbf{y}_{[2]}, \mathbf{a}_{[4]}, \mathbf{b}_{[4]})$, which is abbreviated as $\text{ite}(\mathbf{x}_{[2]} = \mathbf{y}_{[2]}, \mathbf{a}_{[4]}, \mathbf{b}_{[4]})$, or even $\text{ite}(\mathbf{x} = \mathbf{y}, \mathbf{a}, \mathbf{b})$.

For simplicity the interpretations of the extended bitvector function symbols is given in Appendix B.2 in terms of bitvector functions. Useful identities can be derived from their interpretation. However, it is possible to define the meaning of each operation in terms of the basic bitvector operations making their definition independent from the algebra used. The resulting *extended bitvector algebra* is denoted as $\mathcal{A}_{\mathcal{BV}}$. The most complex operations are shown graphically in Table B.2 on Pg. 183.

Remark 6.28 (Higher Data Types)
Bitvector variables could be grouped together as vectors of vectors, (or arrays of bits), however this would be essentially a vector again, when viewed as concatenation of all vector elements. If, for example $\mathbf{a}_{[n]}$ and $\mathbf{b}_{[n]}$ are bitvector variables, then a 2 times n bitvector array would be $(\mathbf{a}_{[n]}, \mathbf{b}_{[n]})$, a vector of 2 bitvectors of width 16. Instead we will use a bitvector variable $\mathbf{m}_{[2n]}$. If $\mathbf{m}_{[2n]} = \mathbf{a}_{[n]} \otimes \mathbf{b}_{[n]}$ we use the operation read to access the elements such that $\text{read}(\mathbf{m}_{[2n]}, \mathbf{0}) = \mathbf{b}_{[n]}$ and $\text{read}(\mathbf{m}_{[2n]}, \mathbf{1}) = \mathbf{a}_{[n]}$.

6.4 Representation of Terms by Term Graphs

Terms can be represented by annotated (or labeled) directed acyclic graphs, called term graphs. They can be defined in numerous ways, most commonly as in [BN98] as directed acyclic graphs with nodes labeled with variables and function symbols as well as a string of successor nodes indicating the intended sequence of subterms in a term, or as in [Plu98], labeling some nodes called hyper edges in the same way. These two variations are essentially the same. We will use another equivalent variation of term graphs, basically following [BN98], but using edge labels instead of strings of successor nodes to model the

order of subterm occurrences. All results from [BN98] and [Plu98] trivially translate to this representation. Additionally it will allow for modeling different syntactic aspects, namely variable permutations and commutativity of function symbols, by altering node and edge labels, respectively. Since the common aspect of all these variations are the use of labeled directed acyclic graphs they will be briefly reviewed first.

6.4.1 Directed Acyclic Graphs

A *directed graph* (DG) is a pair $G = (N, E)$ of a finite set N of *nodes* (or vertices), and a finite multiset[5] $E \subseteq N \times N$ of *edges*. For each DG $G = (N, E)$ we define functions $s, t : E \to N$ assigning to every edge $e = (n_s, n_t) \in E$ its *source node* (n_s), and its *target node* (n_t), respectively. A *path* p in G of length $l + 1$ is a sequence (or string) $(e_1, \ldots, e_l) \in E^{l+1}$ of edges with $t(e_i) = s(e_{i+1})$ for $1 < i \leq l$. The path p is associated with the *node sequence* $(s(e_1), \ldots, s(e_l), t(e_l)) \in N^{l+2}$.

The *out (in) degree* of a node n in G is the number of edges starting (ending) in n, denoted as $\delta(n)$ ($\gamma(n)$). A node n is a *leaf (root)* of G iff no paths are starting (ending) in n, i.e. iff the out (in) degree of n is zero. The *subgraph* G_n of G with root node n is the the restriction of G to all paths starting from n, i.e. to all edges (E_n) and nodes (N_n) occurring in these paths.

A *homomorphism* from $G := (N, E)$ to $G' = (N', E')$ is a mapping $\phi : N \to N'$ such that for all edges $e = (n_s, n_t) \in E$, $(\phi(n_s), \phi(n_t)) \in E'$. Then ϕ extends to a mapping from E to E' also denoted by ϕ with $s'(\phi(e)) = \phi(s(e))$ and $t'(\phi(e)) = \phi(t(e))$. A homomorphism ϕ is an *isomorphism* iff ϕ is a bijective mapping from N onto N' and $\phi(E) = E$. An isomorphism is an *automorphism* iff $N = N'$ and $E = E'$.

A path is a *cycle* iff its first and last node are identical. Cycle free DGs are called *directed acyclic graphs* (DAGs). A homomorphism of a DAG is a DG homomorphism which does not create cycles. The set of nodes N of a DAG $G = (N, E)$ carries a partial order derived from the natural order on node sequences of cycle free paths, such that $n_1 > n_2$ iff n_1 occurs before n_2 in all paths. W.r.t. to this order leaves are minimal, and roots are maximal.

Lemma 6.29
If $G = (N, E)$ is a DAG and $\phi \in Aut(G)$ then $n \in N : rk(n) = rk(\phi(n))$.

Example 6.30 (DAGs)
Let $N = \{1, \ldots, 5\}$, $E = \{(3, 1), (3, 1), (4, 1), (4, 2), (5, 3), (5, 4)\}$. Then $G = (N, E)$ is a DAG, which is depicted in Fig. 6.3. Let $N' = \{1, \ldots, 7\}$, and $E' = \{(5, 1), (5, 2), (6, 3), (6, 4), (7, 5), (7, 6)\}$ then $G' = (N', E')$ is also a DAG, in particular a tree. The DAGs G and G' are shown graphically in Fig. 6.3. Then $\phi : N' \to N$ with $\phi = (1 \mapsto 1, 2 \mapsto 1, 3 \mapsto 1, 4 \mapsto 2, 5 \mapsto 3, 6 \mapsto 4, 7 \mapsto 5)$ is a homomorphism form G' to G.

[5] [Wol03] 'Multiset: A set-like object in which order is ignored, but multiplicity is explicitly significant. Therefore, multisets $\{1, 2, 3\}$ and $\{2, 3, 1\}$ are equivalent, but $\{1, 1, 2, 3\}$ and $\{2, 3, 1\}$ differ. The number of multisets of length k on n symbols is called n multichoose k, denoted as $\binom{n}{k}$.'

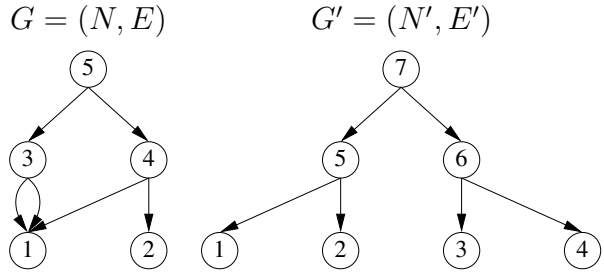


Figure 6.3: Homomorphic DAGs G and G'

6.4.2 Term Graphs as Labeled Directed Acyclic Graphs

Let (N, E) be a DAG and let L_N, L_E be sets of *node* and *edge labels*, respectively. Given two (possibly partial) *labeling functions* $ln : N \to L_N$ and $le : E \to L_E$ assigning node and edge labels to nodes and edges respectively, a *labeled* DAG (LDAG) is a tuple $G = (N, E, ln, le)$. A DAG homomorphism ϕ from (N, E) to (N', E') is an LDAG homomorphism from $G = (N, E, ln, le)$ to $G' = (N', E', ln', le')$ iff it is compatible with the node and edge labeling, i.e. iff $ln'(\phi(n)) = ln(n)$ and $le'(\phi(e)) = le(e)$ for all nodes $n \in N$ and edges $e \in E$.

Definition 6.31 (Term Graph)
Let $\Sigma : F \to \mathcal{S}^+$ be a (many-sorted) signature specification and let X be an $\mathcal{S}$-sorted set of variables (where F, $\mathcal{S}$, and X are defined as usual). Let $G = (N, E, ln, le)$ be an LDAG with the labeling functions below,

$$\begin{aligned} ln : N &\to F \cup X \\ le : E &\to \mathbb{N} \end{aligned}$$

and with the following properties.

1. If a node is labeled with a variable, then it is a leaf node. (Formally: $\forall n \in N : ln(n) = x \in X \Rightarrow \delta(n) = 0$)

2. If a node is labeled with a function symbol then the out degree of this node is the same as the arity of the function symbol. (Formally: $\forall n \in N : ln(n) = f \in F \Rightarrow \delta(n) = arity(f)$)

3. The edge labels of all outgoing edges of a node labeled with a function symbol are uniquely labeled with naturals from 1 to the arity of that function symbol. (Formally: For a node $n \in N$, let $E(n)$ be the set of edges starting in n. Then $\forall n \in N : ln(n) = f \in F \Rightarrow le|_{E(n)} : E(n) \to \{1, \ldots, arity(f)\}$ such that $le|_{E(n)}$ is bijective.)

Then G is called a *term graph* (TG).

The *i'th successor* of a node n with $ln(n) = f$ and $1 \leq i \leq arity(f)$, which is denoted as $n[i]$, is the target node of the unique edge starting in n whose label is i. The *term represented by* n is then recursively defined as

$$term(n) := ln(n)(term(n[1]), \ldots, term(n[\delta(n)]))$$

Table 6.2: Labeling Functions for DAGs

n	$ln(n)$
1	$\mathbf{x}_{[2]}$
2	$\mathbf{y}_{[2]}$
3	$+_{[2,2]}$
4	$-_{[2,2]}$
5	$*_{[2,2]}$

e	$le(e)$
$(3,1)$	1
$(3,1)$	2
$(4,1)$	1
$(4,2)$	2
$(5,3)$	1
$(5,4)$	2

n'	$ln'(n')$
1	$\mathbf{x}_{[2]}$
2	$\mathbf{y}_{[2]}$
3	$\mathbf{x}_{[2]}$
4	$\mathbf{y}_{[2]}$
5	$+_{[2,2]}$
6	$-_{[2,2]}$
7	$*_{[2,2]}$

e'	$le'(e')$
$(5,1)$	1
$(5,2)$	2
$(6,3)$	1
$(6,4)$	2
$(7,5)$	1
$(7,6)$	2

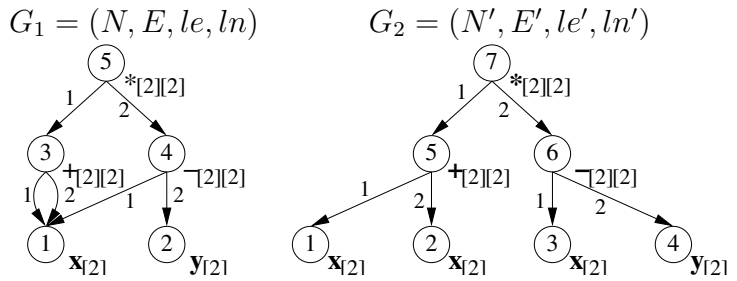


Figure 6.4: Term Graphs Representing $(\mathbf{x}_{[2]} + \mathbf{x}_{[2]}) * (\mathbf{x}_{[2]} - \mathbf{y}_{[2]})$

Example 6.32 (Term Graphs)
Let (N, E), and (N', E') be the DAGs from Example 6.30, and let $ln : N \to \Sigma_{\mathcal{BV}} \cup \{\mathbf{x}_{[2]}, \mathbf{y}_{[2]}\}$, $le : E \to \mathbb{N}$, $ln' : N' \to \Sigma_{\mathcal{BV}} \cup \{\mathbf{x}_{[2]}, \mathbf{y}_{[2]}\}$, and $le' : E' \to \mathbb{N}$ be defined as in Table 6.2 (Pg. 92), then $G_1 = (N, E, le, ln)$ and $G_2 = (N', E', le', ln')$ are both term graphs representing the bitvector term $(\mathbf{x}_{[2]} + \mathbf{x}_{[2]}) * (\mathbf{x}_{[2]} - \mathbf{y}_{[2]})$. They are depicted in Fig. 6.4 (Pg. 92).

Obviously any node in a TG represents a unique term. Conversely terms can be represented by term graphs taking the set of positions in the term as nodes, labeling them with the function symbol or variable at this position, and taking the subterm relationship as edges labeling them with the last letter in the position of the subterm. The term is then represented by the root node ϵ of the term graph, which is a tree.

A common operation on TGs is collapsing, where isomorphic subgraphs are shared. Given two TGs G and H, G *collapses* to H if there is a term graph homomorphism $\phi : G \to H$. This is denoted by $G \succeq H$. The inverse relation of collapsing is called *copying* and is denoted by $\preceq$. The non-injective homomorphisms $\succ$ and $\prec$ are called proper collapsing and proper copying.

Definition 6.33 (Fully Collapsed Term Graph)
A term graph G is *fully expanded* if there exists no H with $H \succ G$, and G is *fully collapsed* if there is no H with $G \succ H$.

A term graph G is fully collapsed if and only if for all nodes v and w, $term(v) = term(w)$ implies $v = w$.

Proposition 6.34
For every TG G there are a unique (up to isomorphism) fully expanded $\triangle\, G$, and a unique

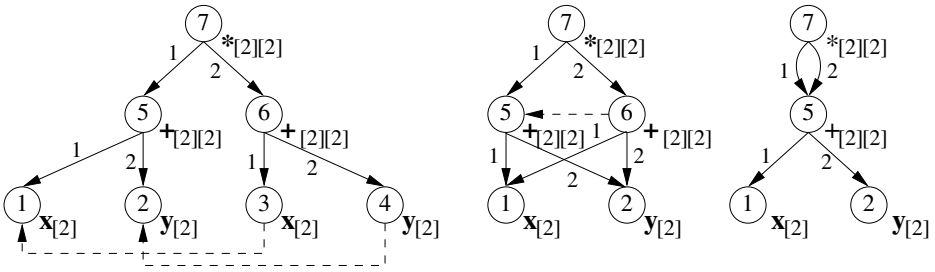


Figure 6.5: Collapsing a Term Graph

(up to isomorphism) fully collapsed TG $\triangledown G$ such that $\vartriangle G \succeq G$ and $G \succeq \triangledown G$.

Proof: See for example [BN98], [Plu98]. ■

Note that the fully expanded TG $\vartriangle G$ is a forest, and if G is a single-rooted TG, so are $\triangledown G$ and $\vartriangle G$, in which case $\vartriangle G$ is a tree. The fully collapsed terms graph $\triangledown G$ for G can be computed by congruence closure on an empty relation[6].

Example 6.35 (Collapsing Term Graphs)

1. The term graphs G_1 and G_2 from Example 6.32 are fully collapsed and fully expanded, respectively. The term graphs starting in nodes 1, 2, and 3 of G_2 are isomorphic subgraphs of G_2. Sharing them leads to a term graph isomorphic to G_1.

2. Fig. 6.5 shows the collapsing process for a labeled tree representing the bitvector term $(\mathbf{x}_{[2]}+\mathbf{y}_{[2]})*(\mathbf{x}_{[2]}+\mathbf{y}_{[2]})$ where dashed lines denote the LDAG morphism applied, as well as the congruence relation on subterms.

In the following we will only consider fully collapsed single rooted term graphs as unique representation of terms by their root node. For a single rooted TG G with root r we define $term(G) := term(r)$.

Corollary 6.36
Let s and t be terms and let G_s and G_t be single rooted TGs with $term(G_s) \equiv s$ and $term(G_t) = t$, then

$$\triangledown G_s \cong \triangledown G_t \Leftrightarrow s \equiv t$$

(Note that the same holds for the unique trees (see [Plu98]))

Instead of working with the isomorphism class of a term graph we usually take a unique representative, which can be computed by canonically numbering nodes in a term graph (see [Plu98].) Let in the following each term be represented by such unique representative fully collapsed graph.

[6][BN98]:'A relation $\sim$ on nodes of a TG is a congruence if it is an equivalence and if for all nodes u and v with $ln(u) = ln(v)$, $(\forall 1 \leq i \leq \delta(u) : u[i] \sim v[i])$ implies $u \sim v$.' Let $u \sim v$ iff $term(u) = term(v)$, then the union find algorithm (Alg. 18) can be used to compute the congruence closure, where two nodes are identified iff they represent the same subterm.

Chapter 7

Comparing Bitvector Functions

As shown in Chapter 6, the algebra of bitvector functions $({}^n\mathbf{B})^{({}^n\mathbf{B})^m}$ is isomorphic to the algebra of bitvector terms modulo bitvector identities ${}^n\mathcal{T}_{\mathcal{BV}}(X)/_{\approx_{E_{\mathcal{BV}}}}$. The bitvector (function) equivalence problem is the question whether for two bitvector terms s and t, representing functions $g, h \in ({}^n\mathbb{B})^{({}^n\mathbb{B})^m}$, $s = t$ holds in the algebra of bitvector functions $({}^n\mathbf{B})^{({}^n\mathbf{B})^m}$. As in the Boolean case, it can be normalized to the question whether $\neg(s \oplus t)$ represents 1, as $g = h$ iff $\neg(g \oplus h) = 1$. Just like for Boolean terms, the problem can be solved either semantically This is also the case for bitvector functions represented by terms over the extended bitvector signature, since the respective bitvector operations are defined in terms of the basic bitvector operations.

7.1 Decision Procedures

As on the Boolean level, the values of bitvector functions can be enumerated by evaluating them under all possible assignments, thus enumerating all possible bit combinations of the bits in all bitvector variables. This is in most cases not permissible for obvious reasons, however it can also not be avoided in the worst case since the bitvector function equivalence problem is Co-NP-complete. This is easy to see, as it is it contains the Boolean equivalence problem, and as it can be transformed into an equivalent Boolean equivalence problem in linear time and space, as shown below.

7.1.1 Boolean Domain Translation

An obvious way of solving the bitvector equivalence problem is to transfer it into a Boolean equivalence problem, for which we already know sophisticated decision procedures, as described before. For a k-ary bitvector function $f : \mathbb{B}^{n_1} \times \cdots \times \mathbb{B}^{n_m}$, in m bitvector variables $X = \{\mathbf{x}_1, \ldots, \mathbf{x}_m\}$, and with codomain $\mathbb{B}$, we construct a $n_1 + \cdots n_m$-ary Boolean function f' over Boolean variables X', with

$$X' = \{x_1^{n_1-1}, \ldots, x_1^1, x_1^0, x_2^{n_2-1}, \ldots, x_2^1, x_2^0, \ldots, x_m^{n_m-1} \ldots, x_k^1, x_m^0\}$$

such that $f' = 1$ iff $f = 1$ in the obvious way.

For example, f can be written as sum of products over literals over X', as follows: Treat bit projections as Boolean functions and let $x_i^k := \mathbf{x}_i[k]_{[n_i]}$, such that $\mathbf{x}_i = x_i^{n_i-1} \otimes$

$\cdots \otimes x_i^1 \otimes x_i^0$, and use x_i^k instead of $\mathbf{x}_i[k]_{[n_i]}$ in the construction of bitvector terms from bitvector functions. The topmost concatenation is missing since the codomain of f is $\mathbb{B}$. Then the constructed term is a Boolean term, and the represented function is a Boolean function. There is, however, a practical solution allowing a linear time and space conversion, which is described briefly below.

Each bitvector function symbol for a bitvector function of width n is represented by n Boolean terms. Let c be the largest number of Boolean subterms in either of these terms. Traversing a bitvector term (graph) of width 1 in topological order (bottom up), an equivalent Boolean term is constructed piecewise. At each subterm node of width n (bitvector function symbol), at most $c * n$ Boolean subterm nodes are constructed. The process is illustrated in Fig. 7.1.

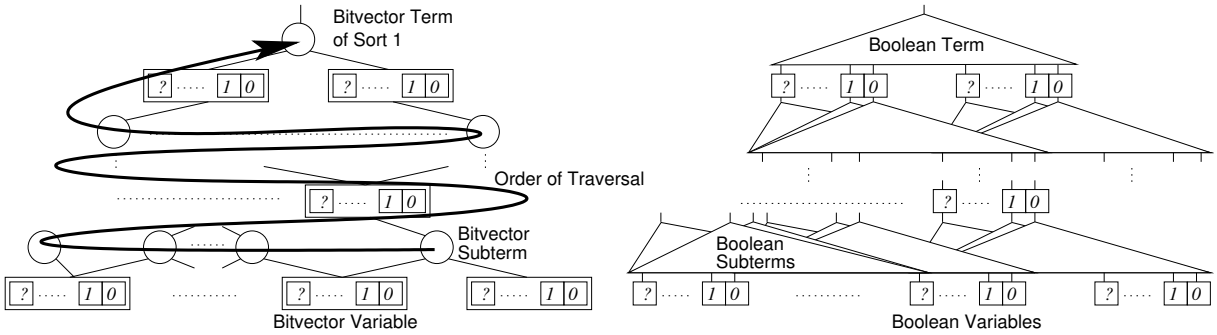


Figure 7.1: Translation of a Bitvector Term into a Boolean Term

The general advantage of solving the problem in the Boolean domain is that it benefits directly from continuous improvements on the Boolean domain. The disadvantage is that the compact structure of bitvector terms is mostly lost in big Boolean terms.

7.1.2 Alternative Approaches

In order to utilize the compact structure of bitvector terms the concept of ROBDDs and derivative approaches for the word level has been extended to bitvectors. For example binary moment diagrams, Taylor expansion diagrams, and others, have been proposed (see e.g. [SBW98, HD99, CKZR02]). In practice, they did not live up to the expectations so far, as they suffer from the same principle limitations as ROBDDs.

There have been some efforts to translate the bitvector equivalence problem into other domains than the Boolean, e.g. integer programming. The disadvantage of state of the art approaches is that they do not perform better than using Boolean provers on problems of control oriented nature, i.e. where the bitvector function is mainly described using single-bit operations, such as Boolean connectives.

However, for certain sub theories enormous speed-ups have been reported. Such sub theories are for example the theory of fixed sized bitvectors with extraction, concatenation and negation and theories for arithmetic bitvector functions (see e.g. [CRM97, MR98, Joh01]), or reductions to Presburger arithmetic (cf. [Pre27]) using integer linear programming techniques as solvers (see e.g. [BD02]). Therefore some effort has been put into combinations of Boolean provers, namely SAT and ATPG, with specialized decision procedures for sub theories of the bitvector theory lately (see e.g. [FDK98, ZKC01]). Their

integration into complete methods, handling bitvector algebras to their full extend, are for example based on combinations of equational theories (ala [Sho84]) or direct coupling of search procedures.

Abstraction techniques (see e.g. [HIKB96, VB98, BBCZ98, BGV99, PRSS99]) also seem very promising, as they can be combined with Boolean Satisfiability [VB01]. This combination yields a complete method, although the abstraction itself is not. Combining them with methods described in the following, seems worth trying, but is out of focus of this thesis.

7.2 Syntactic Preprocessing on the Register Transfer Level

As for Boolean functions, the equivalence problem for bitvector terms can be solved syntactically based on an axiomatic description of the properties of the underlying algebra. It has been shown in Lemma 6.22 that the bitvector identities $E_{\mathcal{BV}}$ form a complete set of identities for basic bitvector algebras.

The equivalence problem for extended bitvector terms can be solved by reduction to this basic case. On the other hand, useful algebraic properties of this extension can be derived from the basic bitvector axioms, as shown for bitwise conjunction in Example 6.26.

As in the single sorted case, convergent rewrite systems can be used to decide the equivalence problem for two terms, if such a system can be found. Deriving a convergent many-sorted rewrite system for the extended bitvector terms is not attempted here, because of the huge amount of operations involved. Instead, we will show below, how many-sorted rewriting can be used as preprocessing procedure for semantic decision procedures (described above).

7.2.1 Many-Sorted Rewriting

Since identities and substitutions for many-sorted terms respect sorts, but in all other aspects were defined as in the single-sorted case, all results from single-sorted rewriting carry over to the many-sorted case. According to this translation, a many-sorted term rewrite system (MTRS) is a set of rewrite rules $l \to r$, where l and r are terms of the same sort. The definitions and results for reduction relations, Birkhoff's theorem, confluence, termination, and convergence of a reduction relation can be translated word by word, and are therefore not repeated here.

There are, however, some additional results for MTRS. For example, removing all sort information from rewrite rules yields a single-sorted rewrite system. A property of a MTRS is persistent if the reduced TRS has this property. For example termination is a persistent property of an MTRS with variables of the same sort only. Conversely, if the underlying TRS of a MTRS is terminating, the MTRS has this property as well. (Note that for termination of an MTRS only well typed terms have to be considered.) The problem of persistence has been studied mostly in the context of proving properties for single sorted TRSs by imposing some sorting and proving the properties for the MTRS.

7.2.2 Representation of Rewrite Rules

In the single-sorted case rewrite rules were constructed from a given set of identities derived from or being the identities characterizing the equational properties of an algebra. The set of identities characterizing equivalences for many-sorted algebras is possibly large or even infinite. The same holds for derived sets of rewrite rules. For example bitvectors have finite but arbitrary widths less then or equal to some $n \in \mathbb{N}$ which is only bound by the available memory of the computer hardware used. Hence, for the extended bitvector signature and term algebra there is a huge amount of possible sort symbols, function symbols, and identities involved. We have to use a compact representation of such large sets which is described here.

A set of rewrite rules is specified by a *rule specification*. A basic rule specification has the $l \to r$ structure of a rewrite rule, but leaves the sorts of function symbols, variables and constants variable. Replacing these variables with values yields rewrite rules.

Example 7.1 (Rule Specification)
The rule specification below describes a set of rewrite rules.

$$\mathbf{x}_{[n]} \circlearrowright \mathbf{0}_{[m]} \quad \to \quad \mathbf{x}_{[n]}$$

Replacing n and m with sorts (naturals) yields a rewrite rule. The specified rules reduce the right rotation of a bitvector by 0 bits to the bitvector itself.

Furthermore, to generalize rules which have the same structure but use different constants, we use so called *parameters* in rule specifications. Parameters are special variables that match only constants but not any other subterm or variable. The same parameter can only have one value throughout a rule.

Example 7.2 (Rule Specification with Parameters)
Let $\mathbf{c}$ be a parameter, then for each $n \in \mathbb{N}$ and for each $\mathbf{c}_{[n]} \in \mathbb{B}^n$, the following rule specification describes rules, reducing the application of the arithmetic minus operation onto a constant bitvector to its algebraic result.

$$-_{[n][n]}\mathbf{c}_{[n]} \quad \to \quad [\![-_{[n][n]}]\!](\mathbf{c}_{[n]})$$

Here n is any natural and $\mathbf{c}$ matches any bitvector constant of with n. To ease the notation we will omit this sort specification whenever it is obvious and simply write

$$-\mathbf{c} \quad \to \quad [\![-]\!](\mathbf{c})$$

Note that we require each constant symbol to be interpreted by itself and that each domain value is a constant symbol. This is the case for bitvector algebras. Therefore it is possible to use the interpretation of a ground term inside a rewrite rule.

Some rewrite rules are specified using an additional condition, for example describing relations between sorts and values of some constant. To specify sort constraints we allow the use of some helper functions like vec_n, nat to convert between bitvectors and naturals. In general a *conditional rule specificatin* has the form $\xi \Rightarrow l \to r$, where ξ is a formula called *condition*, and l, r specify the lhs and rhs of a rewrite rule, respectively. The set of variables of the condition ξ must be a subset of the variables and sorts occurring in the lhs l of the rule.

Example 7.3 (Rule Specification with Sort Constraints)
The conditional rule specification below describes a set of rewrite rules, such that shifting a bitvector to the right by a number of bits larger than its own width is reduced to the bitvector of all zeros.

$$(nat_m(\mathbf{c}_{[m]}) >= n) \Rightarrow \mathbf{x}_{[n]} \gg \mathbf{c}_{[m]} \quad \rightarrow \quad \mathbf{0}_{[n]}$$

Here $\mathbf{c}$ is a parameter. Note that replacing $\mathbf{c}$ and m with some concrete values leads to a rewrite rule if the condition evaluates to true. This does not lead to conditional rewriting, since the generated rules are unconditional.

To deal with commutative operations and ensure termination we will use ordered rewriting. Ordering constraints are specified by conditions for rule specifications, leading to ordering constraints for rules.

Example 7.4 (Rule Specification with Ordering Constraints)
The bitwise conjunctions of bitvectors are commutative. The rule specification below describes a set of rewrite rules ordered by the rank order $<_{rko}$ such that for each $n \in \mathbb{N}$, $\mathbf{x}_{[n]} \wedge_{[n][n]} \mathbf{y}_{[n]}$ is replaced with $\mathbf{y}_{[n]} \wedge_{[n][n]} \mathbf{x}_{[n]}$ if the rank of $\mathbf{y}$ is smaller than the rank of $\mathbf{x}$.

$$(\mathbf{y} <_{rko} \mathbf{x}) \Rightarrow \mathbf{x} \wedge \mathbf{y} \quad \rightarrow \quad \mathbf{y} \wedge \mathbf{x}$$

7.2.3 Rewrite Systems for Bitvector Terms

We now develop increasingly stronger rewrite systems for bitvector terms. As in the single sorted case we start with a very basic rewrite system (constant folding) which will be subsequently extended to constant propagation, and a normalization heuristics.

Constant Folding

Consider the ground terms $t_0 \equiv \text{ite}(0,0,0)$ and $t_1 \equiv \text{ite}(1,0,0)$. Obviously $t_0 = t_1$ holds. Interpreting such ground terms, i.e. replacing them with a constant for their interpretation, can be used as rewriting approach. For example using the rewrite rules $\text{ite}(0,0,0) \rightarrow 0$ and $\text{ite}(1,0,0) \rightarrow 0$ it can be shown that $s_0 \equiv \mathbf{x} \vee \text{ite}(0,0,0)$ and $s_1 \equiv \mathbf{x} \vee \text{ite}(1,0,0)$ are equal.

The general idea is to replace ground subterms with their interpretation, whenever possible. It can be implemented by a simple form of rewriting, ground term rewriting. Recall that an identity $l = r$ is *ground* iff the lhs and rhs do not contain variables ($Var(l) = Var(r) = \emptyset$). Given a set G of ground identities a convergent term rewrite system for bitvector terms is constructed as follows. As set of ground identities take the set of all ground terms with the following properties. On the lhs there is a function symbol at the root position, which is not a constant. On the rhs there is only a constant which is equal to the interpretation of the lhs in the bitvector algebra.

We orient these equations from left to right. Then the resulting rules have the form

$$f(c_0, c_1, \ldots, c_n) \rightarrow [\![f]\!](c_0, c_1, \ldots, c_n)$$

Let R be a rewrite system constructed in such a way. By this construction no lhs term l_1 is reducible by any other rule $l_2 \rightarrow r_2$. Therefore there are no overlapping redexes for

any term t, and there are no critical pairs of rules. Then R is locally confluent, hence it is confluent. Furthermore for each rule the rhs is smaller than the lhs ($l >_{sz} r$) which ensures termination.

These rules are built into the construction process of bitvector terms. Whenever constructing a term, which is usually done bottom-up starting with subterms, they are implicitly applied.

Example 7.5 (Bottom-Up Construction of Bitvector Terms)
Consider the following term t.

$$t \equiv \text{ite}(\neg\mathbf{x}, (011), (111)) + ((100) \ll \mathbf{x})$$

We want to construct and simplify the terms $t[\mathbf{x}/\mathbf{0}]$ and $t[\mathbf{x}/\mathbf{1}]$ and check if they are equal. For $t[\mathbf{x}/\mathbf{0}]$ the following subterms are generated and immediately replaced with their normal form.

$$\begin{array}{rcl} \neg\mathbf{0} & \rightarrow_R & \mathbf{1} \\ \text{ite}(\mathbf{1}, (011), (111)) & \rightarrow_R & (011) \\ (100) \ll \mathbf{0} & \rightarrow_R & (100) \\ (011) + (100) & \rightarrow_R & (111) \end{array}$$

Similarly $t[\mathbf{x}/\mathbf{1}]$ is generated and rewritten into (111) as well. Hence $t[\mathbf{x}/\mathbf{0}] = t[\mathbf{x}/\mathbf{1}]$ holds. In some cases the approach still works even though variables are still present. For example let s_1 be as below.

$$s_1 \equiv \mathbf{y}_{[3]} + (\text{ite}(\neg\mathbf{x}, (011), (111)) + ((100) \ll \mathbf{x}))$$

Then $s_1[\mathbf{x}/\mathbf{0}] = s_1[\mathbf{x}/\mathbf{1}]$ holds. However for s_2 below $s_2[\mathbf{x}/\mathbf{0}] = s_2[\mathbf{x}/\mathbf{1}]$ cannot be shown using constant folding, since $\text{ite}(\mathbf{0}, \mathbf{y}_{[3]}, \mathbf{y}_{[3]})$ and $\text{ite}(\mathbf{1}, \mathbf{y}_{[3]}, \mathbf{y}_{[3]})$ are not reduced.

$$\begin{array}{rcl} s_2 & \equiv & \text{ite}(\mathbf{x}, \mathbf{y}_{[3]}, \mathbf{y}_{[3]}) + (\text{ite}(\neg\mathbf{x}, (011), (111)) + ((100) \ll \mathbf{x})) \\ s_2[\mathbf{x}/\mathbf{0}] & \equiv & \text{ite}(\mathbf{0}, \mathbf{y}_{[3]}, \mathbf{y}_{[3]}) + (\text{ite}(\neg\mathbf{0}, (011), (111)) + ((100) \ll \mathbf{0})) \\ s_2[\mathbf{x}/\mathbf{1}] & \equiv & \text{ite}(\mathbf{1}, \mathbf{y}_{[3]}, \mathbf{y}_{[3]}) + (\text{ite}(\neg\mathbf{1}, (011), (111)) + ((100) \ll \mathbf{1})) \end{array}$$

Therefore we will develop a stronger normalizing rewrite system below, where rules of the form $\text{ite}(\mathbf{1}, \mathbf{x}, \mathbf{y}) \rightarrow \mathbf{x}$, and $\text{ite}(\mathbf{0}, \mathbf{x}, \mathbf{y}) \rightarrow \mathbf{y}$ are allowed.

Constant Propagation

We will now extend the constant folding concept to constant propagation. The idea here is to propagate the effect of assigning a variable in a term with some value as far as possible into this term by replacing subterms involving constants with simpler terms. In contrast to constant folding, not all but some arguments of a function application in a term must be constants. (Constant folding is then a special case of constant propagation.)

Example 7.6 (Constant Propagation)
A simple example are the rules $\text{ite}(\mathbf{1}, \mathbf{x}, \mathbf{y}) \rightarrow \mathbf{x}$, and $\text{ite}(\mathbf{0}, \mathbf{x}, \mathbf{y}) \rightarrow \mathbf{y}$, for propagating the effect of assigning the condition of ite with some value. Let s_2 be as in the last example. Using the two rules above together with the constant folding rules from the last section both $s_2[\mathbf{x}/\mathbf{0}]$ and $s_2[\mathbf{x}/\mathbf{1}]$ are rewritten into $\mathbf{y}_{[3]} + (111)$, i.e. $s_2[\mathbf{x}/\mathbf{0}] \rightarrow \mathbf{y}_{[3]} + (111) \leftarrow s_2[\mathbf{x}/\mathbf{1}]$.

Let $\mathbf{x}$, $\mathbf{y}$, $\mathbf{z}$ be bitvector variables, let $\mathbf{a}$, $\mathbf{b}$, $\mathbf{c}$ be parameters, and let m, n, o, p, l, k be naturals. Let $norm(\mathbf{c}_{[m]})$ be a function cut off leading zeros of the bitvector constant $\mathbf{c}$ (i.e. $norm(\mathbf{c}_{[m]}) = vec_{[[ld(nat_m(\mathbf{c}_{[m]}))]]}(nat_m(\mathbf{c}_{[m]})))$. Then the constant propagation rules are given in Tables B.3 and B.4. These rules, together with the constant folding rules are built into the generation process of bitvector terms.

Again the properties of termination and confluence of the resulting rewrite system have to be shown to establish convergence of the resulting rewrite relation. They are easily established as follows. The rewrite system is terminating since for all rules we have $l >_{sz} r$. Since there is only one function symbol of arity greater 0 involved in the lhs of each rule there are no overlaps between rules were this symbol is different. Hence it is sufficient to consider overlaps of rules with the same function symbol. It is easy to see by inspecting the rules that no overlaps are present. Therefore the rewrite system is confluent. Convergence follows from termination and confluence.

Increasing Sharing by Partial Normalization

Finally the constant propagation approach above is extended to a convergent rewrite system which can increase the sharing in bitvector terms. This is necessary because the extended bitvector syntax is very rich and its semantics allows many ambiguities in the representation of bitvector functions since some operators have overlapping semantics. This means that one operator can be expressed by other operators. (In fact, bitvector functions can be represented by basic bitvector terms as shown before.)

As a result there are many different ways of writing the same thing. The idea of normalization is to remove ambiguities as much as possible while leaving the representation compact and avoid exponential explosion of the term. Since these are contradicting goals the following rules are used as normalizing rewrite heuristics and are not complete. Tables B.5 and B.6 show sets of rewrite rules for normalization. They are built into the generation process of terms, together with the constant folding and normalization rules described before.

These rules lead to better sharing of common subfunctions. Confluence of the resulting term rewrite system is established by a precedence on rule application which avoids critical pairs. Termination has not been shown formally, however divergence has never been observed in practice. As some rewrite rules are built into the term generation procedures, analysing the properties of the resulting rewrite system by a program is infeasable.

Chapter 8

Bounded Interval Model Checking on the Register-Transfer-Level

In this chapter the BIMC-approach introduced in Chapter 4 is extended from the bitlevel to the register transfer level. This extension is straight forward. It is described using the register file example (see Example 5.6). Then the class of regular designs is studied. On one hand designs of this class are often hard to verify, on the other hand they are especially well suited for the reduction method proposed in this thesis. A simple memory is used to illustrate the regularity. It is used as a running example for the remainder of the thesis.

8.1 Extension to the Register Transfer Level

Hardware description languages, in which the functionality of a design is first formulated, operate on the register transfer level (RTL). On the register transfer level structure is explicit. It is prevalent in data types and functional units. Bitvectors are the prevailing data type. The usual representation of sequential circuits as FSMs is based on Boolean terms. Unfortunately, some classes of sequential circuits, including large regular designs, still impose serious practical problems for bounded interval model checking, when performed on the bit level. On this level the syntactic correlation of the single bits, which is available on the register transfer level, is lost. Therefore, the regular structure of a design is not explicit in its formal representation. We want to exploit structural properties, in particular symmetries, when verifying behavior using bounded interval model checking. Therefore the explicit structure of designs and properties must be preserved. This can be achieved by representing FSMs on the register transfer level.

On the register transfer level, input, output and state variables are grouped as bitvector variables. Then output and transition functions of FSMs are bitvector functions which are represented by bitvector terms.

Definition 8.1 (RTL-design)
Let $\mathcal{M} = (\mathbb{B}^n, \mathbb{B}^m, B^k, S_0 \subseteq \mathbb{B}^k, f_O, f_S)$ be a Mealy machine. Let $\alpha_1, \ldots, \alpha_p$, $\beta_1, \ldots, \beta_q$, and $\gamma_1, \ldots, \gamma_r$ be natural numbers for bitvector widths, i.e. sorts, such that $n = \alpha_1 + \cdots + \alpha_p$, $m = \beta_1 + \cdots + \beta_q$ and $k = \gamma_1 + \cdots + \gamma_r$. Then $\mathbb{B}^n$, $\mathbb{B}^m$ and $\mathbb{B}^k$ can be grouped such

that:

$$\mathbb{B}^n = \mathbb{B}^{\alpha_1} \times \cdots \times \mathbb{B}^{\alpha_p}, \quad \mathbb{B}^m = \mathbb{B}^{\beta_1} \times \cdots \times \mathbb{B}^{\beta_q}, \text{and} \quad \mathbb{B}^k = \mathbb{B}^{\gamma_1} \times \cdots \times \mathbb{B}^{\gamma_r}$$

Then the output and transition functions f_O and f_S of the Mealy machine $\mathcal{M}$ are bitvector functions with

$$f_o : (\mathbb{B}^{\alpha_1} \times \cdots \times \mathbb{B}^{\alpha_p}) \times (\mathbb{B}^{\gamma_1} \times \cdots \times \mathbb{B}^{\gamma_r}) \rightarrow \mathbb{B}^{\beta_1} \times \cdots \times \mathbb{B}^{\beta_q}$$

$$f_S : (\mathbb{B}^{\alpha_1} \times \cdots \times \mathbb{B}^{\alpha_p}) \times (\mathbb{B}^{\gamma_1} \times \cdots \times \mathbb{B}^{\gamma_r}) \rightarrow \mathbb{B}^{\gamma_1} \times \cdots \times \mathbb{B}^{\gamma_r}$$

A constructive representation of $\mathcal{M}$ on the register transfer level is a tuple

$$\mathcal{D} = (\overline{\imath}, \overline{o}, \overline{s}, f_{S,0}, f_O, f_S)$$

called a *RTL hardware design*, where:

1. $\overline{\imath} = (i_1, \ldots, i_p)$ is the vector of bitvector *input variables* of widths $\alpha_1, \ldots, \alpha_p$,
2. $\overline{o} = (o_1, \ldots, o_q,)$ is the vector of bitvector *output variables* of widths $\beta_1, \ldots, \beta_q$,
3. $\overline{s} = (s_1, \ldots, s_r)$ is the vector of bitvector *state variables* of widths $\gamma_1, \ldots, \gamma_r$,
4. $f_{S,0} : \mathbb{B}^{\gamma_1} \times \cdots \times \mathbb{B}^{\gamma_r} \rightarrow \mathbb{B}$ is a characteristic bitvector function describing *initial states* of $\mathcal{M}$, i.e. $\overline{s} \in S_0$ iff $f_{S,0}(\overline{s}) = 1$, which is represented by a bitvector term with the same name.
5. f_O is the output function of $\mathcal{M}$ represented componentwise by bitvector terms $f_O^1, \ldots, f_O^k$ of sort $\beta_1, \ldots, \beta_q$, respectively, over input and state variables.
6. f_S is the transition function of $\mathcal{M}$ represented componentwise by bitvector terms $f_S^1, \ldots, f_S^k$ of sort $\gamma_1, \ldots, \gamma_r$, respectively, over input and state variables.

Example 8.2 (Register File)
Consider the register file from example 5.6. Its Mealy machine can be represented by the following RTL-design:

$$\begin{array}{lrll}
D = (& \overline{\imath} = & (\mathbf{w}_{[1]}, \mathbf{c}_{[2]}, \mathbf{i}_{[1]}), & \text{input variables} \\
 & \overline{o} = & (\mathbf{o}_{[1]}), & \text{output variables} \\
 & \overline{s} = & (\mathbf{m}_{[4]}), & \text{state variables} \\
 & f_{S_0} = & 1, & \text{initial states function} \\
 & f_O(\overline{\imath}, \overline{s}) = & (\text{read}(\mathbf{m}_{[4]}, \mathbf{c}_{[2]})), & \text{output function} \\
 & f_S(\overline{\imath}, \overline{s}) = & (\text{ite}(\mathbf{w}_{[1]}, \text{write}(\mathbf{m}_{[4]}, \mathbf{c}_{[2]}, \mathbf{i}_{[1]}), \mathbf{m}_{[4]}))\ ) & \text{transition function}
\end{array}$$

The process of constructing bitvector terms representing RTL-designs from hardware description languages is not covered here, instead we will assume that an RTL-design is given. Note that the grouping of binary input, output and state variables of the Mealy machine into vectors is naturally given by the data types of input, output and state variables in the HDL description of the RTL-design as shown in the following example.

Table 8.1: Textual Representation of Register File and Write Property

VERILOG-HDL code of Register File Design	Write Property in VERILOG-ITL

`module regfile (clk, w, c, i, o);`
`  input clk;`
`  input w;`
`  input [1:0] c;`
`  input i;`
`  output o;`
`  reg  m [3:0];`
`  always @(posedge clk)`
`  begin`
`  if (w)`
`  m[c] = i;`
`  end`
`  assign o = m[c];`
`endmodule // regfile`
	`theorem write;`
	
	`freeze: it = i@t,`
	`  ct = c@t;`
	`assume:`
	`  at t:  w == 1'b1;`
	`prove:`
	`  at t+1: m[ct] == it;`
	`end theorem;`

Example 8.3 (RTL-Code for Register File)
The register file in the example above is a (`clk`-synchronous) model of the VERILOG-HDL module shown in Table 8.1.

Given an RTL-design $\mathcal{D} = (\overline{\imath}, \overline{o}, \overline{s}, f_{S,0}, f_O, f_S)$ a bounded interval property for $\mathcal{D}$ is a bitvector term p over temporal input, output and state variables of $\mathcal{D}$ within a finite time interval $[\mathsf{t}, \mathsf{t}+\mathsf{n}]$, analogous to definition 4.15.

Example 8.4 (A Property of the Register File)
To verify that if $\mathbf{w}_{[1]}$ is active at time t this results in writing the current data value $\mathbf{i}_{[1]}$ to the register entry indexed by $\mathbf{c}_{[2]}$ the following property p is formulated:

$$p \equiv \mathbf{w}_{\mathsf{t}} \Rightarrow \mathrm{read}(\mathbf{m}_{\mathsf{t}+1}, \mathbf{c}_{\mathsf{t}}) = \mathbf{i}_{\mathsf{t}}$$

(This property is shown in its textual representation as VERILOG-ITL theorem in Table 8.1.)

The question whether such a property p holds for an RTL design can be reduced to the question, whether for some bitvector term p', $p' = 1$ holds in the bitvector algebra along the lines of section 4.6.2.

Example 8.5 (A Property of the Register File)
Applying the construction of section 4.6.2 results in the bitvector term p' below for which it is easy to see here that $p' = 1$ holds.

$$p' \equiv \mathbf{w}_{\mathsf{t}} \Rightarrow \mathrm{read}(\mathrm{ite}(\mathbf{w}_{\mathsf{t}}, \mathrm{write}(\mathbf{m}_{\mathsf{t}}, \mathbf{c}_{\mathsf{t}}, \mathbf{i}_{\mathsf{t}}), \mathbf{m}_{\mathsf{t}}), \mathbf{c}_{\mathsf{t}}) = \mathbf{i}_{\mathsf{t}}$$

Treating all temporal variables in the bitvector term p' as usual bitvector variables it can be transformed into a Boolean term $p'_{\mathbf{B}}$ along the lines of section 7.1 such that $p' = 1$ holds iff $p'_{\mathbf{B}} = 1$ holds (in the respective algebras).

Example 8.6 (Boolean Term for Property)
Figure 8.1 shows the the Boolean term resulting from the translation of the property generated from the design and property of Table 8.1 as and-inverter-graph with annotated signal names. (This is the actual representation which includes the freeze variables from the property.)

Checking whether $p'_{\mathbf{B}} = 1$ holds is sometimes not possible using nowadays technology for complexity reasons. We want to attack this problem by exploiting the structure of p', in particular symmetrical values of variables occurring in p'. This is done by constructing a bitvector term p'' from p' with $p'' = 1$ iff $p' = 1$ such that $p''_{\mathbf{B}} = 1$ is significantly easier to check than $p'_{\mathbf{B}} = 1$. This is mainly due to reduction of the size of p' and the number of variables involved. This approach is described in detail in the next chapter.

8.2 Regular Designs – A Comprehensive Example

Here we give a comprehensive example of a regular design where the search space for a DLL procedure operating on the Boolean level is very large and the symmetry reduction approach described in the next chapter is effective. It serves as running example within this chapter.

Regular designs, which are a special class of designs, are of special interest, since they are often hard to verify. They form a large and important class of hardware designs. Regular designs often lead to very large search spaces when solving resulting bounded model checking problems. This makes bounded model checking impractical.

Regular designs are usually constructed by repetitive use of some basic building blocks. Often many of these building blocks are treated uniformly. If this uniformity is reflected by the internal structure of the design, it is called regular. They can contain many instances of similar building blocks, such as memories, bus-systems, multiplexers, and arbiters and thus have many state variables. For example even a small memory with 8 address lines and 16 data lines has at least 2^{4096} states.

The regularity of the design structure can be observed in the structure of the terms when representing the design on the register transfer level. This structure can be exploited to ease the bounded model checking task. In particular we will see that RTL-BMC-problems for regular designs can contain many variables with symmetrical values. We want to exploit this property.

8.2.1 A Regular Memory Design

As an example for an RTL-design consider a memory. In general, a memory provides two functions: Writing data into the memory, and reading data from the memory. They constitute two modes of operation, called read-mode and write-mode. A memory has a number of memory cells, which store one data word each. Each such memory cell has an unique address. Selecting a cell for reading from it or writing to it is done by means of this address. There are two more functions, besides the usual operations of 'read' and 'write', which memories usually provide, 'reset' and 'chip select'. After power on, the memory has to be reset to a well known state. Then the data in the memory cells is erased. This function is performed by the 'reset' operation. If the environment of the memory wants

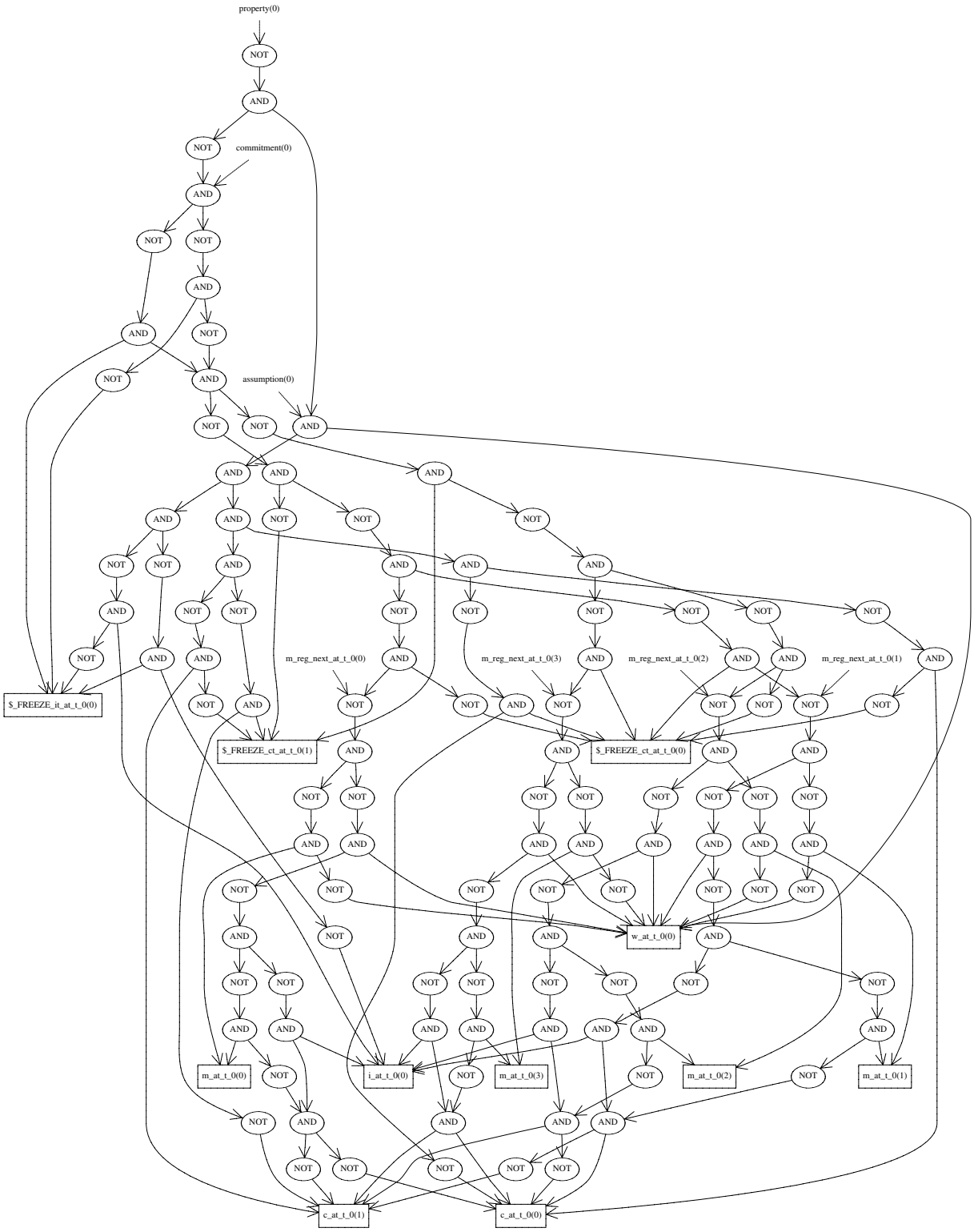


Figure 8.1: Boolean Term Representing the Write Property of the Register File

it to ignore all input, the so called chip-select signal is set to inactive mode. If this signal is inactive, the memory should not change its state. This operation is called 'inactive'.

Such a memory is depicted in Fig. 8.2 in a schematic way. The memory has 256 memory

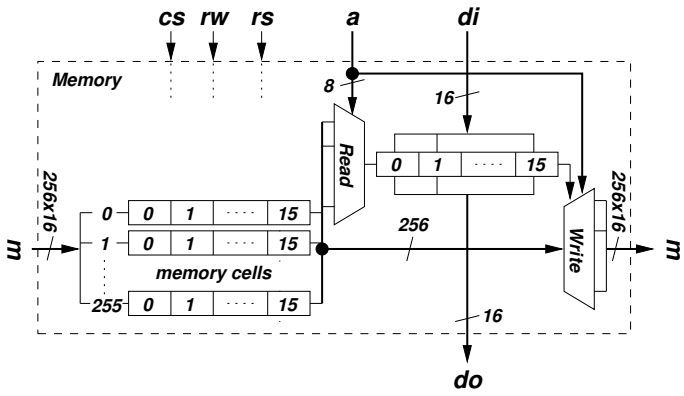


Figure 8.2: Memory Design

cells of width 16. Each cell is storing one of the data values $\mathbf{0}_{[16]}, \mathbf{1}_{[16]}, \ldots, \mathbf{65535}_{[16]}$. The memory is organized as an array of 256 vectors of width 16.

Let the memory design $\mathcal{D}_{\text{Mem}}$ be formally defined as:

$$\mathcal{D}_{\text{Mem}} = (\overline{\imath}, \overline{o}, \overline{s}, f_{S,0}, f_O, f_S)$$

such that:

1. The vector of input variables is $\overline{\imath} = (\mathbf{rs}_{[1]}, \mathbf{cs}_{[1]}, \mathbf{rw}_{[1]}, \mathbf{a}_{[8]}, \mathbf{di}_{[16]})$, where:
   - $\mathbf{a}_{[8]}$ denotes the address input,
   - $\mathbf{di}_{[16]}$ is the input data to be stored,
   - $\mathbf{rw}_{[1]}$ is a read/write-mode selector,
   - $\mathbf{cs}_{[1]}$ is chip-select, and
   - $\mathbf{rs}_{[1]}$ is the reset input.
2. The vector of output variables is $\overline{o} = (\mathbf{do}_{[16]})$, where the only output is the data output $\mathbf{do}_{[16]}$.
3. The vector of state variables is $\overline{s} = (\mathbf{m255}_{[16]}, \mathbf{m254}_{[16]}, \ldots, \mathbf{m1}_{[16]}, \mathbf{m0}_{[16]})$, where $\mathbf{mi}_{[16]}$ denotes the i'th memory cell.
4. The output function f_O is defined as

   $$f_O : \mathbb{B}^1 \times \mathbb{B}^1 \times \mathbb{B}^1 \times \mathbb{B}^8 \times \mathbb{B}^{16} \times \mathbb{B}^{256\times16} \quad \rightarrow \quad \mathbb{B}^{16}$$

   which is represented by the one element vector of bitvector terms (t_O):

   $$t_O \equiv \text{ite}(\mathbf{rs} = \mathbf{1}, \mathbf{0}_{[16]}, \text{ite}(\mathbf{cs} = \mathbf{1} \wedge \mathbf{rw} = \mathbf{0}, \text{read}(\mathbf{m255} \otimes \cdots \otimes \mathbf{m1} \otimes \mathbf{m0}, \mathbf{a}), \mathbf{0})) \quad (8.1)$$

5. The transition function f_S is defined as

$$f_S : \mathbb{B}^1 \times \mathbb{B}^1 \times \mathbb{B}^1 \times \mathbb{B}^8 \times \mathbb{B}^{16} \times \mathbb{B}^{256\times16} \quad \rightarrow \quad \mathbb{B}^{256\times16}$$

The vector of transition terms is defined as $\overline{t}_S$ with:

$$\begin{aligned} \overline{t}_S \equiv \quad (\ & \text{ite}(\mathbf{rs} = \mathbf{1}, \mathbf{0}_{[16]}, (\text{ite}(\mathbf{cs} = \mathbf{1} \wedge \mathbf{a} = \mathbf{0}_{[8]} \wedge \mathbf{rw} = \mathbf{1}, \mathbf{di}, \mathbf{m255}_{[16]}))), \\ & \text{ite}(\mathbf{rs} = \mathbf{1}, \mathbf{0}_{[16]}, (\text{ite}(\mathbf{cs} = \mathbf{1} \wedge \mathbf{a} = \mathbf{1}_{[8]} \wedge \mathbf{rw} = \mathbf{1}, \mathbf{di}, \mathbf{m254}_{[16]}))), \\ & \qquad\qquad\qquad\qquad \vdots \\ & \text{ite}(\mathbf{rs} = \mathbf{1}, \mathbf{0}_{[16]}, (\text{ite}(\mathbf{cs} = \mathbf{1} \wedge \mathbf{a} = \mathbf{255}_{[8]} \wedge \mathbf{rw} = \mathbf{1}, \mathbf{di}, \mathbf{m0}_{[16]}))) \ ) \end{aligned} \tag{8.2}$$

The output and transition functions of the memory are represented by vectors of bitvector terms. (For example '$\cdot \otimes \cdot$' denotes the concatenation of bitvectors and 'read$(\cdot, \cdot)$' denotes multiplex read[1].)

The whole content of the memory can be represented by a single bitvector variable $\mathbf{m}_{[256\times16]}$ of width 256×16, where $\mathbf{m}_{[256\times16]} = \mathbf{m255} \otimes \cdots \otimes \mathbf{m1} \otimes \mathbf{m0}$. Let in the following $\mathbf{m}_{[256\times16]}$, or simply $\mathbf{m}$, denote the vector of state variables of the memory. Then 'read$(\mathbf{m}, \mathbf{i})$' denotes the $\mathbf{i}$'th memory cell. Then for example 'read$(\mathbf{m}, \mathbf{170}) = \mathbf{m170}$'.

8.2.2 Informal specification of the memory's behavior

For the memory above, the following specification of expected behavior can be given and has to be verified to ensure its functional correctness: For any given point in time $\mathsf{t} \in \mathbb{N}$ and for any possible computation sequence of the memory, the following properties must hold:

1. *Reset*: Assume $\mathbf{rs}_\mathsf{t} = \mathbf{1}$, then the content of the memory will be erased, i.e. $\mathbf{m}_{\mathsf{t}+1} = \mathbf{0}_{[256\times16]}$.

2. *Inactive*: Assume $\mathbf{cs}_\mathsf{t} = \mathbf{0}$ and $\mathbf{rs}_\mathsf{t} = \mathbf{0}$, then the content of the memory does not change, i.e. $\mathbf{m}_{\mathsf{t}+1} = \mathbf{m}_\mathsf{t}$.

3. *Write*: Assume $\mathbf{cs}_\mathsf{t} = \mathbf{1}$, $\mathbf{rs}_\mathsf{t} = \mathbf{0}$, and $\mathbf{rw}_\mathsf{t} = \mathbf{1}$, then the input data $\mathbf{di}$ is stored at address $\mathbf{a}$, i.e. $\mathbf{di}_\mathsf{t} = \text{read}(\mathbf{m}_{\mathsf{t}+1}, \mathbf{a}_\mathsf{t})$.

4. *Read*: Assume $\mathbf{cs}_\mathsf{t} = \mathbf{1}$, $\mathbf{rs}_\mathsf{t} = \mathbf{0}$, and $\mathbf{rw}_\mathsf{t} = \mathbf{0}$, then at $\mathsf{t}+1$ the contents of the memory at address $\mathbf{a}$ is propagated to the output $\mathbf{do}$, i.e. $\mathbf{do}_{\mathsf{t}+1} = \text{read}(\mathbf{m}_\mathsf{t}, \mathbf{a}_\mathsf{t})$.

8.2.3 Formal specification of the write operation

As an example consider the property p_{write} defined below which describes a correct write operation of the memory design.

$$p_{write} \equiv (\mathbf{rs}_\mathsf{t} = \mathbf{0} \wedge \mathbf{cs}_\mathsf{t} = \mathbf{1} \wedge \mathbf{rw}_\mathsf{t} = \mathbf{1}) \Rightarrow (\mathbf{di}_\mathsf{t} = \text{read}(\mathbf{m}_{\mathsf{t}+1}, \mathbf{a}_\mathsf{t})) \tag{8.3}$$

The write operation is correctly implemented by the design if the property p_{write} holds for all possible computation sequences of the design, and at all times.

[1] See Tables B.1, Pg. 182, and B.2.25. Pg. 187.

8.2.4 Constructing an RTL-BMC-problem for the write operation

To prove that this is the case a bitvector term p'_{write} is constructed from p_{write} by replacing $\mathbf{m}_{\mathsf{t}+1}$ in the property Eq. (8.3) with the transition function for $\mathbf{m}_{\mathsf{t}+1}$ from Eq. (8.2). In this example the recursion stops after the first iteration, because the property is bound by the time interval $[\mathsf{t}, \mathsf{t}+1]$. The resulting RTL-BMC-problem is:

$$
\begin{aligned}
p'_{write} \equiv \quad & \mathbf{rs}_{\mathsf{t}} = \mathbf{0} \wedge \mathbf{cs}_{\mathsf{t}} = \mathbf{1} \wedge \mathbf{rw}_{\mathsf{t}} = \mathbf{1} \Rightarrow \\
& \mathbf{di}_{\mathsf{t}} = \text{read}(\\
& \quad (\ \text{ite}(\mathbf{rs}_{\mathsf{t}} = \mathbf{1}, \mathbf{0}, (\text{ite}(\mathbf{cs}_{\mathsf{t}} = \mathbf{1} \wedge \mathbf{a}_{\mathsf{t}} = \mathbf{255} \wedge \mathbf{rw}_{\mathsf{t}} = \mathbf{1}, \mathbf{di}_{\mathsf{t}}, \mathbf{m255}_{\mathsf{t}}))) \\
& \quad \otimes \text{ite}(\mathbf{rs}_{\mathsf{t}} = \mathbf{1}, \mathbf{0}, (\text{ite}(\mathbf{cs}_{\mathsf{t}} = \mathbf{1} \wedge \mathbf{a}_{\mathsf{t}} = \mathbf{254} \wedge \mathbf{rw}_{\mathsf{t}} = \mathbf{1}, \mathbf{di}_{\mathsf{t}}, \mathbf{m254}_{\mathsf{t}}))) \\
& \quad \otimes \cdots \\
& \quad \otimes \text{ite}(\mathbf{rs}_{\mathsf{t}} = \mathbf{1}, \mathbf{0}, (\text{ite}(\mathbf{cs}_{\mathsf{t}} = \mathbf{1} \wedge \mathbf{a}_{\mathsf{t}} = \mathbf{0} \wedge \mathbf{rw}_{\mathsf{t}} = \mathbf{1}, \mathbf{di}_{\mathsf{t}}, \mathbf{m0}_{\mathsf{t}}))) \\
& \quad), \mathbf{a}_{\mathsf{t}})
\end{aligned}
\tag{8.4}
$$

If the formula $p_{write} = 1$ is valid in the bitvector algebra, then the property p_{write} holds for the memory design.

8.2.5 Solving the RTL-BMC-problem

The formula $p'_{write} = 1$ is valid iff p'_{write} evaluates to 1 in the bitvector algebra for all possible assignments to its variables. The domains of $\mathbf{rs}_{\mathsf{t}}$, $\mathbf{cs}_{\mathsf{t}}$ and $\mathbf{rw}_{\mathsf{t}}$ are Boolean, i.e. have size 2. The data input $\mathbf{di}_{\mathsf{t}}$ has a domain of 2^{16} values. The address input $\mathbf{a}_{\mathsf{t}}$ has a domain size of 2^8. The 256 memory cells have a width of 16 bit each, i.e. $\mathbf{m}_{\mathsf{t}}$ can take 2^{256*16} different values. All the domain sizes have to be multiplied to get the number of possible assignments. In this case the number of possible assignments is $2^{3+16+8+256*16} = 2^{4123}$. This is the search space to be considered by a DP search procedure on the Boolean level. Alternatively a BDD representing p' is to be constructed over 4123 Boolean variables.

Chapter 9

Permutation Equivalence of Bitvector Functions

Permutation equivalence is an extension of equivalence of two functions to equivalence under renaming of variables. Later the problem of finding symmetrical values in bitvector functions is reduced to the permutation equivalence problem. Symmetrical values in bitvector functions can be used for preprocessing RTL-BMC-problems as described in Chapter 10.

Therefore variable renaming in bitvector functions and many-sorted terms, as well as permutation equivalence of these objects are introduced here. Then decidability of the permutation equivalence problem for bitvector functions is shown. Since this problem is Co-NP-complete, some semi-decision procedures are given to solve this problem for the particular case of bitvector terms. These procedures are based on a combination of rewriting techniques and isomorphism computation of certain term graphs (LDAGs).

To improve computational efficiency the isomorphism problem for sets of terms is translated into an automorphism problem. The relation of the automorphism groups of these graphs with automorphisms of term algebras modulo identities is described. Finally the automorphism problem for LDAGs is reduced to the automorphism problem for colored undirected graphs (CUGs) which can be solved using standard techniques.

9.1 Variable Renamings

In this section variable renamings are defined and some examples are given. Let in the following $\mathcal{S}$ be a finite set of sort symbols, let $X = \dot{\bigcup}_{\alpha \in \mathcal{S}} X_\alpha$ be an $\mathcal{S}$-sorted set. Let $\pi_\alpha : X_\alpha \to X_\alpha$ be bijections on X_α, then $\pi := \dot{\bigcup}_{\alpha \in \mathcal{S}} \pi_\alpha$ is a sort respecting bijection on X. Let X_α be finite, then X is finite and π is a permutation of X, i.e. $\pi \in Sym(X)$, where $Sym(X)$ denotes the symmetric group on X. (For a brief overview of permutations, permutation groups, and the notation used see e.g. [DH78].)

Definition 9.1 (Variable Renaming)
Let X be a finite $\mathcal{S}$-sorted set of variables, then $\pi \in Sym(X)$ is called a *variable renaming*.

Let $\mathcal{A} = (A, F)$ be an algebra free in some class C of algebras with (finite) generating set X. Then any $\pi \in Sym(X)$ extends to an automorphism of $\mathcal{A}$ [1]. In fact, any automorphism of $\mathcal{A}$ is derived from a permutation in the above fashion.

Example 9.2

1. Given a signature specification $\Sigma : F \to \mathcal{S}^+$, variable renamings extend to Σ-terms over X as follows. Let $\pi \in Sym(X)$ be a variable renaming, then π extends to an automorphism of $\mathcal{T}_\Sigma(X)$ also denoted by π, where $\pi : T_\Sigma(X) \to T_\Sigma(X)$ is defined as

$$\pi(t(s_1, \ldots, s_n)) := t(\pi s_1, \ldots, \pi s_n)$$

   for each term $t \in T_\Sigma(X)$ with subterms $s_1, \ldots, s_n$. (According to common practice, application of a variable renaming π to a term t is also denoted as $t\pi$ instead of $\pi(t)$.)

2. All automorphisms of $\mathcal{T}_\mathbb{B}(X)$ are derived from the permutations of X, the reason being that any nontrivial automorphism ψ is defined by its action on X. Would $\psi(x)$ for some $x \in X$ contain any operators then $\psi^{-1}(\psi(x)) = x$ would imply nontrivial identities in $\mathcal{T}_\mathbb{B}(X)$, which is not possible.

3. Let ${}^nT_{\mathcal{BV}}(X)$ be a set of bitvector terms[2] then $\pi \in Sym(X)$ extends to an automorphism of ${}^n\mathcal{T}_{\mathcal{BV}}(X)$. Since ${}^n\mathcal{T}_{\mathcal{BV}}(X)$ is free in the class of $\Sigma_{\mathcal{BV}}$-algebras with generating set X, π also extends to an endomorphism of any other bitvector term algebra modulo some set of identities E, ${}^n\mathcal{T}_{\mathcal{BV}}(X)/_{\approx_E}$, as long as E is not trivial.

4. Variable renamings extend to surjective homomorphisms from the algebra of bitvector terms ${}^n\mathcal{T}_{\mathcal{BV}}(X)$ to the algebra of bitvector functions $\mathbb{B}^{({}^n\mathbb{B})^m}$. Let $\pi \in Sym(X)$ be a variable renaming, then $\pi : \mathbb{B}^{({}^n\mathbb{B})^m} \to \mathbb{B}^{({}^n\mathbb{B})^m}$ is defined, in slight abuse of notation, as

$$\pi(f)(\mathbf{x}_1, \ldots, \mathbf{x}_m) := f(\pi(\mathbf{x}_1), \ldots, \pi(\mathbf{x}_m))$$

   for each bitvector function $f \in \mathbb{B}^{({}^n\mathbb{B})^m}$ over bitvector variables $X = \{\mathbf{x}_1, \ldots, \mathbf{x}_m\}$.

9.2 The Permutation Equivalence Problem

First permutation equivalence in a free algebra is defined and examples for some specific algebras of interest are given. Then the general result for permutation equivalence of terms by equivalence under variable renaming is stated. Finally the connection with automorphisms of free algebras is indicated.

Definition 9.3 (Permutation Equivalence)
Let $\mathcal{A} = (A, F)$ be an algebra free in some class of algebras with generating set X. Then two elements $a, b \in A$ are *permutation equivalent* iff there exists a permutation $\pi \in Sym(X)$ such that $a = \pi(b)$.

[1] Since $\mathcal{A}$ is free in X, π extends to an automorphism of $\mathcal{A}$.

[2] For the notation used for bitvector terms and their properties as well as certain algebras over bitvector terms and functions please refer to Chapter 6. The algebra of bitvector terms modulo identities ${}^n\mathcal{T}_{\mathcal{BV}}(X)/_{\approx_E}$ was introduced in Section 6.2.3.

Example 9.4

1. Let $f, g \in \mathbb{B}^{(^n\mathbb{B})^m}$ be two bitvector functions over bitvector variables $X = \{\mathbf{x}_1, \ldots, \mathbf{x}_m\}$. Then the permutation equivalence problem for bitvector functions f and g is the question whether there exists a variable renaming $\pi \in Sym(X)$ such that $f = \pi(g)$.

   Consider for example the bitvector function $f, g : \mathbb{B}^2 \times \mathbb{B}^2 \times \mathbb{B}^2 \to \mathbb{B}^2$ over $\mathbf{x}, \mathbf{y}, \mathbf{z}$ with $f(\mathbf{x}, \mathbf{y}, \mathbf{z}) = (00) * \mathbf{y} + (\neg(00)) * \mathbf{z}$ and $g(\mathbf{x}, \mathbf{y}, \mathbf{z}) = (11) * \mathbf{y} + (\neg(11)) * \mathbf{z}$. (Both functions are independent of $\mathbf{x}$.) They are permutation equivalent with π interchanging $\mathbf{y}$ and $\mathbf{z}$ and fixing $\mathbf{x}$ (written $\pi = (\mathbf{y}\ \mathbf{z})$) as shown below:

$$\begin{aligned} f(\mathbf{y}, \mathbf{z}) &= (00) * \mathbf{y} + (\neg(00)) * \mathbf{z} \\ &= (00) * \mathbf{y} + (11) * \mathbf{z} \\ &= (11) * \mathbf{z} + (00) * \mathbf{y} \\ &= (11) * \mathbf{z} + (\neg(11) * \mathbf{y}) \\ &= (11) * \pi(\mathbf{y}) + (\neg(11)) * \pi(\mathbf{z}) \\ &= \pi(g)(\mathbf{y}, \mathbf{z}) \end{aligned}$$

2. Two bitvector terms $s, t \in T_{\mathcal{BV}}(X)$ are permutation equivalent in the algebra of bitvector terms ${}^n\mathcal{T}_{\mathcal{BV}}(X)$ iff $\exists \pi \in Sym(X) : s \equiv t\pi$

3. Given a set of variables X, a signature specification Σ and a set of identities E, the Σ-term algebra modulo E is free in the class of Σ-algebras modeling E with generating set X. It has the E-equivalence classes of terms as carrier. Since it is free with generating set X any variable permutation $\pi \in Sym(X)$ extends to an automorphism. Therefore two bitvector terms s and $t \in T_{\mathcal{BV}}(X)$ are permutation equivalent in the algebra of bitvector terms modulo bitvector identities $({}^n\mathcal{T}_{\mathcal{BV}}(X)/_{\approx_{E_{\mathcal{BV}}}})$ iff

$$\exists \pi \in Sym(X) : [s]_{\approx_{E_{\mathcal{BV}}}} \equiv \pi([t]_{\approx_{E_{\mathcal{BV}}}})$$

In general we have the following result stating that two terms are permutation equivalent iff there is a variable renaming which makes them equivalent.

Lemma 9.5

Let $\mathcal{S}$ be a set of sort symbols, and let F be a set of function symbols. Let $\Sigma : F \to \mathcal{S}^+$ be a signature specification and let $X = \dot{\bigcup}_{\alpha \in \mathcal{S}} X_\alpha$ be an $\mathcal{S}$-sorted set of variables. Let $\mathcal{A}$ be a Σ-algebra free in a class of Σ-algebras with generating set X. Let $s, t \in T_\Sigma(X)$ be Σ-terms then s, t are permutation equivalent in $\mathcal{A}$ iff

$$\exists \pi \in Sym(X) : \mathcal{A} \models s = t\pi$$

Example 9.6

1. Two bitvector terms s and $t \in T_{\mathcal{BV}}(X)$ are permutation equivalent in ${}^n\mathcal{T}_{\mathcal{BV}}(X)/_{\approx_{E_{\mathcal{BV}}}}$ iff $\exists \pi \in Sym(X) : {}^n\mathcal{T}_{\mathcal{BV}}(X)/_{\approx_{E_{\mathcal{BV}}}} \models s = t\pi$.

2. Two extended bitvector terms s and t are permutation equivalent in the extended bitvector term algebra $\mathcal{A}_{\mathcal{BV}}(X)$ (See Section 6.3.) iff $\exists \pi \in Sym(X) : \mathcal{A}_{\mathcal{BV}}(X) \models s = t\pi$. Consider for example the extended bitvector terms $s \equiv (00) * \mathbf{y} + (\neg(00)) * \mathbf{z}$ and $t \equiv (11) * \mathbf{y} + (\neg(11)) * \mathbf{z}$, representing the bitvector functions f and g from Example 9.4 above. Then s and t are permutation equivalent.

Let $\mathcal{A} = (A, F)$ be free with generating set $X \subseteq A$ in some class of algebras. The automorphisms $Aut(\mathcal{A})$ of $\mathcal{A}$ form a group under function composition, acting on the set A. The *orbits* of $Aut(\mathcal{A})$ on A define an equivalence relation on A, where two elements a, b of A are in the same orbit iff they are permutation equivalent, i.e. iff $\exists \pi \in Sym(X) : a = \pi(b)$. (Therefore variable renamings π are called symmetries of $\mathcal{A}$, permutation equivalent functions and terms are called symmetrical, and $Aut(\mathcal{A})$ the symmetry group of $\mathcal{A}$.)

For $B \subseteq A$ we define $\pi B := \{\pi a : a \in B\}$ as the image of B under π. Let Π_B be the set of all automorphisms π of $\mathcal{A}$ such that the image of B under π is B. Then the permutations Π_B form a sub group of $Aut(\mathcal{A})$.

Example 9.7
Consider the bitvector functions f, g from Example 9.4 and $h : \mathbb{B}^2 \times \mathbb{B}^2 \times \mathbb{B}^2 \to \mathbb{B}^2$ over $\mathbf{x}$, $\mathbf{y}$ and $\mathbf{z}$, with $h(\mathbf{x}, \mathbf{y}, \mathbf{z}) := (00) * \mathbf{x} + (\neg(00)) * \mathbf{y}$ then $f = \pi_1 g$ with $\pi_1 = (\mathbf{y}\ \mathbf{z})$, $f = \pi_2 h$ with $\pi_2 = (\mathbf{y}\ \mathbf{x}\ \mathbf{z})$, and $g = \pi_3 h$ with $\pi_3 = (\mathbf{z}\ \mathbf{x}\ \mathbf{y})$. The generated group $< \pi_1, \pi_2, \pi_3 >$ equals the full symmetric group on 3 elements which is of order 6. (The group generated by π_2 is of order 3, and does not contain π_1, which generates a group of order 2.)

9.3 Decidability and Complexity

Often the permutation group $Aut(\mathcal{A})$ is not of interest in itself but only the equivalence relation on a (finite) subset of A (induced by the orbits[3] of $Aut(\mathcal{A})$ on A) has to be found, as it is sufficient to decide whether two terms are permutation equivalent.[4]

In principle, the decidability of permutation equivalence can be reduced to the decidability of equivalence.

For finite X there are only finitely many permutations $\pi \in Sym(X)$. Then we can decide whether two elements $a, b \in A$ are permutation equivalent in $\mathcal{A}$ by checking for each permutation $\pi \in Sym(X)$ whether $a = \pi(b)$ holds, provided that this is a decidable problem for $\mathcal{A}$.

Example 9.8
The permutation equivalence problem for two (extended) bitvector terms $s, t \in T_{\mathcal{BV}}(X)$ (or respective bitvector functions) is decidable, since X is finite and there are decision procedures for the equivalence problem of two terms (s and $t\pi$) as shown before.

Testing for all permutations π of the variables in two bitvector terms s and t whether $\mathcal{A}_{\mathcal{BV}} \models s = t\pi$ holds decides permutational equivalence for bitvector terms by virtue of the following simple algorithm. Let '`Boolean equivalent(Term s, Term t)`' be an algorithm that decides whether $s = t$ holds in $\mathcal{A}_{\mathcal{BV}}$. Let $\mathtt{s} \equiv s$, and $\mathtt{t} \equiv t$, then Algorithm[5] 2 is a decision procedure for the permutation equivalence problem for s and t. Using `symmetrical` in place of `equiv` in algorithm `equivrel` (Algorithm 18, Pg. 199) the latter computes the equivalence relation on a set of terms induced by permutation equivalence.

[3]See above.

[4]For deciding whether there exists an automorphism mapping an element $a \in A$ onto $b \in A$, it is sufficient to know whether they are in the same orbit of $Aut(\mathcal{A})$. Then the automorphism itself is of no interest.

[5]Algorithm 2 is not intended for practical use, but to show that it is possible to determine whether two terms are permutation equivalent. Later in this chapter better algorithms will be developed.

Algorithm 2 Symmetry of Two Terms

```
Boolean symmetrical(Term s, Term t)
{
  Set<Variable> V = Union(Var(s),Var(t));
  foreach Permutation pi in Sym(V)
  {
    if (equivalent(s, apply(t,pi)))
      return true;
  }
  return false;
}
```

Since these decision procedures are Co-NP-complete, considering all $|X|!$ variable renamings is likely to be prohibitively expensive for most problems. In general the problem is computationally hard, since each bitvector term is a Boolean term and the Boolean isomorphism problem is Co-NP-complete (see e.g. [BRS98]). Therefore we will develop sufficient criteria for permutation equivalence of bitvector terms which are easier to check.

9.4 Sufficient Criteria for Permutation Equivalence

Let in the following $\Sigma : F \to \mathcal{S}^+$ be some fixed (many-sorted) signature specification, let X be a finite set of $\mathcal{S}$-sorted variables, let s, t be Σ-terms over X, and let $\mathcal{A}$ be a (many-sorted) Σ-algebra. We consider the problem whether s, t are permutation equivalent in $\mathcal{A}$.

9.4.1 Equivalence

Obviously it holds that equivalence of terms in the algebra $\mathcal{A}$ implies permutation equivalence where the variable renaming π is the identity, i.e.

$$\mathcal{A} \models s = t \;\Rightarrow\; \exists \pi \in Sym(X) : \mathcal{A} \models s = t\pi$$

Example 9.9
Let s, t be (extended) bitvector terms. Then any semantic or syntactic decision procedure for the bitvector function equivalence problem can be used to decide the equivalence of s and t. For example the terms can be translated into equivalent Boolean terms and checked for equivalence in the Boolean domain.

Since these approaches can be expensive, we prefer to use weaker syntactic equivalences implying $\mathcal{A} \models s = t$. The simplest such syntactic equivalence on terms is identity.

$$s \equiv t \;\Rightarrow\; \mathcal{A} \models s = t$$

However, this equivalence is trivial. Employing 'some more semantics' of $\mathcal{A}$ captured in a set of identities E with $\mathcal{A} \models E$ we get

$$E \vdash s = t \;\Rightarrow\; \mathcal{A} \models s = t$$

Then E induces a congruence relation on terms denoted by $\approx_E$. For certain E a confluent many-sorted term rewrite system can be constructed which can be used to decide whether $E \vdash s = t$ holds. For example constant propagation in bitvector terms is such a rewrite approach (See 7.2.3). (Note that even if it is decidable whether $s = t$ holds in $\mathcal{A}$ the problem is in general undecidable for arbitrary E with $\mathcal{A} \models E$. Therefore some care has to be taken when choosing E.)

9.4.2 Syntactic Symmetry

If two terms are identical after renaming of variables then they are permutation equivalent, i.e.

$$\exists \pi \in Sym(Var(s) \cup Var(t)) : s \equiv t\pi \;\Rightarrow\; \exists \pi \in Sym(X) : \mathcal{A} \models s = t\pi$$

If there is such a variable renaming then s and t are identical except variable names, i.e. they have the same structure. Then s, t are called *syntactically symmetrical*.

Example 9.10 (Syntactically Symmetric Terms)

1. Let s_1, t_1 be bitvector terms with $s_1 \equiv (\mathbf{x}_{[2]} + \mathbf{x}_{[2]}) * (\mathbf{x}_{[2]} - \mathbf{y}_{[2]})$, and $t_1 \equiv (\mathbf{z}_{[2]} + \mathbf{z}_{[2]}) * (\mathbf{z}_{[2]} - \mathbf{x}_{[2]})$. Then $\pi_1 = (\mathbf{z}_{[2]}\ \mathbf{x}_{[2]}\ \mathbf{y}_{[2]})$ is a variable renaming with $s_1 \equiv t_1\pi_1$.

2. Let s_2, t_2 be bitvector terms with $s_2 \equiv (\mathbf{x}_{[2]} < \mathbf{y}_{[2]}) \wedge \mathbf{z}_{[1]}$ and $t_2 \equiv (\mathbf{y}_{[2]} < \mathbf{x}_{[2]}) \wedge \mathbf{w}_{[1]}$ then $\pi_2 = (\mathbf{x}_{[2]}\ \mathbf{y}_{[2]})(\mathbf{w}_{[1]}\ \mathbf{z}_{[1]})$ is a variable renaming with $s_2 \equiv t_2\pi_2$.

Following the notation for E-equivalence, the equivalence relation[6] on terms induced by syntactic symmetry is denoted as $\approx_S$.

Modifying the labels of fully collapsed term graphs[7] for terms s and t such that node labels with variables are replaced with their sorts, the problem of syntactic symmetry is identical to the isomorphism problem for such modified term graphs as shown below.

Note 9.11

Note that an isomorphism ϕ from an LDAG $G' = (N_G, E_G, ln'_G, le'_G)$ to $H' = (N_H, E_H, ln'_H, le'_H)$ is a mapping of the nodes of G' to the nodes of H', and of the edges of G' to the edges of H'. The labeling functions are not involved in this mapping in any regard. They only restrict the set of all isomorphisms between G' and H'. Therefore ϕ is a well defined mapping from $G = (N_G, E_G, ln_G, le_G)$ to $H = (N_H, E_H, ln_H, le_H)$, no matter how their labeling functions are defined. However, it is in general not an isomorphism between G and H. In the following we will consider term graphs $G_s = (N_s, E_s, ln_s, le_s)$ and $G_t = (N_t, E_t, ln_t, le_t)$ for many-sorted terms s and t and construct certain labeling functions ln'_s, le'_s, ln'_t, and le'_t from ln_s, le_s, ln_t, and le_t, respectively, which allow isomorphisms between $G'_s = (N_s, E_s, ln'_s, le'_s)$ and $G'_t = (N_t, E_t, ln'_t, le'_t)$ (not present between G_s and G_t). Since an isomorphisms ϕ with $\phi(G'_t) = G'_s$ is a mapping from G_t to G_s (i.e. $\phi(G_t) = G_s$), it identifies the subterms represented by the nodes N_t and $\phi(N_t) = N_s$ in G_t and G_s, respectively.

[6] It is in fact a congruence relation, however this property is not of interest here.

[7] For an overview on term graphs as labeled acyclic graphs (LDAGs) refer to Section 6.4. Fully collapsed term graphs are defined in 6.33.

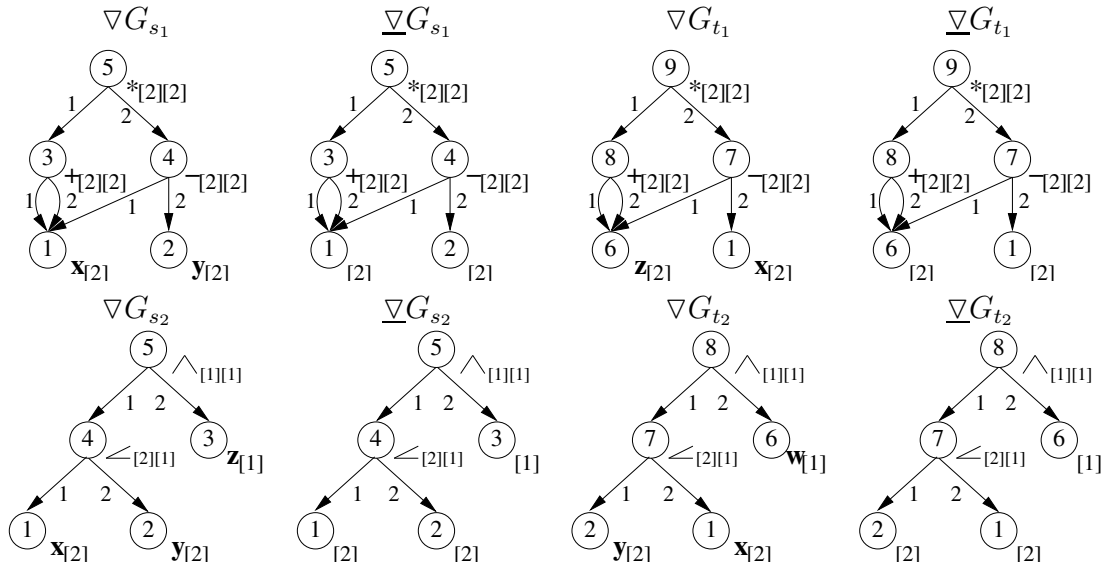


Figure 9.1: Fully Collapsed Term Graphs and their Term Sort Graphs

Definition 9.12 (Term Sort Graph)
Let $\Sigma : F \to \mathcal{S}^+$ be a signature specification and let $X := \dot{\bigcup}_{\alpha \in \mathcal{S}} X_\alpha$ be an $\mathcal{S}$ sorted set of variables, as usual. Let $G = (N, E, ln, le)$ be a term graph with $ln : N \to F \cup X$. Let $ln' : N \to F \cup \mathcal{S}$ be a function labeling nodes either with function symbols or with sorts as below.

$$ln'(n) \quad = \begin{cases} ln(n) & : ln(n) \in F \\ \alpha & : ln(n) \in X_\alpha \end{cases}$$

Then the LDAG $G' := (N, E, ln', le)$ is called *term sort graph* (TSG) of G. For a fully collapsed term graph ∇G its TSG is denoted by $\underline{\nabla} G$.

Note that $\underline{\nabla} G$ results from changing only the node labels of the fully collapsed term graph ∇G.

Example 9.13
Fig. 9.1 shows fully collapsed term graphs (∇G_{s_1}, ∇G_{t_1}, ∇G_{s_2}, and ∇G_{t_2}) and the corresponding TSGs ($\underline{\nabla} G_{s_1}$, $\underline{\nabla} G_{t_1}$, $\underline{\nabla} G_{s_2}$, and $\underline{\nabla} G_{t_2}$) for the BV-terms s_1, t_1, s_2 and t_2 from Example 9.10.

Now we state the important result that two (many-sorted) terms are syntactically symmetrical iff their fully collapsed term sort graphs are isomorphic.

Lemma 9.14 (Syntactic Symmetry by TSG-Isomorphism)
Let s and t be terms and let $X = Var(s) \cup Var(t)$. Let G_s and G_t be (single rooted) term graphs such that $term(G_s) \equiv s$ and $term(G_t) \equiv t$. Let $\nabla G_s = (N_s, E_s, ln_s, le_s)$ and $\nabla G_t = (N_t, E_t, ln_t, le_t)$ be fully collapsed and let $\underline{\nabla} G_s = (N_s, E_s, ln'_s, le_s)$ and $\underline{\nabla} G_t = (N_t, E_t, ln'_t, le_t)$ be the corresponding TSGs. Then s and t are syntactically symmetrical iff $\underline{\nabla} G_s$ and $\underline{\nabla} G_t$ are isomorphic, i.e.

$$\exists \pi \in Sym(X) : s \equiv t\pi \Leftrightarrow \text{ there is an isomorphism } \phi : \underline{\nabla} G_s = \phi(\underline{\nabla} G_t)$$

Proof: (sketch) The application of ϕ onto ∇G_s does not change the labels except for leave nodes. For leave nodes labeled with a variable the sort of the is not changed. This gives rise to a variable permutation constructed from ϕ. Conversely, a variable permutation π extends to bijective mapping of variable nodes of ∇G_t to variable nodes of ∇G_s. This mapping can be lifted to all other nodes of ∇G_t which is an isomorphism between $\underline{\nabla} G_t$ and $\underline{\nabla} G_s$. The full (constructive) proof can be found in Appendix A.3 ■

Example 9.15 (Isomorphic TSGs)
Let s_1, t_1, s_2, t_2 be as in Example 9.10. The TSGs $\underline{\nabla} G_{s_1}$ and $\underline{\nabla} G_{t_1}$ for the terms s_1 and t_1, as well as $\underline{\nabla} G_{s_2}$ and $\underline{\nabla} G_{t_2}$ for s_2 and t_2 shown in Fig. 9.1 and are obviously isomorphic. The mapping $\phi_1 = (1 \mapsto 2, 6 \mapsto 1, 7 \mapsto 4, 8 \mapsto 3, 9 \mapsto 5)$ is an isomorphism mapping $\underline{\nabla} G_{t_1}$ to $\underline{\nabla} G_{s_1}$. The resulting variable renaming is $\pi_1 = (\mathbf{z}_{[2]}\ \mathbf{x}_{[2]}\ \mathbf{y}_{[2]})$. The mapping $\phi_2 = (1 \mapsto 2, 2 \mapsto 1, 6 \mapsto 3, 7 \mapsto 4, 8 \mapsto 5)$ is an isomorphism mapping $\underline{\nabla} G_{t_2}$ to $\underline{\nabla} G_{s_2}$. The resulting variable renaming is $\pi_2 = (\mathbf{x}_{[2]}\ \mathbf{y}_{[2]})(\mathbf{w}_{[1]}\ \mathbf{z}_{[1]})$. Therefore $s_1 \approx_S t_1$, and $s_2 \approx_S t_2$.

9.4.3 Syntactic Symmetry Modulo Identities

Equivalence modulo some set of identities and syntactic symmetries can be combined. Let E be a set of identities. Then two terms s and t are *syntacticly symmetrical modulo* E iff there is a variable renaming π such that $E \vdash s = t\pi$. Let E be a set of identities with $\mathcal{A} \models E$, then we have $E \vdash s = u \Rightarrow \mathcal{A} \models s = u$, hence for all terms s and t:

$$\exists \pi \in Sym(Var(s) \cup Var(t)) : E \vdash s = t\pi \Rightarrow \exists \pi \in Sym(X) : \mathcal{A} \models s = t\pi$$

Following the notation for E-equivalence and syntactic symmetry, the equivalence relation on terms induced by syntactic symmetry modulo E is denoted by $\approx_{ES}$.

Example 9.16 (Syntactic Symmetry Modulo E)

1. Let $s_2 \equiv (\mathbf{x}_{[2]} < \mathbf{y}_{[2]}) \wedge \mathbf{z}_{[1]}$, and let $u_2 \equiv \mathbf{w}_{[1]} \wedge (\mathbf{y}_{[2]} < \mathbf{x}_{[2]})$ be bitvector terms. Let $E_2 = \{(\mathbf{x}_{[2]} \wedge \mathbf{y}_{[2]}) = (\mathbf{y}_{[2]} \wedge \mathbf{x}_{[2]})\}$, then $E_2 \vdash s_2 = u_2\pi_2$ since $E_2 \vdash u_2 = t_2$, $s_2 \equiv t_2\pi_2$ with $\pi_2 = (\mathbf{x}_{[2]}\ \mathbf{y}_{[2]})(\mathbf{w}_{[1]}\ \mathbf{z}_{[1]})$ and $t_2 \equiv (\mathbf{y}_{[2]} < \mathbf{x}_{[2]}) \wedge \mathbf{w}_{[1]}$ as in Example 9.10.

2. Let $k \in \mathbb{N}$ be some upper bound for bitvector widths. Let $E_3 = \bigcup_{i \leq k} \{\mathbf{x}_{[i]} + \mathbf{0}_{[i]} = \mathbf{x}_{[i]}, \mathbf{0}_{[i]} + \mathbf{x}_{[i]} = \mathbf{x}_{[i]}, \mathbf{x}_{[i]} * \mathbf{1}_{[i]} = \mathbf{x}_{[i]}, \mathbf{1}_{[i]} * \mathbf{x}_{[i]} = \mathbf{x}_{[i]}\}$. Let $s_3 \equiv (\mathbf{x}_{[2]} + \mathbf{0}_{[2]}) - (\mathbf{y}_{[2]} * \mathbf{1}_{[2]})$, and let $t_3 \equiv (\mathbf{1}_{[2]} * \mathbf{z}_{[2]}) - (\mathbf{1}_{[2]} * \mathbf{x}_{[2]})$. Then $\pi_3 = (\mathbf{z}_{[2]}\ \mathbf{x}_{[2]}\ \mathbf{y}_{[2]})$ is a variable renaming such that $E_3 \vdash s_3 = t_3\pi_3$.

Because of the general undecidability of E-equivalence for arbitrary E, the undecidability carries over to syntactic symmetry modulo E. Let in the following $\approx_E$ be decidable. Given a convergent rewrite system R derived from E such that $s \downarrow_R \equiv t \downarrow_R$ iff $E \vdash s = t$, then s and t are syntacticly symmetrical modulo E iff their normal forms are syntacticly symmetrical, i.e.

$$\exists \pi \in Sym(Var(s) \cup Var(t)) : s \downarrow_R \equiv t \downarrow_R \pi \Leftrightarrow \exists \pi \in Sym(X) : E \vdash s = t\pi$$

(This is the case since for all $\pi \in Sym(X)$ and convergent rewrite systems R we have $(t \downarrow_R)\pi \equiv (t\pi) \downarrow_R$.) Finally we get the following result.

$$\exists \pi \in Sym(Var(s) \cup Var(t)) : s \downarrow_R \equiv t \downarrow_R \pi \Rightarrow \exists \pi \in Sym(X) : \mathcal{A} \models s = t\pi$$

Now first computing normal forms for s and t and, secondly, checking whether they are syntactically symmetrical gives us a decision procedure for syntactic symmetry modulo some set of identities (provided that an appropriate convergent rewrite system is available.)

Example 9.17 (Syntactic Symmetry after Rewriting)
$R_3 = \bigcup_i \{\mathbf{x}_{[i]} + \mathbf{0}_{[i]} \to \mathbf{x}_{[i]}, \mathbf{0}_{[i]} + \mathbf{x}_{[i]} \to \mathbf{x}_{[i]}, \mathbf{x}_{[i]} * \mathbf{1}_{[i]} \to \mathbf{x}_{[i]}, \mathbf{1}_{[i]} * \mathbf{x}_{[i]} \to \mathbf{x}_{[i]}\}$. Let s_3 and t_3 be as above. Then $s_3 \downarrow_{R_3} \equiv \mathbf{x}_{[2]} - \mathbf{y}_{[2]}$, and $t_3 \downarrow_{R_3} \equiv \mathbf{z}_{[2]} - \mathbf{x}_{[2]}$. Obviously, $s_3 \downarrow_{R_3} \equiv t_3 \downarrow_{R_3} \pi_3$ with π_3 as above. (Note that $(t_3\pi_3) \downarrow_{R_3} \equiv ((1 * \mathbf{x}_{[2]}) - (\mathbf{1}_{[2]} * \mathbf{y}_{[2]})) \downarrow_{R_3} \equiv \mathbf{x}_{[2]} - \mathbf{y}_{[2]} \equiv t_3 \downarrow_{R_3} \pi_3$.)

Note that there is no convergent rewrite system for E_2 from above. Therefore this approach does not work in this case. But it is covered by syntactic symmetry described next.

9.4.4 Syntactic Symmetry Modulo Commutativity

Let f be an n-ary function symbol, let $x_1, \ldots, x_n$ be variables compatible with the signature of f, and let $\pi \in Sym(x_1, \ldots, x_n)$ be a variable renaming, then $f(x_1, \ldots, x_n) = f(\pi x_1, \ldots, \pi x_n)$ is a *(generalized) commutativity identity.* Let E_C be a set of generalized commutativity identities defined by certain $\pi \in Sym(x_1, \ldots, x_n)$ such that $f(x_1, \ldots, x_n) = f(\pi x_1, \ldots, \pi x_n)$ is in E_C. Then the equational theory induced by E_C is a *(generalized) commutative theory.* The resulting congruence relation on terms is denoted by $\approx_{E_C}$. Note that for binary operations the situation is covered by one identity for each operation describing its commutativity.

Example 9.18 (Commutativity Identities)

1. Let $F = \{\wedge, \vee, \neg, 0, 1\}$ be Boolean function symbols with their usual signature as specified in Example 2.3, Pg. 15[8]. A set of commutativity identities for a Boolean algebra is then $E_{C,\mathbf{B}} = \{x \wedge y = y \wedge x, x \vee y = y \vee x\}$.

2. For common and-inverter structures the functions symbols $F' = \{\wedge, \neg, 0\} \subset F_{\mathbf{B}}$ are used, and the set of commutativity identities is $E'_{C,\mathbf{B}} = \{x \wedge y = y \wedge x\}$. The problem of finding symmetries in and-inverter graphs has been studied for example in [MHB98].

3. For the extended bitvector signature $\Sigma_{\mathcal{BV}}$ the set of function symbols for commutative operations is very large. The set of commutativity identities contains all identities of the form $\mathbf{x}_{[i]} \circ_{[i][i]} \mathbf{y}_{[i]} = \mathbf{y}_{[i]} \circ_{[i][i]} \mathbf{x}_{[i]}$ where $i \leq k \in \mathbb{N}$ and $\circ_{[i][i]} \in \{\wedge_{[i][i]}, \vee_{[i][i]}, \oplus_{[i][i]}, +_{[i][i]}, *_{[i][i]}, =_{[i][i]}\}$.

4. The bitvector signature can be extended further such that it contains for all $2 \leq m \leq l \in \mathbb{N}$ the m-ary versions of the commutative operations above. I.e. for each binary commutative operation f there are $f^2 := f$, f^3, f^4, $\ldots$, f^l. (Here l is some fixed bound.) Then the identities have the form

$$f_{m*[i][i]}(\mathbf{x}_{[i],1}, \ldots, \mathbf{x}_{[i],m}) = f_{m*[i][i]}(\mathbf{x}_{[i],\pi(1)}, \ldots, \mathbf{x}_{[i],\pi(m)})$$

[8]The signature can be specified in a many-sorted fashion using just one sort (1) as $\Sigma_{\mathbf{B}} = \{\wedge : 1 \times 1 \to 1, \vee : 1 \times 1 \to 1, \neg : 1 \to 1, 0 : 1, 1 : 1\}$. Then it fits nicely into the many-sorted framework.

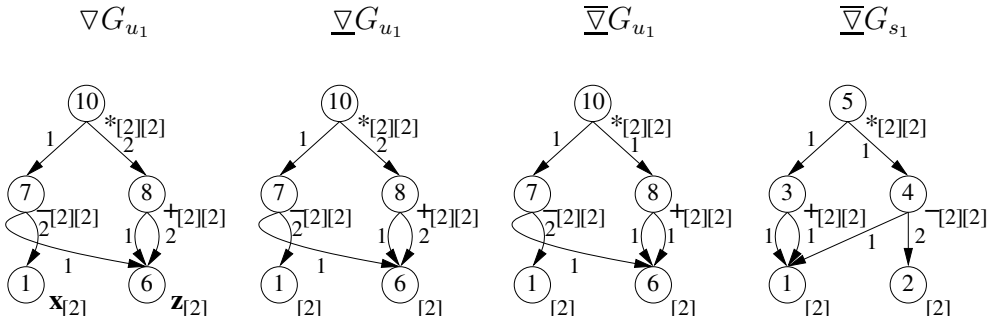


Figure 9.2: Construction of Permutation Term Sort Graph with Isomorphic PTSG

with $i \leq n$, $m \in \mathbb{N}$, $\pi \in Sym(n)$, and $f \in \{\bigwedge, \bigvee, \bigoplus, \sum, \prod, =\}$. For example $\sum_{3*[4][4]}(\mathbf{x}_{[4],1}, \mathbf{x}_{[4],2}, \mathbf{x}_{[4],3}) = \sum_{3*[4][4]}(\mathbf{x}_{[4],2}, \mathbf{x}_{[4],3}, \mathbf{x}_{[3],1})$ describes the identity under one-left rotation of the arguments of a ternary sum of bitvectors of width 4.

Let in the following $\Sigma : F \to \mathcal{S}^+$ be a many-sorted signature specification as usual, and let E_C be a set of commutativity identities such that $\approx_{E_C}$ is a commutative theory. Let $F_C \subseteq F$ denote the set of all function symbols for commutative operations. The equivalence relation on terms describing syntactic symmetry modulo E_C is then denoted by $\approx_{CS}$. The problem of detecting syntactic symmetry of two terms s and t modulo E_C can be reduced to isomorphism of LDAGs.

The idea is to modify the edge labels of TSGs allowing permutations of the successor nodes of nodes labeled with a commutative function symbols. This can be achieved by changing the labels of all outgoing edges to the same value. If the correspondingly modified fully collapsed term graphs for some terms s and t have the same structure modulo permutation of subgraphs, then syntactic symmetry modulo E_C follows. (Therefore this kind of symmetry is also called *structural symmetry.*)

Definition 9.19 (Permutation Term Sort Graph)
Let $\triangledown G = (N, E, ln, le)$ be a fully collapsed term graph for Σ-terms. Let $\underline{\triangledown} G = (N, E, ln', le)$ be the corresponding TSG. Let $le' : E \to \mathbb{N}$ be a function labeling edges with naturals such that:

$$le'(e) \quad = \begin{cases} le(e) & : s(e) \notin F_C \\ 1 & : s(e) \in F_C \end{cases}$$

Then the LDAG$\overline{\underline{\triangledown}} G = (N, E, ln', le')$ is called the *permutation term sort graph* (PTSG) for $\triangledown G$.

Example 9.20 (PTSGs)
Let $s_1 \equiv (\mathbf{x}_{[2]} + \mathbf{x}_{[2]}) * (\mathbf{x}_{[2]} - \mathbf{y}_{[2]})$, and $u_1 \equiv (\mathbf{z}_{[2]} - \mathbf{x}_{[2]}) * (\mathbf{z}_{[2]} + \mathbf{z}_{[2]})$. Let $+$ and $*$ be commutative operations, but not $-$. (I.e. $F_C = \bigcup_{i \leq k}\{+_{[i][i]}, *_{[i][i]}\}$). The PTSGs $\overline{\underline{\triangledown}} G_{s_1}$ and $\overline{\underline{\triangledown}} G_{u_1}$ are shown in Fig. 9.2 together with the previous steps $\triangledown G_{u_1}$ and $\underline{\triangledown} G_{u_1}$.

We show now that isomorphism of PTSGs implies syntactic symmetry modulo commutativity (structural symmetry). This result has direct applications in the reduction of RTL-BMC-problems.

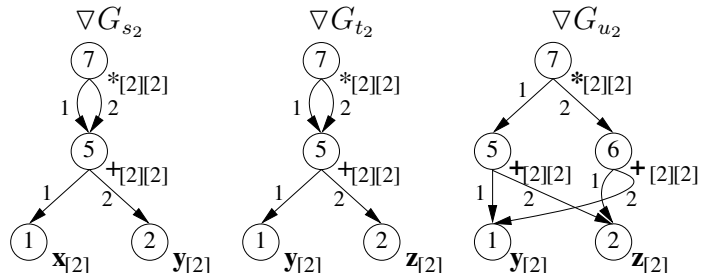


Figure 9.3: Different and Same Structures of Underlying DAGs

Lemma 9.21
Let Σ, E_C, $\approx_{E_C}$ and F_C be as before. Let s, t be Σ terms, and let $X = Var(s) \cup Var(t)$. Let G_s and G_t be (single rooted) term graphs such that $term(G_s) \equiv s$ and $term(G_t) \equiv t$. Let $\nabla G_s = (N_s, E_s, ln_s, le_s)$ and $\nabla G_t = (N_t, E_t, ln_t, le_t)$ be fully collapsed and let $\underline{\overline{\nabla}} G_s = (N_s, E_s, ln'_s, le'_s)$ and $\underline{\overline{\nabla}} G_t = (N_t, E_t, ln'_t, le'_t)$ be the corresponding PTSGs.

$$\exists \phi : \phi(\underline{\overline{\nabla}} G_t) = \underline{\overline{\nabla}} G_s \Rightarrow \exists \pi \in Sym(X) : E_C \vdash s = t\pi$$

Proof: See Appendix A.4. ■

The converse of the lemma above does not hold in general, as shown in the example below. Hence checking PTSGs for isomorphism does not reveal all symmetries (permutation equivalences).

Example 9.22 (Non-Isomorphic PTSGs of CS-equivalent Terms)
Consider the terms $s_2 \equiv (\mathbf{x}_{[2]} + \mathbf{y}_{[2]}) * (\mathbf{x}_{[2]} + \mathbf{y}_{[2]})$, $t_2 \equiv (\mathbf{y}_{[2]} + \mathbf{z}_{[2]}) * (\mathbf{y}_{[2]} + \mathbf{z}_{[2]})$, and $u_2 \equiv (\mathbf{y}_{[2]} + \mathbf{z}_{[2]}) * (\mathbf{z}_{[2]} + \mathbf{y}_{[2]})$. Then the underlying DAGs of their fully collapsed term graphs ∇G_{s_2} and ∇G_{t_2} shown in Fig. 9.3 have the same structure but it is different from the structure of ∇G_{u_2}, respectively. Therefore the resulting PTSGs $\underline{\overline{\nabla}} G_{s_2}$ and $\underline{\overline{\nabla}} G_{u_2}$ (not shown) may not be isomorphic. Note that the PTSGs $\underline{\overline{\nabla}} G_{s_2}$ and $\underline{\overline{\nabla}} G_{t_2}$ are isomorphic.

Improvements by Ordered Rewriting

This problem can be solved partially by normalizing the terms s_2 and u_2 using ordered rewriting as follows. Let $>$ be a total order on variables and function symbols, such that in this order variables are smaller than all function symbols. Then the lexicographic path ordering (See 3.19) can be extended to a total order on terms which is a rewrite order if variables are treated as new free constants.

Let E_C be a set of commutativity identities as usual, and let R_C be a set of rewrite rules derived from E_C by directing the identities from left to right. If $f \in F_C$ is an n-ary commutative function symbol and there is a rule $l \to r$ with $l \equiv f(x_1, \ldots, x_n)$ and $r \equiv f(\pi(x_1), \ldots, \pi(x_n))$, its application is restricted to cases where it decreases the size of the term w.r.t. $>$, the (extended) lexicographic path ordering. The resulting system is terminating and confluent. (See 3.2.7.) (It can be implemented by sorting the arguments of function symbols along $>$.)

Example 9.23 (Isomorphic TSGs after Ordered Rewriting)
Let E_C be the set of commutativity identities defined for the bitvector signature. Let the total order on variables be the alphabetical order. Then the rewrite scheme above reduces

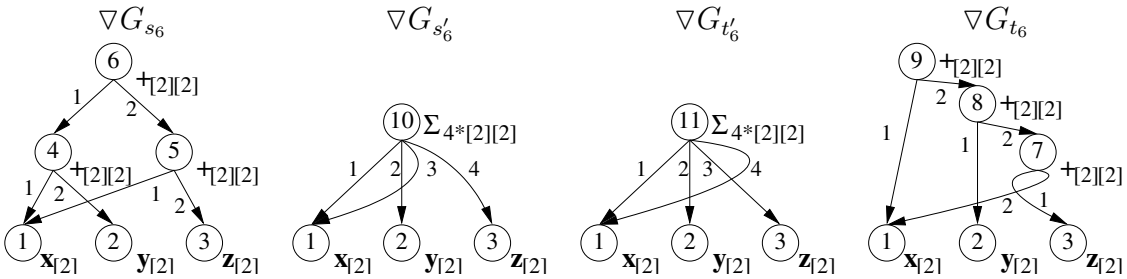


Figure 9.4: Term Graphs with Isomorphic PTSGs after Pruning

$u_2 \equiv (\mathbf{y}_{[2]} + \mathbf{z}_{[2]}) * (\mathbf{z}_{[2]} + \mathbf{y}_{[2]})$ to $t_2 \equiv (\mathbf{y}_{[2]} + \mathbf{z}_{[2]}) * (\mathbf{y}_{[2]} + \mathbf{z}_{[2]})$. (The fully collapsed term graphs of t_2 and s_2 then have the same structure as can be observed in Fig. 9.3.)

9.4.5 Syntactic Symmetry Modulo Associativity and Commutativity

Finally we consider algebras involving associative-commutative operations. Let E_{AC} be a set of identities describing associativity and commutativity of some binary operations. Then the question whether for two terms s and t there is a variable renaming π such that $s \approx_{E_{AC}} t\pi$ can also be translated to the LDAGisomorphism problem. (The induced equivalence relation on terms induced by syntactic symmetry modulo E_{AC} is then denoted as $\approx_{ACS}$.)

Let F_{AC} be a set of binary function symbols for AC operations. Enriching the signature by new commutative function symbols and rewriting terms described below, the problem reduces to structural symmetry described above. The idea is to collect trees of applications of $f \in F_{AC}$ under a single function application of appropriate arity. Then the respective PTSGs can be checked for isomorphism.

Example 9.24
For example $+_{[2][2]}$ is a binary AC-function symbol. Let $s_6 \equiv (\mathbf{x}+\mathbf{y})+(\mathbf{z}+\mathbf{x})$, $t_6 \equiv \mathbf{x}+(\mathbf{y}+(\mathbf{x}+\mathbf{z}))$. Then s and t can be collected under a sum Σ such that $s'_6 \equiv \sum(\mathbf{x}, \mathbf{y}, \mathbf{z}, \mathbf{x})$ and $t'_6 \equiv \sum(\mathbf{x}, \mathbf{y}, \mathbf{x}, \mathbf{z})$. Now s' and t' are structurally symmetrical, i.e. permutation equivalent modulo full commutativity of Σ which can in this case be detected by isomorphism of their PTSGs. The fully collapsed term graphs for s_6, t_6, s'_6 and t'_6 are shown in Fig. 9.4.

This idea is now formalized. Let $f \in F_{AC}$ be a binary AC-function symbol with the signature $f : \alpha \times \alpha \to \alpha$. Let $l \in \mathbb{N}$ be the maximum number of nested occurrences of f in s and t. (For a finite number of terms this number is finite and computable since terms are finite by definition.) For all m with $1 < m \leq l$ let f_m with $f_m : \alpha \times \cdots \times \alpha \to \alpha$ be new m-ary commutative function symbols. Let $f_2(x, y) = f(x, y)$, and let all other relations between these function symbols be described by identities $e_{f,m,k,i}$ of the form

$$\begin{aligned} e_{f,m,k,i} &:= \\ (f_m(x_1, \ldots, x_{i-1}, f_k(y_1, \ldots, y_k), x_{i+1}, \ldots, x_m) &= \\ f_{m+k}(x_1, \ldots, x_{i-1}, y_1, \ldots, y_k, x_{i+1}, \ldots, x_m)) & \end{aligned}$$

with $1 < k \leq l - m$ and $1 \leq i \leq m$. For f we obtain the following set of identities

$$E_f := \bigcup_{1<m\leq l} \bigcup_{1<k\leq l-m} \bigcup_{1\leq i\leq m} e_{f,m,k,i}$$

Then the semantics the commutative operations f_m is completely specified by the semantics of f and the set of identities E_f. This means that the commutativity of f extends to all f_m and for all l $(1 < m \leq l)$ and $\pi \in Sym(m)$ we have

$$E'_f := \bigcup_{1<m\leq l} \bigcup_{\pi\in Sym(m)} \big(f_m(x_1, \ldots, x_m) = f_m(x_{\pi(1)}, \ldots, x_{\pi(m)})\big)$$

Let $E_A = \bigcup_{f\in F_{AC}} E_f$ and $E_C = \bigcup_{f\in F_{AC}} E'_f$. Directing the identities in E_A from left to right leads to a set of rewrite rules which constitutes a convergent rewrite system R_A. Directing the identities E_C from left to right yields R_C which is convergent for ordered rewriting (assuming an appropriate order on terms as described above).

Now applying R_A to terms leads exactly to the situation described for commutativity before. In general we have the following result for terms s and t.

$$s \downarrow_{R_A} \approx_{E_C} \pi(t \downarrow_{R_A}) \Rightarrow s \approx_{E_{AC}} t\pi$$

Example 9.25

1. Consider the terms s_5 and t_5 below.

$$s_5 \equiv \mathbf{x}_{[2]} + \big((\mathbf{x}_{[2]} - (-\mathbf{y}_{[2]})) + (-\mathbf{y}_{[2]})\big)$$

$$t_5 \equiv \big(\mathbf{y}_{[2]} + (\mathbf{y}_{[2]} - (-\mathbf{z}_{[2]}))\big) + (-\mathbf{y}_{[2]})$$

The PTSGsof their fully collapsed term graphs cannot be isomorphic (since the ranks of the target nodes of edges starting in the root nodes are different Fig. 9.5). The only binary AC-function symbol is $+$ and $l = 3$. Hence we define $R_A = \{(x+(y+z)) \to \sum_3(x, y, z), ((y+z)+x) \to \sum_3(y, z, x)\}$. Applying R_A as described above to both terms yields s'_5 and t'_5 below as normal forms.

$$s'_5 \equiv \sum_3 \big(\mathbf{x}_{[2]},\ \mathbf{x}_{[2]} - (-\mathbf{y}_{[2]}),\ -\mathbf{y}_{[2]}\big)$$

$$t'_5 \equiv \sum_3 \big(\mathbf{y}_{[2]},\ \mathbf{y}_{[2]} - (-\mathbf{z}_{[2]}),\ -\mathbf{z}_{[2]}\big)$$

The TSGs (and thus the PTSGs) of their fully collapsed term graphs are isomorphic. The situation is depicted in Fig. 9.5.

2. As shown in Example 9.18 for the extended bitvector signature $\Sigma_{\mathcal{BV}}$ the set of function symbols for commutative operations is very large. All of them are associative as well. The extension to m-ary commutative operations is also shown in this example. The example above is then subsumed by this general case.

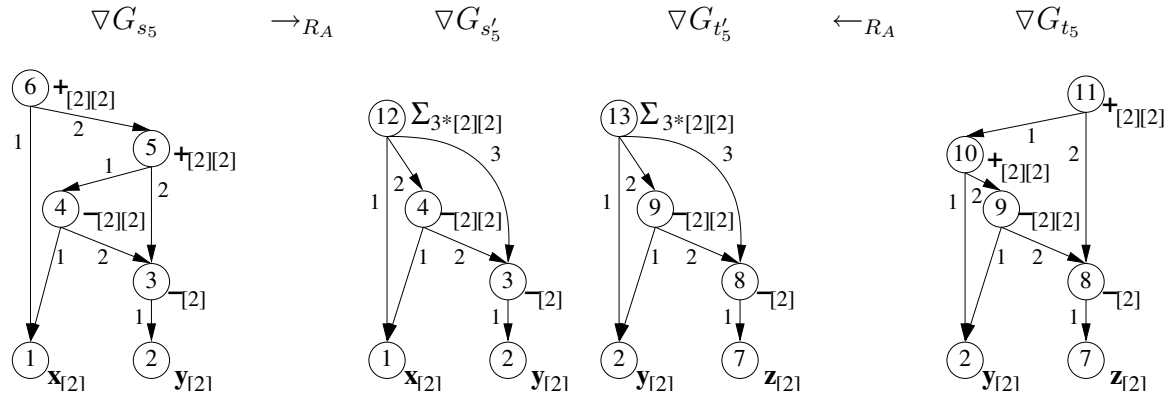


Figure 9.5: Term Graphs with Isomorphic PTSGs after AC-Rewriting

9.5 Sets of Terms and LDAG-Automorphisms

An important extension of the isomorphism approach is to consider sets of terms whose term graphs are shared instead of individual pairs of terms. This allows to consider them at once and reduces the space required to store these term graphs.

Given a set of terms the permutation equivalence relation can be approximated by either of the equivalence relations $\approx_S$, and $\approx_{CS}$ described before. They can be computed by pairwise finding isomorphisms between the respective LDAGs for two terms.

Example 9.26 (Computing Syntactic Symmetry for a Set of Terms)
Using isomorphism of TSGs the relation $\approx_S$ on a set of terms T can be computed as follows. Let `LDAG` denote an object representing an LDAG. Let `LDAG tg(Term t)` construct a term graph for a term, let `LDAG fc(LDAG g)`, and `LDAG tsg(LDAG g)` perform collapsing and TSG computation for a given term graph, respectively. Let `Boolean isomorphic(LDAG g1, g2)` return `true` iff the LDAGs `g1` and `g2` are isomorphic. Then `Boolean equiv(Term t1, t2)` below returns `true` iff `t1` and `t2` are syntactically symmetric.

```
Boolean equiv(Term t1, t2)
{
  return isomorphic(tsg(fc(tg(t1))),tsg(fc(tg(t1))));
}
```

Replacing `<T>` with `Term` in Algorithm 18, Pg. 199 and using `Boolean equiv(Term t1, t2)` as above, Algorithm 18 computes the relation $\approx_S$ for a set of terms.

Instead of considering pairs of terms and LDAGs, we want to consider a single LDAG and its automorphisms. This can reduce the complexity since nodes belonging to different terms may be shared.

First the connection of sets of isomorphisms of LDAGs with automorphisms of algebras is established, then automorphisms of LDAGs are considered. Finally the automorphism problem for LDAGs is transformed into the automorphism problem for colored undirected graphs, for which existing algorithms can be used.

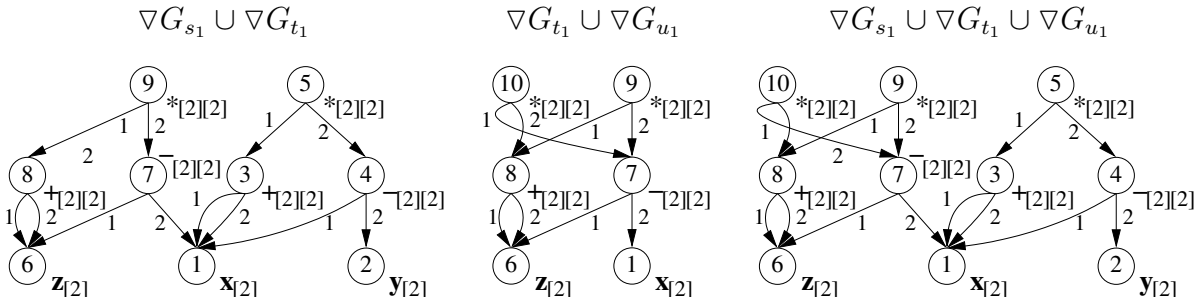


Figure 9.6: Merged LDAGs

9.5.1 Relation between LDAG-Isomorphisms and Automorphisms of Merged LDAGs

Two DAGs (N_1, E_1) and (N_2, E_2) are called *compatible* if $\forall e \in E_1 \cap E_2 : s_1(e) = s_2(e) \wedge t_2(e) = t_2(e)$, i.e. source node and target node of an edge in $E_1 \cap E_2$ must be in $N_1 \cap N_2$. Then the union DAG $(N_1 \cup N_2, E_1 \cup E_2)$ is well defined and is called a *merged* DAG. Merging a set of compatible DAGs again yields a DAG. (Note that DAGs with disjoint nodes are always compatible.)

Definition 9.27 (Compatible LDAGs)
The LDAGs $G_1 = (N_1, E_1, ln_1, le_1)$ and $G_2 = (N_2, E_2, ln_2, le_2)$, are *compatible* iff the underlying DAGs (N_1, E_1) and (N_2, E_2) are compatible, and they have compatible labels, i.e. $\forall n \in N_1 \cap N_2 \neq \emptyset : ln_1(n) = ln_2(n)$ and $\forall e \in E_1 \cap E_2 \neq \emptyset : le_1(e) = le_1(e)$.

The *union* $G_1 \cup G_2 := (N_1 \cup N_2, E_1 \cup E_2, ln_1 \cup ln_2, le_1 \cup le_2)$ *of two compatible* LDAG*s* G_1 and G_2 is then well defined, and $G_1 \cup G_2$ is called *merged* LDAG. (Note that if G_1 and G_2 are single rooted LDAGs, $G_1 \cup G_2$ is in general many rooted.) In general, given a set $\{G_1, \ldots, G_n\}$ of pairwise compatible LDAGs, they can be treated as a single LDAG $G = (N, E, ln, le)$, with $N = \bigcup_{i \in \mathbb{N}_n} N_{G_i}$, $E = \bigcup_{i \in \mathbb{N}_n} E_{G_i}$, $ln = \bigcup_{i \in \mathbb{N}_n} ln_{G_i}$, and $le = \bigcup_{i \in \mathbb{N}_n} le_{G_i}$.

Example 9.28 (Merging Compatible LDAGs)
The LDAGs ∇G_{s_1}, ∇G_{t_1} and ∇G_{u_1} shown in Fig. 9.1 and Fig. 9.2 are pairwise compatible. The merged LDAGs $\nabla G_{s_1} \cup \nabla G_{t_1}$, $\nabla G_{t_1} \cup \nabla G_{u_1}$, and $\nabla G_{s_1} \cup \nabla G_{t_1} \cup \nabla G_{u_1}$ are shown in Fig. 9.6. The only merged node of $\nabla G_{s_1} \cup \nabla G_{t_1}$ is node 1 and there are no merged edges. The merged nodes of $\nabla G_{t_1} \cup \nabla G_{u_1}$, and $\nabla G_{s_1} \cup \nabla G_{t_1} \cup \nabla G_{u_1}$ are 1, 6, 7, and 8, and the merged edges are the edges starting in 7 and 8.

If two compatible LDAGs G_1 and G_2 are merged and an automorphism of the merged LDAG maps the roots of G_1 and G_2 onto each other then G_1 and G_2 are isomorphic. This is stated below after proving the DAG splitting theorem.

Proposition 9.29 (DAG Splitting Theorem)
Let $G = (N, E)$ be a DAG and $\phi \in Aut(G)$. Let $n \in N$ and let $\phi(n) = m$. Let $G_n = (N_n, E_n)$ and $G_m = (N_m, E_m)$ be the subgraphs of G with root n and m respectively. then $G_n \cong G_m$ and $\phi(G_n) = G_m$.

Proof: By induction over the rank of the node $n \in N$.

IB: If $rk(n) = 0$, obviously G_n is degenerated to a leave node and has no outgoing edges ($\delta(n) = 0$).

IH: Assume claim proven for nodes with rank less than r.

IS: (a) Let $rk(n) = r$. As ϕ is a bijection on N and E, $\delta(n) = \delta(m)$, and there are as many paths starting in n as in m.

(b) ϕ maps paths starting in n to paths starting in $\phi(n) = m$ (in a length preserving fashion). Let $k = \delta(n)$, and let $e_1, \ldots, e_k$ be the edges starting in n with $e_i = (n, n_i)$ $(1 \leq i \leq k)$. Then $\phi(e_1), \ldots, \phi(e_k)$ with $\phi(e_i) = (\phi(n), \phi(n_i))$ are the edges starting in m.

(c) Then $rk(n_i) = rk(\phi(n_i)) < r$ and by induction $G_{n_i} = (N_i, E_i) \cong G_{\phi(n_i)} = (\phi(N_i), \phi(E_i))$ for all target nodes n_i $(1 \leq i \leq k)$ of the edges starting in n.

(d) We have $N_n = \{n\} \cup \bigcup_{1 \leq i \leq k} N_i$ and $E_n = \{e_1, \ldots, e_k\} \cup \bigcup_{1 \leq i \leq k} E_i$. Then $\phi(N_n) = \{m\} \cup \bigcup_{1 \leq i \leq k} \phi(N_i) = N_m$ and $\phi(E_n) = \{\phi(e_1), \ldots, \phi(e_k)\} \cup \bigcup_{1 \leq i \leq k} \phi(E_i) = E_m$. Hence $G_n = (N_n, E_n)$ and $G_m = (N_m, E_m)$ are isomorphic, and by construction $\phi(G_n) = G_m$.

■

Corollary 9.30 (LDAG Splitting Theorem)
Let $G = (N, E, ln, le)$ be an LDAG and $\phi \in Aut(G)$. Let $n \in N$ and let $\phi(m) = m$. Let $G_n = (N_n, E_n, ln_n, le_n)$ and $G_m = (N_m, E_m, ln_m, le_m)$ be the subgraphs of G with root n and m, respectively. Then $G_n \cong G_m$ and $\phi(G_n) = G_m$.

Proof: If ϕ is an automorphism of G then it is an automorphism of the underlying DAG (N, E). By Theorem 9.29 ϕ is an isomorphism between the two underlying DAGs (N_n, E_n) and (N_m, E_m). Since ϕ preserves labels, it is an isomorphism between G_n and G_m with $\phi(G_n) = G_m$. ■

Corollary 9.31
Let $\{G_1, \ldots, G_n\}$ be a set of pairwise compatible single rooted LDAGs with roots $r_1, \ldots, r_n$, respectively. If ϕ is an automorphism of their union LDAG G mapping r_i to r_j for some $1 \leq i, j \leq n$, then G_i and G_j are isomorphic.

The converse of this corollary usually does not hold. However, if ϕ is an isomorphism between two LDAGs G_1 and G_2 with disjoint node sets such that $\phi(G_1) = G_2$, then ϕ is an automorphism of $G_1 \cup G_2$.

Example 9.32 (Automorphisms of Merged LDAGs)
Fig. 9.7 shows the permutation term sort graphs of the merged term graphs $\nabla G_{s_1} \cup \nabla G_{t_1}$, $\nabla G_{t_1} \cup \nabla G_{u_1}$, and $\nabla G_{s_1} \cup \nabla G_{t_1} \cup \nabla G_{u_1}$, which are identical to the unions of the permutation term sort graphs $G_1 = \underline{\overline{\nabla}} G_{s_1} \cup \underline{\overline{\nabla}} G_{t_1}$, $G_2 = \underline{\overline{\nabla}} G_{t_1} \cup \underline{\overline{\nabla}} G_{u_1}$, and $G_3 = \underline{\overline{\nabla}} G_{s_1} \cup \underline{\overline{\nabla}} G_{t_1} \cup \underline{\overline{\nabla}} G_{u_1}$. As can be observed by looking at these pictures there is no automorphism of G_1 mapping node 9 to 5, but there is an automorphism ϕ_2 of G_2 mapping 10 to 9. Note that ϕ_2 extends to an automorphism of G_3, but G_3 has no automorphisms mapping 9 to 5. This gives rise to an isomorphisms mapping $\underline{\overline{\nabla}} G_{u_1}$ to $\underline{\overline{\nabla}} G_{t_1}$, derived from ϕ_2. An isomorphism from $\underline{\overline{\nabla}} G_{s_1}$ to $\underline{\overline{\nabla}} G_{t_1}$ cannot be derived from automorphisms of G_3.

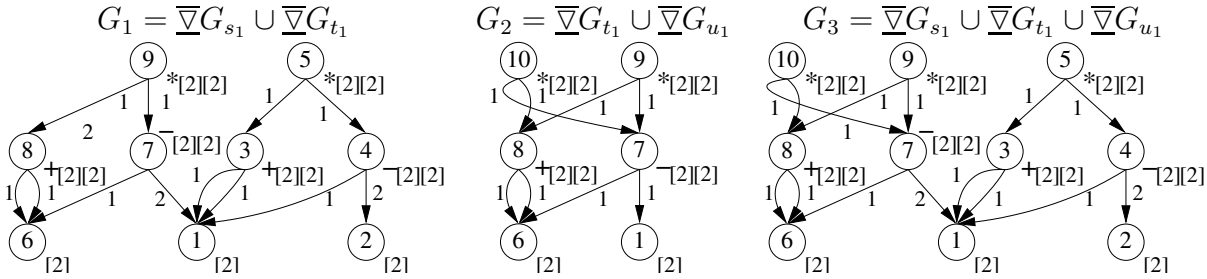


Figure 9.7: Merged LDAGs With and Without Automorphisms

9.5.2 Connection between LDAG-Automorphisms and Permutation Equivalence

Let ∇G be a fully collapsed term graph. It is well known that the set of automorphisms of the LDAG $\underline{\nabla} G$ forms a group under function composition. Its action on the nodes of G defines an equivalence relation on the set of nodes by its orbits. Then two nodes are equivalent iff they are in the same orbit. This equivalence relation translates to the terms represented in ∇G. If two nodes are equivalent, the represented terms are syntactically symmetrical.

The same holds for the automorphisms of the LDAG $\overline{\underline{\nabla}} G$, where, if two nodes are equivalent, the represented terms are syntactically symmetrical modulo the given set of commutativity identities E_C. This is formally stated below.

Theorem 9.33
Let $\nabla G = (N, E, ln, le)$ be a fully collapsed term graph, and let $\sim_1$ and $\sim_2$ be the equivalence relations on nodes induced by the action of the automorphism group of $\underline{\nabla} G$ and $\overline{\underline{\nabla}} G$ (w.r.t. E_C) on N, respectively. Then for all nodes $n, m \in N$, $n \sim_1 m$ implies $term(n) \approx_S term(m)$, and $n \sim_2 m$ implies $term(n) \approx_{CS} term(m)$.

Proof: This is a direct consequence of Lemma 9.14 and Lemma 9.21 and the fact that the orbits of a group define an equivalence relation. ■

Corollary 9.34
Let $T = \{t_1, \ldots, t_n\}$ be a set of terms, and let $G = \bigcup_{1 \leq i \leq n} \nabla G_i$ be the union of a set of pairwise compatible fully collapsed single rooted term graphs with roots $r_1, \ldots, r_n$, respectively such that $t_i \equiv term(\nabla G_i) \in T_\Sigma(X)$ for all $1 \leq i \leq n$. Let $\sim_1$ and $\sim_2$ be as above. Then for all r_i, r_j $(1 \leq i, j \leq n)$, $r_i \sim_1 r_j$ implies $t_i \approx_S t_j$, and $r_i \sim r_j$ implies $t_i \approx_{CS} t_j$.

If the LDAGs $\nabla G_1, \ldots, \nabla G_n$ from above are pairwise disjoint the automorphisms of G define isomorphisms between pairs of LDAGs $\{\nabla G_1, \ldots, \nabla G_n\}$. I.e. if there is an automorphism of G mapping r_i to r_j then there is an isomorphism between ∇G_i and ∇G_j, and conversely if there is such an isomorphism then G has an automorphism mapping r_i to r_j.

Corollary 9.35
Let $T = \{t_1, \ldots, t_n\}$ be a set of terms, and let $\{\nabla G_1, \ldots, \nabla G_n\}$ be a set of fully collapsed single rooted term graphs, such that $t_i \equiv term(\nabla G_i) \in T_\Sigma(X)$ for all $1 \leq i \leq n$.

1. Then the set of isomorphisms Φ between $\{\underline{\triangledown}G_1, \ldots, \underline{\triangledown}G_n\}$ define a set of variable renamings Π. These variable renamings generates a subgroup $\langle\Pi\rangle$ of the automorphism group of $\mathcal{T}_\Sigma(X)$. Then two terms of T are syntactically symmetric iff they are in the same orbit of $\langle\Pi\rangle$ on $T_\Sigma(X)$.

2. Let E_C be a set of commutativity identities such that $\approx_{EC}$ is a commutative theory. Then the set of isomorphisms Φ between $\{\underline{\overline{\triangledown}}G_1, \ldots, \underline{\overline{\triangledown}}G_n\}$ defines a set of bijective mappings Π from terms to terms. These mappings generate a subgroup of the automorphism group of $\mathcal{T}_\Sigma(X)/_{\approx_{EC}}$.

However, we are usually not interested in the automorphism groups of these algebras nor are we interested in the permutation equivalence relation on all terms. Instead, the permutation equivalence relation on a given set of terms is of interest. In case of syntactic symmetry, it can be computed by considering automorphisms of merged LDAGs or by considering isomorphisms of pairs as shown before. Since the latter is identical to considering automorphisms of an LDAG resulting from merging LDAGs with pairwise disjoint node sets, it is sufficient to provide means of computing the automorphism groups of LDAGs. This is done in the next section. In the case of syntactic symmetry modulo identities an approximation can be computed in the same way.

9.5.3 Automorphisms of Colored Undirected Graphs

Here we are concerned with finding generators for the automorphism groups of LDAGs and computing the action of these groups on the respective node sets. LDAGs can be transformed into colored undirected graphs such that their automorphisms coincide. Automorphisms of colored undirected graphs can be computed efficiently using standard algorithms from computation group and graph theory.

Definition 9.36 (CUG)
Let N be a set of nodes, let $E \subseteq N \times N$ be a symmetric irreflexive relation for edges, let C be a set of colors (labels), and let $c : N \to C$ be a function mapping nodes to their colors. Then the tuple $G = (N, E, c)$ is a *colored undirected graph* (CUG).

Note that, in contrast to directed graphs a CUG is always simple, i.e. it does not have multiple edges between two nodes, and it contains no loops, i.e. edges of the form (n, n).

Definition 9.37 (CUG-Morphism)
A CUG-*morphism* between $G_1 = (N_1, E_1, c_1)$ and $G_2 = (N_2, E_2, c_2)$ is a function $\phi : N_1 \to N_2$ that preserves the structure and coloring of G, i.e. $(n_1, m_1) \in E_1$ implies $(\phi(n_1), \phi(m_1)) \in E_2$ and $\forall n_1 \in N_1 : c_1(n_1) = c_2(\phi(n_1))$.

If ϕ is bijective, it is called CUG-*isomorphism*, and G_1 and G_2 are called isomorphic ($G_1 \cong G_2$). An isomorphism of G onto itself is called CUG-*automorphism*.

Definition 9.38 (CUG of an LDAG)
Let $G = (N, E, ln, le)$ be an LDAG, with $ln : N \to L_N$, $le : E \to L_E$. Let L_E and L_N be disjoint. Let N', E', C, and c be defined as below.

1. Let $N' := N \cup E$ be a set of nodes.

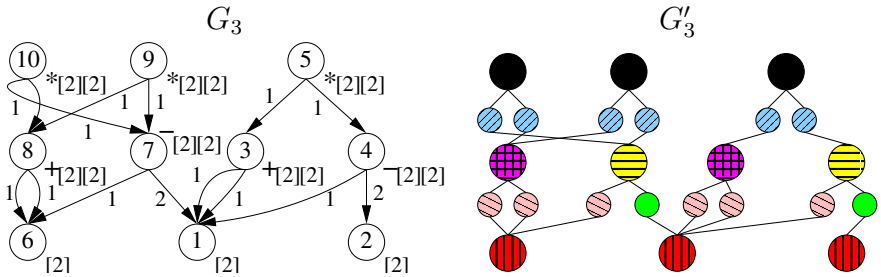


Figure 9.8: LDAG and Corresponding CUG

2. Let $E' := \{(s(e), e) : e \in E\} \cup \{(t(e), e) : e \in E\} \subseteq N \times E$ be a symmetric relation on N' for edges.

3. Let $C := \{0, 1\} \times (L_N \cup L_E) \times \mathbb{N}_0$ be a set of colors.

4. Let $c : N' \to C$ be a coloring function with

$$c(n') = \begin{cases} (0, ln(n'), rk(n)) & : n' \in N \\ (1, le(n'), rk(t(n'))) & : n' \in E \end{cases}$$

Then $G' = (N', E', c)$ is the CUG *for* G.

By these conditions E' is irreflexive since $N \cap E = \emptyset$, and $\forall e \in E \forall n \in N : c(e) \neq c(n)$. Therefore G' is well defined, i.e. it is indeed a CUG.

Example 9.39
Fig. 9.8 shows the LDAG G_3 from Example 9.32 and its CUG.

Lemma 9.40 (LDAG and CUG Automorphisms)
Let $G = (N, E, ln, le)$ be an LDAG and let $G' = (N', E', c)$ with $N' = N \cup E$ be its CUG. Let n, m be two nodes of G then

$$\exists \phi \in Aut(G) : \phi(n) = m \Leftrightarrow \exists \phi' \in Aut(G') : \phi'(n) = m$$

Proof: (sketch) For the extension of ϕ to edges of G ($\varphi : E \to E$), and the extension of ϕ' to the edges of G' ($\varphi' : E' \to E'$) we define the relation of $\phi' \in Aut(G')$ and $\phi \in Aut(G)$ as follows:

$$\phi'(n') = \begin{cases} \phi(n') & : n' \in N \\ \varphi(n') & : n' \in E \end{cases}$$

Then we show that given one mapping and defining the other in terms of it as above, they are well defined and that they are automorphisms, respectively. For the full proof see Appendix A.5. ■

The complexity class of finding automorphisms of colored undirected graphs is not precisely known, but is believed to be somewhere between NP and P. The automorphism groups of CUGs may be large. Generators are often a much more compact representation of a group than the set of its members. Practically generators for automorphism groups of

CUGs can be computed for example using algorithms presented by McKay [McK81]. Given a set of generators of the automorphism group the orbits can be computed efficiently. The orbits of an automorphism group on the nodes of a CUG coincide with the orbits of the automorphism group on the nodes of the respective LDAG. These orbits give rise to permutation equivalence of many-sorted terms.

Chapter 10

Symmetry Reduction of Bitvector Terms

This chapter explains how symmetrical values in bitvector terms for RTL-BMC-problems can be exploited to reduce the size of these terms, and in particular how the number of variables involved can be reduced dramatically. This reduction is very effective for bounded interval model checking regular designs on the register transfer level. First, the concepts of symmetrical values and cofactor expansion are extended from Boolean functions to bitvector functions. This extension will lead to symmetry relations which provide the framework for dealing with symmetrical values computationally. Then a basic reduction scheme is provided, considering symmetrical values in one variable at a time. This reduction scheme is then extended to symmetrical value vectors. Finally it is shown how symmetry relations on different variables are combined, and further extensions based on type transformations of bitvector functions are given.

10.1 Symmetrical Values in Bitvector Functions

In order to extend the concept of symmetry reduction to bitvector terms the concept of symmetrical values is extended from Boolean functions, as described in Chapter 5, to functions with finite domains and Co-domains.

First, equivalent values and equivalent value vectors are extended to bitvector functions. Then the concepts of symmetrical values and symmetrical value vectors are translated into this context. Symmetry relations and extended symmetry relations as equivalence relations on values are introduced allowing to capture complex relationships between symmetrical values and symmetrical value vectors. Finally cofactor expansion of bitvector functions and bitvector terms are considered as a basis for extending the symmetry reduction scheme from Boolean functions to bitvector functions and their representations.

10.1.1 Equivalent Bitvector Values

First we extend the notion of equivalent values from Boolean functions to bitvector functions.

Definition 10.1 (Equivalent Bitvector Values)
Let $f : \mathbb{B}^{n_1} \times \ldots \times \mathbb{B}^{n_m} \to \mathbb{B}^{n_f}$ be a bitvector function in bitvector variables $\mathbf{x}_1, \ldots, \mathbf{x}_m$. For some $i \leq m$ let $\mathbf{a}, \mathbf{b} \in \mathbb{B}^{n_i}$. If for all $\mathbf{x}_1 \in \mathbb{B}^{n_1}, \ldots, \mathbf{x}_{i-1} \in \mathbb{B}^{n_{i-1}}, \mathbf{x}_{i+1} \in \mathbb{B}^{n_{i+1}}, \ldots, \mathbf{x}_m \in \mathbb{B}^{n_m}$

$$f(\mathbf{x}_1, \mathbf{x}_2, \ldots, \mathbf{x}_{i-1}, \mathbf{a}, \mathbf{x}_{i+1}, \ldots, \mathbf{x}_m) = f(\mathbf{x}_1, \mathbf{x}_2, \ldots, \mathbf{x}_{i-1}, \mathbf{b}, \mathbf{x}_{i+1}, \ldots, \mathbf{x}_m)$$

holds we call $\mathbf{a}$ and $\mathbf{b}$ *equivalent bitvector values* (or equivalent values for short) for $\mathbf{x}_i$ in f.

Example 10.2
Consider the bitvector function $f_1 : \mathbb{B}^2 \times \mathbb{B} \to \mathbb{B}$ over $\mathbf{x}$ and $\mathbf{y}$ given by the truth table below. Then the values 00, 01, and 11 are equivalent values for $\mathbf{x}$ in f_1.

Table 10.1: A Bitvector Function with Equivalent Values for $\mathbf{x}$

$\mathbf{x}$	$\mathbf{y}$	$f_1(\mathbf{x}, \mathbf{y})$
00	0	0
00	1	1
01	0	0
01	1	1
10	0	0
10	1	0
11	0	0
11	1	1

Following the notion for Boolean functions we call

$$f_{\mathbf{x}_i = \mathbf{a}} := f(\mathbf{x}_1, \mathbf{x}_2, \ldots, \mathbf{x}_{i-1}, \mathbf{a}, \mathbf{x}_{i+1}, \ldots, \mathbf{x}_m)$$

the $\mathbf{a}$-*cofactor* of f w.r.t. $\mathbf{x}_i$, where $\mathbf{a} \in \mathbb{B}^{n_i}$. Then $\mathbf{a}$ and $\mathbf{b}$ are equivalent values for $\mathbf{x}$ in f iff $f_{\mathbf{x}_i = \mathbf{a}} = f_{\mathbf{x}_i = \mathbf{b}}$.

Obviously bitvector functions with Boolean codomain can be treated as Boolean functions. Then equivalent values of a bitvector function correspond to equivalent value vectors of the respective Boolean function.

Example 10.3
The bitvector function $f_1 : \mathbb{B}^2 \times \mathbb{B} \to \mathbb{B}$ in bitvector variables $\mathbf{x}$ and $\mathbf{y}$ from the last example can be treated as a Boolean function $f_1 : \mathbb{B}^3 \to \mathbb{B}$ in Boolean variables x_0, x_1 and y. Then the equivalent values 00, 01, and 11 for $\mathbf{x}$ are equivalent value vectors for the vector of variables (x_1, x_0).

However, the converse usually does not hold. This is only the case if $\mathbf{a}$ and $\mathbf{b}$ are equivalent value vectors for some vector of Boolean variables $(x_k, \ldots, x_1)$, and there is a corresponding bitvector variable $\mathbf{x}$ of width k.

As for Boolean functions the concept of equivalent values can be extended to equivalent value vectors.

Definition 10.4 (Equivalent Value Vectors)
Let $f : \mathbb{B}^{n_1} \times \ldots \times \mathbb{B}^{n_m} \rightarrow \mathbb{B}^{n_f}$ be a bitvector function in m bitvector variables X. Let $X_k \subseteq X$ with $X_k = \{\mathbf{x}_1, \ldots, \mathbf{x}_k\}$. Let $(\mathbf{a}_1, \ldots, \mathbf{a}_k)$ and $(\mathbf{b}_1, \ldots, \mathbf{b}_k)$ with $\mathbf{a}_i, \mathbf{b}_i \in Dom(\mathbf{x}_i)$ be vectors of bitvector values for $(\mathbf{x}_1, \ldots, \mathbf{x}_k)$. Then $(\mathbf{a}_1, \ldots, \mathbf{a}_k)$ and $(\mathbf{b}_1, \ldots, \mathbf{b}_k)$ are called *equivalent value vectors* for $(\mathbf{x}_1, \ldots, \mathbf{x}_k)$ w.r.t. f iff

$$f_{\mathbf{x}_1=\mathbf{a}_1 \cdots \mathbf{x}_k=\mathbf{a}_k} = f_{\mathbf{x}_1=\mathbf{b}_1 \cdots \mathbf{x}_k=\mathbf{b}_k}$$

Let φ, φ' be two variable assignments for X_k with $\varphi(\mathbf{x}_i) = \mathbf{a}_i$ and $\varphi'(\mathbf{x}_i) = \mathbf{b}_i$, then φ, φ' are called *equivalent assignments* for X_k w.r.t. f.

By this definition two vectors of values are equivalent iff the corresponding cofactors are equivalent. We will in the following switch between equivalent value vectors, and corresponding assignments, as well as equivalent cofactors w.r.t. these value vectors at need.

For bitvector functions with Boolean codomain, equivalent values for a vector of variables correspond to equivalent value vectors of the respective Boolean function. Again, the converse usually doesn't hold.

10.1.2 Symmetrical Bitvector Values

In the same way as equivalent values have been extended to bitvector functions above, now the notion of symmetrical values is extended to bitvector functions.

Definition 10.5 (Symmetrical Bitvector Values)
Let $f : \mathbb{B}^{n_1} \times \ldots \times \mathbb{B}^{n_m} \rightarrow \mathbb{B}^{n_f}$ be a bitvector function in m bitvector variables $X = \{\mathbf{x}_1, \ldots, \mathbf{x}_m\}$. For some $i \leq m$ let $\mathbf{a}, \mathbf{b} \in \mathbb{B}^{n_i}$ be two possible values for $\mathbf{x}_i$. If there is a variable renaming $\pi \in Sym(X \backslash \{\mathbf{x}_i\})$ such that

$$f_{\mathbf{x}_i=\mathbf{a}} = \pi(f_{\mathbf{x}_i=\mathbf{b}})$$

holds we call $\mathbf{a}$ and $\mathbf{b}$ *symmetrical bitvector values* (or symmetrical values for short) for $\mathbf{x}_i$ in f, written $\mathbf{a} \sim_{f,\mathbf{x}_i} \mathbf{b}$ and $(\mathbf{x}_i \mapsto \mathbf{a})$, $(\mathbf{x}_i \mapsto \mathbf{b})$ are *symmetrical (partial) assignments* for $\mathbf{x}_i$ in f.

Example 10.6
Consider the bitvector function $f_2 : \mathbb{B}^2 \times \mathbb{B}^2 \times \mathbb{B}^2 \rightarrow \mathbb{B}^2$ over $\mathbf{x}$, $\mathbf{y}$, $\mathbf{z}$ with $f_2(\mathbf{x}, \mathbf{y}, \mathbf{z}) = \mathbf{x} * \mathbf{y} + (\neg \mathbf{x}) * \mathbf{z}$. Then the values 00, and 11 are symmetrical values for $\mathbf{x}$ in f_2, since with π interchanging $\mathbf{y}$ and $\mathbf{z}$ we have $f_{2_{\mathbf{x}=(00)}} = \pi(f_{2_{\mathbf{x}=(11)}})$ as shown below:

$$\begin{aligned}
f_{2_{\mathbf{x}=(00)}}(\mathbf{y}, \mathbf{z}) &= (00) * \mathbf{y} + (\neg(00)) * \mathbf{z} \\
&= (00) * \mathbf{y} + (11) * \mathbf{z} \\
&= (11) * \mathbf{z} + (00) * \mathbf{y} \\
&= (11) * \mathbf{z} + (\neg(11) * \mathbf{y} \\
&= (11) * \pi(\mathbf{y}) + (\neg(11)) * \pi(\mathbf{z}) \\
&= \pi(f_{2_{\mathbf{x}=(11)}}(\mathbf{y}, \mathbf{z}))
\end{aligned}$$

The values 01 and 10 are also symmetrical values for $\mathbf{x}$ in f_2, but 00 and 10 are not symmetrical.

Obviously equivalent values are also symmetrical values but in general not vice versa. Considering bitvector functions with Boolean codomain it is easy to see that symmetrical values for some bitvector variable in such a function correspond to symmetrical value vectors of some vector of Boolean variables for the respective Boolean function. As for Boolean functions the concept of symmetrical variables can be extended to vectors of symmetrical variables.

Definition 10.7 (Symmetrical Value Vectors)
Let $f : \mathbb{B}^{n_1} \times \ldots \times \mathbb{B}^{n_m} \rightarrow \mathbb{B}^{n_f}$ be a bitvector function in m bitvector variables X. Let $X_m \subseteq X$ with $X_k = \{\mathbf{x}_1, \ldots, \mathbf{x}_k\}$. Let $(\mathbf{a}_1, \ldots, \mathbf{a}_k)$ and $(\mathbf{b}_1, \ldots, \mathbf{b}_k)$ with $\mathbf{a}_i, \mathbf{b}_i \in Dom(\mathbf{x}_i)$ be vectors of bitvector values for $(\mathbf{x}_1, \ldots, \mathbf{x}_k)$. Then $(\mathbf{a}_1, \ldots, \mathbf{a}_k)$ and $(\mathbf{b}_1, \ldots, \mathbf{b}_k)$ are called *symmetrical value vectors* for $(\mathbf{x}_1, \ldots, \mathbf{x}_k)$ w.r.t. f, written as $(\mathbf{a}_1, \ldots, \mathbf{a}_k) \sim_{f(\mathbf{x}_1,\ldots,\mathbf{x}_k)} (\mathbf{b}_1, \ldots, \mathbf{b}_k)$ iff

$$\exists \pi \in Sym(X \backslash X_k) : f_{\mathbf{x}_1=\mathbf{a}_1 \cdots \mathbf{x}_k=\mathbf{a}_k} = \pi(f_{\mathbf{x}_1=\mathbf{b}_1 \cdots \mathbf{x}_k=\mathbf{b}_k})$$

Let φ, φ' be two variable assignments for X_k with $\varphi(\mathbf{x}_i) = \mathbf{a}_i$ and $\varphi'(\mathbf{x}_i) = \mathbf{b}_i$, then φ, φ' are called *symmetrical assignments* for X_k w.r.t. f.

Again symmetrical value vectors for some bitvector variables in a bitvector function with Boolean codomain imply symmetrical value vectors in Boolean variables of the respective Boolean function, but in general not vice versa.

10.1.3 Symmetry Relations

By the Definition 10.5 $\sim_{f,\mathbf{x}_i}$ is an equivalence relation on $\mathbb{B}^{n_i}$. Conversely, an equivalence relation on $\mathbb{B}^{n_i}$ is called symmetry relation iff all equivalent values are symmetrical, as defined below.

Definition 10.8 (Symmetry Relation)
Let $f : \mathbb{B}^{n_1} \times \ldots \times \mathbb{B}^{n_m} \rightarrow \mathbb{B}^{n_f}$ be a bitvector function in m bitvector variables $X = \{\mathbf{x}_1, \ldots, \mathbf{x}_m\}$. A binary relation $\sim$ on $\mathbb{B}^{n_i}$ is called a *symmetry relation* for $\mathbf{x}_i$ in f iff for all $\mathbf{a}, \mathbf{b} \in \mathbb{B}^{n_i}$, $\mathbf{a} \sim \mathbf{b}$ implies $\mathbf{a} \sim_{f,\mathbf{x}_i} \mathbf{b}$. A symmetry relation $\sim$ is *maximal* iff it is identical to $\sim_{f,\mathbf{x}_i}$, otherwise it is *approximate*.

Obviously each symmetry relation is an equivalence relation. The natural representation of an equivalence relation is a partition. Hence, a symmetry relation $\sim$ for $\mathbf{x}_i$ in f is represented by a partition of $\mathbb{B}^{n_i}$. If all values are pairwise symmetrical for $\mathbf{x}_i$ in f, this is the unit partition $\{\mathbb{B}^{n_i}\}$. Otherwise $\mathbb{B}^{n_i}$ will be partitioned into equivalence classes $S_1, S_2, \ldots, S_k$. In case that a variable does not have symmetrical values, each value forms its own equivalence class, and the partition representing it is the discrete partition. The equivalence relation, which is induced by the partition $S = \{S_1, \ldots, S_k\}$ of $\mathbb{B}^{n_i}$, is also denoted as $\sim_S$. Partitions of variable domains and the symmetry relations induced by them are usually identified with each other, i.e. a partition, which induces a symmetry relation, is the representation of this symmetry relation.

Example 10.9
The maximal symmetry relation for $\mathbf{x}$ in f_1 from Example 10.2 is $\{\{00, 01, 11\}, \{10\}\}$, and $\{\{00, 01\}, \{10\}, \{11\}\}$ is an approximate symmetry relation. The symmetry relation for $\mathbf{x}$ in f_2 from Example 10.6 is $\{\{00, 11\}, \{01, 10\}\}$.

Lemma 10.10
Let $f : \mathbb{B}^{n_1} \times \ldots \times \mathbb{B}^{n_m} \to \mathbb{B}^{n_f}$ be a bitvector function in variables $X = \{\mathbf{x}_1, \ldots, \mathbf{x}_m\}$. Let $\mathbf{x} \in X$. If P, Q are partitions of $Dom(\mathbf{x})$, and $\sim_P$, $\sim_Q$ are symmetry relations for $\mathbf{x}$ in f, then merging all non-disjoint classes of $\sim_P$ and $\sim_Q$ yields a finer approximation of the symmetry relation.

Example 10.11
The partitions $\{\{00, 01\}, \{10\}, \{11\}\}$ and $\{\{00\}, \{01, 11\}, \{10\}\}$ are (approximate) symmetry relations and $\{\{00, 01\}, \{10\}, \{11\}\} \sqcup \{\{00\}, \{01, 11\}, \{10\}\} = \{\{00, 01, 11\}, \{10\}\}$ is a (the maximal) symmetry relation for $\mathbf{x}$ in f_1 from Example 10.2.

If all values for one variable $\mathbf{x}_i$ in f are shown to be pairwise symmetrical, we can conclude that f is independent from $\mathbf{x}_i$, as shown in the following example.

Example 10.12
Consider the bitvector function $f_3 : \mathbb{B}^2 \times \mathbb{B} \to \mathbb{B}$ over $\mathbf{x}$ and $\mathbf{y}$ given by the truth table below. Then the values 00, 01, 10, and 11 are symmetrical values for $\mathbf{x}$ in f_3. Obviously $f_3(\mathbf{x}, \mathbf{y}) = \mathbf{y}$, thus f_3 is independent from $\mathbf{x}$.

Table 10.2: A Bitvector Function with all Values for $\mathbf{x}$ Symmetrical

$\mathbf{x}$	$\mathbf{y}$	$f_3(\mathbf{x}, \mathbf{y})$
00	0	0
00	1	1
01	0	0
01	1	1
10	0	0
10	1	1
11	0	0
11	1	1

If all values of each variable of a function are pairwise symmetrical, then the function is constant as stated formally below.

Lemma 10.13
Let $f : \mathbb{B}^{n_1} \times \ldots \times \mathbb{B}^{n_m} \to \mathbb{B}^{n_f}$ be a bitvector function in variables $\mathbf{x}_1, \ldots, \mathbf{x}_m$. Then f is a constant function iff all values for all variables are pairwise symmetrical.

Proof: If all values for each variable are pairwise symmetrical, f is independent from each variable, and hence constant. Conversely, if f is constant their cannot be a variable $\mathbf{x}$, values $\mathbf{a}$ and $\mathbf{b}$ for $\mathbf{x}$ and a renaming π such that $f_{\mathbf{x}=\mathbf{a}} \neq \pi(f_{\mathbf{x}=\mathbf{b}})$, since then f would not be constant. ■

Now the concept of symmetry relations for one variable is extended to vectors of variables.

Definition 10.14 (Extended Symmetry Relation)
Let $f : \mathbb{B}^{n_1} \times \ldots \times \mathbb{B}^{n_m} \to \mathbb{B}^{n_f}$ be a bitvector function in m bitvector variables X. Let $X_k \subseteq X$ with $X_k = \{\mathbf{x}_1, \ldots, \mathbf{x}_k\}$. A binary relation $\sim$ on $D := Dom(\mathbf{x}_1) \times \cdots \times Dom(\mathbf{x}_k)$ is an *extended symmetry relation* for X_k in f iff for all $(\mathbf{a}_1, \ldots, \mathbf{a}_k), (\mathbf{b}_1, \ldots, \mathbf{b}_k) \in D$, $(\mathbf{a}_1, \ldots, \mathbf{a}_k) \sim (\mathbf{b}_1, \ldots, \mathbf{b}_k)$ implies $(\mathbf{a}_1, \ldots, \mathbf{a}_k) \sim_{f(\mathbf{x}_1, \ldots, \mathbf{x}_k)} (\mathbf{b}_1, \ldots, \mathbf{b}_k)$.

Corollary 10.15
Let $f : \mathbb{B}^{n_1} \times \ldots \times \mathbb{B}^{n_m} \rightarrow \mathbb{B}^{n_f}$ be a bitvector function in m bitvector variables $X = \{\mathbf{x}_1, \ldots, \mathbf{x}_m\}$. All value vectors $(\mathbf{a}_1, \ldots, \mathbf{a}_m), (\mathbf{b}_1, \ldots, \mathbf{b}_m) \in \mathbb{B}^{n_1} \times \ldots \times \mathbb{B}^{n_m}$ are pairwise symmetrical value vectors for $(\mathbf{x}_1, \ldots, \mathbf{x}_m)$ iff f is constant.

Lemma 10.16
Let $f : \mathbb{B}^{n_1} \times \ldots \times \mathbb{B}^{n_m} \rightarrow \mathbb{B}^{n_f}$ be a bitvector function in variables $X = \{\mathbf{x}_1, \ldots, \mathbf{x}_m\}$. Let $X_k \subseteq X$ with $X_k = \{\mathbf{x}_1, \ldots, \mathbf{x}_k\}$, and let $D := Dom(\mathbf{x}_1) \times \cdots \times Dom(\mathbf{x}_k)$. If P, and Q are partitions of D, and $\sim_P$, $\sim_Q$ are extended symmetry relations for X_k in f, then merging the non-disjoint equivalence classes of $\sim_P$ and $\sim_Q$ is an extended symmetry relation for X_k in f which is finer than $\sim_P$, and $\sim_Q$.

10.1.4 Cofactor Expansion for Bitvector Functions

Equivalent and symmetrical assignments to variables in some bitvector function, if known, can be used for preprocessing when deciding the bitvector function equivalence problem as shown below.

The concept of symmetrical cofactor expansion for Boolean functions[1], as described in Chapter 5, is extended to bitvector functions as follows. Let $f : \mathbb{B}^{n_1} \times \ldots \times \mathbb{B}^{n_m} \rightarrow \mathbb{B}$ be a bitvector function with Boolean codomain over m variables X. For $\mathbf{x} \in X$ let $Dom(\mathbf{x}) = \mathbb{B}^n$. Then we have:

$$f = 1 \;\; \Leftrightarrow \;\; f_{\mathbf{x}=(0\cdots00)_{[n]}} = 1 \;\wedge\; f_{\mathbf{x}=(0\cdots01)_{[n]}} = 1 \;\wedge\; \cdots \;\wedge\; f_{\mathbf{x}=(11\cdots1)_{[n]}} = 1$$

If all elements of $\mathbb{B}^n$ are pairwise symmetrical for $\mathbf{x}$ in f then for all $(a, b) \in \mathbb{B}^n \times \mathbb{B}^n$ and for some variable renaming π we have $f_{\mathbf{x}=\mathbf{a}} = \pi(f_{\mathbf{x}=\mathbf{b}})$. Then $f(\mathbf{x}_1, \mathbf{x}_2, \ldots, \mathbf{x}_m)$ is independent from $\mathbf{x}$ and it is sufficient to check that $f_{\mathbf{x}=\mathbf{a}} = 1$ holds for some $\mathbf{a} \in \mathbb{B}^n$ in order to check that $f = 1$ holds. Formally this reads:

$$f = 1 \;\; \Leftrightarrow \;\; \exists \mathbf{a} \in \mathbb{B}^n : f_{\mathbf{x}_i=\mathbf{a}} = 1 \;\wedge\; \forall \mathbf{a}, \mathbf{b} \in \mathbb{B}^n \exists \pi \in Sym(X \backslash \{\mathbf{x}\}) : f_{\mathbf{x}=\mathbf{a}} = \pi(f_{\mathbf{x}=\mathbf{b}}) \tag{10.1}$$

Thus $\mathbf{x}$ has been removed from the problem. Conversely, if there are values $\mathbf{a}$ and $\mathbf{b}$ for $\mathbf{x}$ such that $f_{\mathbf{x}=\mathbf{a}} \neq f_{\mathbf{x}=\mathbf{b}}$, $f \neq 1$ follows immediately. If the maximal symmetry relation $\sim_{f,\mathbf{x}}$ is known this can be decided easily. The symmetry relation extends to cofactors. If only an approximation of the symmetry relation is known it is sufficient to check one cofactor of each equivalence class of the symmetry relation for constantness in order the check whether $f = 1$ holds. This is formally stated in the lemma below.

Lemma 10.17
Let $f : \mathbb{B}^{n_1} \times \ldots \times \mathbb{B}^{n_m} \rightarrow \mathbb{B}$ be a bitvector function over $\mathbf{x}_1, \ldots, \mathbf{x}_m$. For some $\mathbf{x} \in \{\mathbf{x}_1, \ldots, \mathbf{x}_m\}$ let $Dom(\mathbf{x}) = \{\mathbf{a}_1, \ldots, \mathbf{a}_k\}$. Let $S = \{S_1, \ldots S_l\}$ be a partition of $Dom(\mathbf{x})$, and let $\sim_S$ be a symmetry relation for $\mathbf{x}$ in f. For $j \in \{1, \ldots, l\}$, let $\mathbf{d}_j \in S_j$ be any representative of S_j, then:

$$f = 1 \Leftrightarrow f_{\mathbf{x}=\mathbf{d}_1} = 1 \wedge \cdots \wedge f_{\mathbf{x}=\mathbf{d}_l} = 1 \tag{10.2}$$

[1]This has nothing to do with Shannon expansion, except that cofactors are used.

Proof: Obviously $f = 1 \Leftrightarrow f_{\mathbf{x}=\mathbf{a}_1} = 1 \wedge \cdots \wedge f_{\mathbf{x}=\mathbf{a}_k} = 1$. The symmetry relation $\sim_S$ is represented by a partition $S = \{S_1, \ldots S_l\}$ of $Dom(\mathbf{x})$. There is a corresponding partition $P = \{P_1, \ldots P_l\}$ of the set of cofactors $F = \{f_{\mathbf{x}=\mathbf{a}_1}, \ldots, f_{\mathbf{x}=\mathbf{a}_k}\}$ with $f_{\mathbf{x}=\mathbf{a}} = f_{\mathbf{x}=\mathbf{b}}$ iff $\mathbf{a} \sim_S \mathbf{b}$. For all equivalence classes P_j of $\sim_P$ we know that if $f_{\mathbf{x}=\mathbf{a}}, f_{\mathbf{x}=\mathbf{b}} \in P_j$ then for some $\pi \in Sym(X \backslash \{\mathbf{x}\})$, $f_{\mathbf{x}=\mathbf{a}} = \pi(f_{\mathbf{x}=\mathbf{b}})$ and therefore $f_{\mathbf{x}=\mathbf{a}} = 1$ iff $f_{\mathbf{x}=\mathbf{b}} = 1$. Thus for all $f'_j \in P_j$, $f' = 1$ holds iff this is the case for one of them. Therefore $f' = 1$ has to be proven for one f' of each equivalence class of $\sim_P$ in order to prove $f = 1$. ■

The lemma above can be extended to extended symmetry relations as below.

Lemma 10.18
Let $f : \mathbb{B}^{n_1} \times \ldots \times \mathbb{B}^{n_m} \rightarrow \mathbb{B}$ be a bitvector function over m bitvector variables X with $\mathbf{x}_1, \ldots, \mathbf{x}_k \in X$. Let $D := Dom(\mathbf{x}_1) \times \cdots \times Dom(\mathbf{x}_k)$ and let $S = \{S_1, \ldots S_l\}$ be a partition of D such that $\sim_S$ is a symmetry relation for $(\mathbf{x}_1, \ldots, \mathbf{x}_k)$ in f. For $j \in \{1, \ldots, l\}$, let $(\mathbf{a}_{1,j}, \ldots, \mathbf{a}_{k,j}) \in S_j$ then:

$$f = 1 \Leftrightarrow f_{\mathbf{x}_1=\mathbf{a}_{1,1} \cdots \mathbf{x}_k=\mathbf{a}_{k,1}} = 1 \ \wedge \ \cdots \ \wedge \ f_{\mathbf{x}_1=\mathbf{a}_{1,l} \cdots \mathbf{x}_k=\mathbf{a}_{k,l}} = 1 \tag{10.3}$$

Since we are computationally dealing with terms as representations for functions rather than with functions their self we have to extend above concepts to bitvector terms in order to make them applicable for BIMC-preprocessing.

10.1.5 Symmetrical Values in Bitvector Terms

We will see below that two values for a variable in a bitvector function are symmetrical iff two bitvector terms are permutation equivalent. Let t be a bitvector term, and let $\mathbf{x}$ be a variable occurring in t. Bitvector terms represent bitvector functions. The domains of the bitvector variables occurring in these terms are finite. Since the bitvector term algebra modulo bitvector axioms and the algebra of bitvector functions are isomorphic, constant symbols and constant functions coincide, i.e. for each constant symbol there is a constant function with the same name and vice versa. Let $\mathbf{x}$ be of width w then its domain is the finite set $\{\mathbf{a}_1, \ldots, \mathbf{a}_{2^w}\}$. Then as usual $t[\mathbf{x}/\mathbf{a}]$ denotes the term t, where all occurrences of the variable $\mathbf{x}$ are substituted with the value (constant) $\mathbf{a}$. Since t represents a bitvector function f, $t[\mathbf{x}/\mathbf{a}]$ represents the cofactor $f_{\mathbf{x}=\mathbf{a}}$ of f. Therefore, we can translate the concept of symmetrical values and symmetry relations to bitvector terms. We write $\mathbf{a} \sim_{t,\mathbf{x}} \mathbf{b}$ to indicate that $\mathbf{a}$ and $\mathbf{b}$ are symmetrical values for $\mathbf{x}$ in t.

Then symmetry relations naturally extend to cofactor terms in the following way. Let t be a term and $\mathbf{x}$ a variable in t. If $Dom(\mathbf{x}) = \{\mathbf{a}_1, \ldots, \mathbf{a}_{2^w}\}$, then the set of *cofactors for x* is $T = \{t[\mathbf{x}/\mathbf{a}_1], t[\mathbf{x}/\mathbf{a}_2], \ldots, t[\mathbf{x}/\mathbf{a}_{2^w}]\}$. A symmetry relation $\sim$ for $\mathbf{x}$ in t extends to T, such that $t[\mathbf{x}/\mathbf{a}] \sim t[\mathbf{x}/\mathbf{b}]$ iff $\mathbf{a} \sim \mathbf{b}$. The symmetry relation $\sim$ can be represented by a partition S of $\{\mathbf{a}_1, \ldots, \mathbf{a}_{2^w}\}$ as well as by a partition P of T. There is a one to one correspondence between the values $Dom(\mathbf{x})$ and the cofactors T, as well as between the equivalence classes of $\sim_S$ and sim_P. Thus the partition P of a set of cofactors T, and the symmetry relation $\sim_P$, as well as the symmetry relation $\sim_S$ and the partition S are basically the same thing.

Extended symmetry relations extend to sets of cofactors in the following way. Let t be a term with $\mathbf{x}_1, \ldots, \mathbf{x}_k \in X = Var(t)$ representing a bitvector function f over X. Then

$t[\mathbf{x}_1/\mathbf{a}_1]\cdots[\mathbf{x}_k/\mathbf{a}_k]$ with $\mathbf{a}_i \in Dom(\mathbf{x}_i)$ represents the cofactor $f_{\mathbf{x}_1=\mathbf{a}_1\cdots\mathbf{x}_k=\mathbf{a}_k}$ of f w.r.t. $\mathbf{x}_1,\ldots,\mathbf{x}_k$. Let T be the set of all cofactor terms of t w.r.t. $\mathbf{x}_1,\ldots,\mathbf{x}_k$, i.e.

$$T := \bigcup_{\mathbf{a}_1\in Dom(\mathbf{x}_1),\ldots,\mathbf{a}_k\in Dom(\mathbf{x}_k)} (t[\mathbf{x}_1/\mathbf{a}_1]\cdots[\mathbf{x}_k/\mathbf{a}_k])$$

Let $\sim$ be an extended symmetry relation for $\mathbf{x}_1,\ldots,\mathbf{x}_k$ in f then $\sim$ extends to an equivalence relation on T also denoted by $\sim$ with

$$(\mathbf{a}_1,\ldots,\mathbf{a}_k) \sim (\mathbf{b}_1,\ldots,\mathbf{b}_k) \;\Leftrightarrow\; (t[\mathbf{x}_1/\mathbf{a}_1]\cdots[\mathbf{x}_k/\mathbf{a}_k]) \sim (t[\mathbf{x}_1/\mathbf{b}_1]\cdots[\mathbf{x}_k/\mathbf{b}_k])$$

Now the symmetry relation $\sim$ can be represented by a partition of $Dom(\mathbf{x}_1)\times\cdots\times Dom(\mathbf{x}_k)$ as well as of T.

10.2 Basic Symmetry Reduction for RTL-BMC

In this section the exploitation of symmetrical values in bitvector terms of width[2] (sort) 1, representing bitvector functions with Boolean codomain such as for RTL-BMC, is explained. First the basic expansion and reduction scheme known from Chapter 5 is extended to one bitvector variable where all values are symmetrical. Then the maximal symmetry relation for one variable is used for reduction. It is shown that the problem of computing the maximal symmetry relation is reduced to the permutation equivalence problem. (See Chapter 9.) For complexity reasons it is advisable to find and use approximations of the maximal symmetry relations. Approximate symmetry relations can be combined which improves the reduction. A comprehensive example is used to illustrate this approach.

10.2.1 Basic Cofactor Expansion and Reduction

A bounded model checking problem is given as an extended[3] bitvector term t of sort 1 for which $\mathcal{A}_{\mathcal{BV}} \models t = 1$ has to be proven. The term t represents a bitvector function with Boolean codomain. The bounded model checking problem is the question whether the represented function is constantly 1. Now, let t be a bitvector term of width (sort) 1 over variables $\mathbf{x}_1,\ldots,\mathbf{x}_m$, representing a bitvector function $f : \mathbb{B}^{n_1}\times\ldots\times\mathbb{B}^{n_m} \to \mathbb{B}$ in variables $\mathbf{x}_1,\ldots,\mathbf{x}_m$. Therefore the symmetric cofactor expansion and all related results extend to bitvector terms of sort 1. Consequently $t = 1$ holds in the extended bitvector algebra $\mathcal{A}_{\mathcal{BV}}$ (See Section 6.3.) iff the conjunction of its finite instantiations holds, i.e. if for $\mathbf{x} \in Var(t)$ of width w, and $Dom(\mathbf{x}) = \{\mathbf{a}_1,\ldots,\mathbf{a}_{2^w}\}$:

$$\mathcal{A}_{\mathcal{BV}} \models t = 1 \;\Leftrightarrow\; \mathcal{A}_{\mathcal{BV}} \models t[\mathbf{x}/\mathbf{a}_1] = 1 \wedge\cdots\wedge \mathcal{A}_{\mathcal{BV}} \models t[\mathbf{x}/\mathbf{a}_{2^w}] = 1$$

Intuitively two values $\mathbf{a}$ and $\mathbf{b}$ are symmetrical values for a variable $\mathbf{x}$ occurring in a bitvector term t iff $t[\mathbf{x}/\mathbf{a}]$ and $t[\mathbf{x}/\mathbf{b}]$ represent the same function, i.e. iff $f_{\mathbf{x}=\mathbf{a}} = f_{\mathbf{x}=\mathbf{b}}$ i.e.

[2] Bitvector terms are many-sorted terms with naturals as sort symbols (Section 6.2).
[3] An extended bitvector term is a bitvector term in extended syntax as defined in 6.3.

if $\mathcal{A}_{\mathcal{BV}} \models t[\mathbf{x}/\mathbf{a}] = t[\mathbf{x}/\mathbf{b}]$. Suppose that all values $Dom(\mathbf{x}) = \{\mathbf{a}_1, \ldots, \mathbf{a}_k\}$ are pairwise symmetrical for $\mathbf{x}$ in t. Then for any $\mathbf{a} \in \{\mathbf{a}_1, \ldots \mathbf{a}_k\}$ we have:

$$\mathcal{A}_{\mathcal{BV}} \models t = 1 \Leftrightarrow \mathcal{A}_{\mathcal{BV}} \models t[\mathbf{x}/\mathbf{a}] = 1$$

Thus the variable $\mathbf{x}$ is eliminated, and it is easier to check whether the formula $t[\mathbf{x}/\mathbf{a}] = 1$ holds in $\mathcal{A}_{\mathcal{BV}}$ than it is for $t = 1$. Applying syntactic preprocessing techniques the term $t[\mathbf{x}/\mathbf{a}]$ can possibly be reduced further, for example by constant propagation as shown in the example below.

Example 10.19 (Symmetry Reduction)
As an example consider the bitvector term p' which is identical to the term p'_{write} for which $p'_{write} = 1$ has to be shown in order to prove the write property for the memory design.

$$\begin{aligned} p' \equiv\ & \mathbf{rs_t} = \mathbf{0} \wedge \mathbf{cs_t} = \mathbf{1} \wedge \mathbf{rw_t} = \mathbf{1} \Rightarrow \\ & \mathbf{di_t} = \text{read}(\\ & \quad (\ \text{ite}(\mathbf{rs_t} = \mathbf{1}, \mathbf{0}, (\text{ite}(\mathbf{cs_t} = \mathbf{1} \wedge \mathbf{a_t} = \mathbf{255} \wedge \mathbf{rw_t} = \mathbf{1}, \mathbf{di_t}, \mathbf{m255_t}))) \\ & \quad\ \otimes \text{ite}(\mathbf{rs_t} = \mathbf{1}, \mathbf{0}, (\text{ite}(\mathbf{cs_t} = \mathbf{1} \wedge \mathbf{a_t} = \mathbf{254} \wedge \mathbf{rw_t} = \mathbf{1}, \mathbf{di_t}, \mathbf{m254_t}))) \\ & \quad\ \otimes \cdots \\ & \quad\ \otimes \text{ite}(\mathbf{rs_t} = \mathbf{1}, \mathbf{0}, (\text{ite}(\mathbf{cs_t} = \mathbf{1} \wedge \mathbf{a_t} = \mathbf{0} \wedge \mathbf{rw_t} = \mathbf{1}, \mathbf{di_t}, \mathbf{m0_t}))) \\ & \quad), \mathbf{a_t}) \end{aligned} \tag{10.4}$$

The question is whether p' represents the constant function 1. The bitvector function naturally represented by p' is defined as $f : \mathbb{B} \times \mathbb{B} \times \mathbb{B} \times \mathbb{B}^{16} \times \mathbb{B}^{8} \times \mathbb{B}^{256 \cdot 16} \rightarrow \mathbb{B}$.

Suppose that all 256 values for $\mathbf{a_t}$ in p' (Eq. (10.4)) are symmetrical then $p' = 1$ iff $p'[\mathbf{a_t}/\mathbf{0}] = 1$. (Here $\mathbf{0}$ denotes (00000000).) The term $p'[\mathbf{a_t}/\mathbf{0}]$ shown below does not appear to be significantly smaller, although the represented function is independent of $\mathbf{a_t}$ which already leads to a reduction of the search space to 1/256.

$$\begin{aligned} p'[\mathbf{a_t}/\mathbf{0}] \equiv\ & \mathbf{rs_t} = \mathbf{0} \wedge \mathbf{cs_t} = \mathbf{1} \wedge \mathbf{rw_t} = \mathbf{1} \Rightarrow \\ & \mathbf{di_t} = \text{read}(\\ & \quad (\ \text{ite}(\mathbf{rs_t} = \mathbf{1}, \mathbf{0}, (\text{ite}(\mathbf{cs_t} = \mathbf{1} \wedge \mathbf{0} = \mathbf{255} \wedge \mathbf{rw_t} = \mathbf{1}, \mathbf{di_t}, \mathbf{m255_t}))) \\ & \quad\ \otimes \text{ite}(\mathbf{rs_t} = \mathbf{1}, \mathbf{0}, (\text{ite}(\mathbf{cs_t} = \mathbf{1} \wedge \mathbf{0} = \mathbf{254} \wedge \mathbf{rw_t} = \mathbf{1}, \mathbf{di_t}, \mathbf{m254_t}))) \\ & \quad\ \otimes \cdots \\ & \quad\ \otimes \text{ite}(\mathbf{rs_t} = \mathbf{1}, \mathbf{0}, (\text{ite}(\mathbf{cs_t} = \mathbf{1} \wedge \mathbf{0} = \mathbf{0} \wedge \mathbf{rw_t} = \mathbf{1}, \mathbf{di_t}, \mathbf{m0_t}))) \\ & \quad), \mathbf{0}) \end{aligned} \tag{10.5}$$

Now, applying constant propagation yields an equivalent term p''_0 shown below. (Since $\mathbf{a_t}$ has been replaced with $\mathbf{0}$, the resulting equalities like '$\mathbf{0} = \mathbf{0}$' or '$\mathbf{0} = \mathbf{255}$' and the big read access 'read($\cdot \otimes \cdots \otimes \cdot, \mathbf{0}$)' can be evaluated and $p'[\mathbf{a_t}/\mathbf{0}]$ in Eq. (10.5) can be rewritten into p''_0 as in Eq. (10.6))

$$\begin{aligned} p''_0 \equiv\ & (\mathbf{rs_t} = \mathbf{0} \wedge \mathbf{cs_t} = \mathbf{1} \wedge \mathbf{rw_t} = \mathbf{1}) \Rightarrow \\ & \quad \mathbf{di_t} = \text{ite}(\mathbf{rs_t} = \mathbf{1}, \mathbf{0}, (\text{ite}(\mathbf{cs_t} = \mathbf{1} \wedge \mathbf{rw_t} = \mathbf{1}, \mathbf{di_t}, \mathbf{m0_t}))) \end{aligned} \tag{10.6}$$

The term p''_0 is indeed significantly smaller than p'. Note that along with $\mathbf{a_t}$, the 255 variables $\mathbf{m255_t}, \ldots, \mathbf{m1_t}$ have been removed. Each of the variables $\mathbf{m255_t}, \ldots, \mathbf{m1_t}$ has a domain size of 2^{16}, while $\mathbf{a_t}$ has a domain size of 2^8. Thus the search space for the problem has been reduced from 2^{4123} to $2^{3+2*16} = 2^{35}$. It is now sufficient to check whether $p''_0 = 1$ holds in order to prove $p' = 1$ provided that all values for $\mathbf{a_t}$ are symmetrical.

10.2.2 Using the Maximal Symmetry Relation

In general it cannot be expected that all values for a variable are pairwise symmetrical, instead they form an equivalence relation, a symmetry relation. Lemma 10.17 translates to bitvector terms of sort 1 as follows:

Corollary 10.20
Let t be a bitvector term of sort 1. Let $\mathbf{x} \in Var(t)$. Let $Dom(\mathbf{x}) = \{\mathbf{a}_1, \ldots, \mathbf{a}_{2^w}\}$. Let $S = \{S_1, \ldots S_l\}$ be a partition of $Dom(\mathbf{x})$, and let $\sim_S$ be a symmetry relation for $\mathbf{x}$ in t. For $j \in \{1, \ldots, l\}$, let $\mathbf{a}_j \in S_j$ be any representative of S_j, then:

$$\mathcal{A}_{\mathcal{BV}} \models t = 1 \Leftrightarrow \mathcal{A}_{\mathcal{BV}} \models t[\mathbf{x}/\mathbf{a}_1] = 1 \wedge \cdots \wedge \mathcal{A}_{\mathcal{BV}} \models t[\mathbf{x}/\mathbf{a}_l] = 1 \tag{10.7}$$

Given the maximal symmetry relation the following holds. If all values for a variable are symmetrical, just one cofactor has to be checked. If, in the contrary, not all values are symmetrical, then the function represented by t is not constant, and hence $t = 1$ cannot hold. This can now be used to construct a simple algorithm for tautology checking, based on the maximal symmetry relation. Let `T` be the set of all cofactors of the bitvector term `t` w.r.t. `x`. Let algorithm

```
Boolean symRel(Set<Term> T, Partition<Term>& P)
```

return true, iff all cofactors `T` are pairwise symmetrical, and let it return the maximal symmetry relation for `x` in `t` as partition `P` of cofactors `T`. Now, given a bitvector term `t` of sort 1, Algorithm 3, Pg. 141, checks whether `t` = 1 is tautological (holds), is unsatisfiable or satisfiable.

The algorithm works recursively as follows.

1. At the beginning, when `search` is called for the first time, `X` contains all variables of `t`.
2. If `X` is empty `t` is constant, then it is either constantly `1` or `0`, hence it is either `SAT` or `UNSAT`.
3. If there are variables in `X`, the function `choose(Variables X)` selects an arbitrary variable `x` from `X`.
4. Then `D` is the domain of `x` and `P` is a partition of cofactors, initially empty.
5. The function `symRel(Terms T, Partition<Term>& P)` computes the maximal symmetry relation of `x` in `t`. When it returns, `P` contains the partition of cofactors of `t` representing the maximal symmetry relation. It returns true if all values are pairwise symmetrical values for `x` in `t`, and false otherwise.
6. If it returns false, then `t` is not constant. Therefore `SAT` is returned.
7. If it returns true, then `P` is a unit partition, and any value for `x` can be chosen and substituted for `x` into `t`. Then `t` = 1 holds iff `t'` = 1 holds for any cofactor in `T`.
8. The recursion stops eventually since the variables are subsequently eliminated until `X` is empty (then `UNSAT` or `TAUT` is returned), or some values are not symmetrical for a variable (then `SAT` is returned).

Algorithm 3 SAT Using the Maximal Symmetry Relation

```
enum Result { UNSAT, SAT, TAUT };     // Possible return values.
Result search(Term t)                 // Computes whether t=1 is SAT,
{                                     // TAUT or UNSAT.
   Variables X = Var(t);              // The variables in t.
   if (X.empty())                     // If there are no variables in t
   {                                  // then t must be constant.
       if ( t == 1 )                  // Return
           return TAUT;               // TAUT, if t is constantly 1
       else                           // and
           return UNSAT;              // UNSAT, if t is constantly 0.
   }                                  // If there are variables
   Variable x = choose(X);            // select a variable x, and
   Values D = dom(x);                 // get all possible values for x.
   Partition<Term> P;                 // P represents the symmetry relation.
   Terms T;                           // T is the set of cofactors of t
   foreach (Value c in D) {           // Compute set of cofactors
     T.insert(substitute(t,x,c));
   }
   if ( ! symRel(T, P) )              // If not all instances are symetrical
        return SAT;                   // then t is satisfiable, otherwise
   else                               // continue search with a cofactor term.
     return search(choose(T));
}
```

10.2.3 Computing Maximal Symmetry Relations

In order to use the symmetry reduction described in Section 10.2 productively, i.e. within a RTL-BMC-tool, a fully automatic and efficient procedure is needed to find the maximal symmetry relation for variables in RTL-BMC-problems, or at least good approximations.

Example 10.21
In order to reduce the RTL-BMC-problem in Eq. (10.4), it has to be shown that $\{\mathbf{0}, \ldots, \mathbf{255}\}$ are symmetrical values of $\mathbf{a_t}$ in Eq. (10.4).

Two values $\mathbf{a}$ and $\mathbf{b}$ are symmetrical values for $\mathbf{x}$ in a bitvector term t of sort 1 iff the cofactors $t[\mathbf{x}/\mathbf{a}]$ and $t[\mathbf{x}/\mathbf{b}]$ are permutation equivalent, i.e. iff $t[\mathbf{x}/\mathbf{a}] = t[\mathbf{x}/\mathbf{b}]\pi$ holds for some $\pi \in Var(t)\backslash\{\mathbf{x}\}$. This means that $\mathbf{a}$ and $\mathbf{b}$ are symmetrical values for $\mathbf{x}$ in t iff $t[\mathbf{x}/\mathbf{a}]$ and $t[\mathbf{x}/\mathbf{b}]$ are permutation equivalent.

Example 10.22
Consider the two values (00000000) and (10101010) for the variable $\mathbf{a_t}$ in the formula p' in Eq. (10.4). They are written as $\mathbf{0}$ and $\mathbf{170}$ to ease readability. Then the values $\mathbf{0}$ and $\mathbf{170}$ are symmetrical values for $\mathbf{a_t}$ in p' if the following two terms $p'[\mathbf{a_t}/\mathbf{0}]$ and $p'[\mathbf{a_t}/\mathbf{170}]$, as in Eq. (10.5) and Eq. (10.8) are permutation equivalent.

$$\begin{aligned}
p'[\mathbf{a_t}/\mathbf{170}] \equiv\ & \mathbf{rs_t} = \mathbf{0} \wedge \mathbf{cs_t} = \mathbf{1} \wedge \mathbf{rw_t} = \mathbf{1} \Rightarrow \\
& \mathbf{di_t} = \mathrm{read}(\\
& \quad (\ \mathrm{ite}(\mathbf{rs_t} = \mathbf{1}, \mathbf{0}, (\mathrm{ite}(\mathbf{cs_t} = \mathbf{1} \wedge \mathbf{170} = \mathbf{255} \wedge \mathbf{rw_t} = \mathbf{1}, \mathbf{di_t}, \mathbf{m255_t}))) \\
& \quad\ \otimes \mathrm{ite}(\mathbf{rs_t} = \mathbf{1}, \mathbf{0}, (\mathrm{ite}(\mathbf{cs_t} = \mathbf{1} \wedge \mathbf{170} = \mathbf{254} \wedge \mathbf{rw_t} = \mathbf{1}, \mathbf{di_t}, \mathbf{m254_t}))) \\
& \quad\ \otimes \cdots \\
& \quad\ \otimes \mathrm{ite}(\mathbf{rs_t} = \mathbf{1}, \mathbf{0}, (\mathrm{ite}(\mathbf{cs_t} = \mathbf{1} \wedge \mathbf{170} = \mathbf{0} \wedge \mathbf{rw_t} = \mathbf{1}, \mathbf{di_t}, \mathbf{m0_t}))) \\
& \quad), \mathbf{170})
\end{aligned} \tag{10.8}$$

If we can show that the terms $p'[\mathbf{a_t}/\mathbf{0}]$ and $p'[\mathbf{a_t}/\mathbf{170}]$ are permutation equivalent, we know that the two values $\mathbf{0}$ and $\mathbf{170}$ are symmetrical values for $\mathbf{a_t}$ in p'. We show that this is the case by rewriting the two cofactors $p'[\mathbf{a_t}/\mathbf{0}]$ and $p'[\mathbf{a_t}/\mathbf{170}]$ until they are identical modulo renaming of variables, while preserving the function each of them represents. Then the values $\mathbf{0}$ and $\mathbf{170}$ must be symmetrical values for $\mathbf{a_t}$ in p'. Applying constant propagation we get $p'[\mathbf{a_t}/\mathbf{0}] = p''_0$ and $p'[\mathbf{a_t}/\mathbf{170}] = p''_{170}$ with p''_0 and p''_{170} as in Eq. (10.6) and Eq. (10.9) below.

$$\begin{aligned}
p''_{170} \equiv\ & (\mathbf{rs_t} = \mathbf{0} \wedge \mathbf{cs_t} = \mathbf{1} \wedge \mathbf{rw_t} = \mathbf{1}) \Rightarrow \\
& \quad \mathbf{di_t} = \mathrm{ite}(\mathbf{rs_t} = \mathbf{1}, \mathbf{0}, (\mathrm{ite}(\mathbf{cs_t} = \mathbf{1} \wedge \mathbf{rw_t} = \mathbf{1}, \mathbf{di_t}, \mathbf{m170_t})))
\end{aligned} \tag{10.9}$$

Now, the two terms p''_0 and p''_{170} only differ in variable names $\mathbf{m0_t}$ and $\mathbf{m170_t}$. (Rename for example $\mathbf{m170_t}$ in p''_{170} with $\mathbf{m0_t}$.) Then we have $p''_0 \equiv p''_{170}[\mathbf{m170_t}/\mathbf{m0_t}]$, i.e. they are identical. Thus $\mathbf{0}$ and $\mathbf{170}$ are symmetrical values for $\mathbf{a_t}$ in p'. This is also true for any other pair of values from $\{\mathbf{0}, \ldots, \mathbf{255}\}$ since the construction is independent of the value chosen.

The example above gives an idea of how symmetrical values can be found in general. This is indeed always possible, since it is equivalent to the permutation equivalence problem for bitvector functions, which is a Co-NP complete problem, as shown in Chapter 9.

We know that the domains of the variables are finite and that there is an algorithm to decide whether $a \sim_{t,x} b$ holds (by virtue of permutation equivalence of $t[\mathbf{x}/\mathbf{a}]$ and $t[\mathbf{x}/\mathbf{b}]$). Given a term t, the maximal symmetry relation for a variable x in t can than be computed by generating all cofactors of t w.r.t. x, and checking if they are pairwise symmetrical using Algorithm 18 (Pg. 199), with `Boolean symmetrical(ta,tb)` instead of `equiv`. Note that, if '`symmetrical(ta,tb)=true`' implies $a \sim_{t,x} b$, but not conversely, then `P` returned by Algorithm 18 is an approximation of the maximal symmetry relation of x in t.

Example 10.23
In the memory example, finding symmetrical values for $\mathbf{a_t}$ can be done by generating all the cofactors of Eq. (10.4) for $\mathbf{a_t} = \mathbf{0}$, $\mathbf{a_t} = \mathbf{1}$, ..., $\mathbf{a_t} = \mathbf{255}$, and checking if they are pairwise equivalent. (The instance for $\mathbf{a_t} = \mathbf{0}$ is shown in Eq. (10.5), Pg. 139.)

In theory the symmetrical value approach to RTL-BMC does not help much, since the permutation equivalence problem is hard. However, approximate symmetry relations can be used as well to reduce the size of a term. In the following section we will use approximate symmetry relations, which can be found by practical algorithms.

10.2.4 Using Approximate Symmetry Relations

Note that if the maximal symmetry relation of a variable $\mathbf{x}$ in a term t is not known, but only an approximation, Lemma 10.17 can still be used to reduce t.

The following Algorithm 4 '`searchA`' illustrates, that given an Algorithm '`symRelA`' that computes only an approximate symmetry relation, still a complete decision procedure can be constructed. The algorithm recursively eliminates all variables. For each variable that is eliminated first and approximate symmetry relation is computed as partition of the set of cofactors. In the best case the resulting partition is unit. Then all values are pairwise symmetric. In the worst case the partition of cofactor terms is the discrete partition. For each equivalence class one cofactor is searched. Note that in the worst case the algorithm enumerates all possible assignments! Depending on the quality of the computed approximate symmetry relations the complexity may be better.

10.2.5 Combining Symmetry Relations

An approximate symmetry relation is better than another one if the represented equivalence classes are larger, because fewer instances have to be checked. Approximate symmetry relations can be combined to coarser symmetry relations by Lemma 10.16. The resulting approximation is then a better approximation of the maximal symmetry relation leading to fewer cofactor terms to be considered.

A symmetry relation is represented by a partition of values or terms. Then symmetry relations $\sim_P$ and $\sim_Q$ given by partitions P and Q of T are combined by merging non-disjoint classes of $\sim_P$ and $\sim_Q$.

Now, given n algorithms to compute approximate symmetry relations for a given partition of terms, an algorithm that computes the combined symmetry relation is Alg. 5.

Algorithm 4 SAT Using an Approximate Symmetry Relation

```
enum Result { TAUT, UNSAT, SAT };
Result searchA(Term t)
{
   Variables X = Var(t);            // The variables in t.
   if (X.empty())                   // If all variables were removed
   {                                // then tb can be evaluated.
       if ( t == 1 )
           return TAUT;
       else
           return UNSAT;
   }
   Variable x = choose(X);          // Select a variable.
   Values D = dom(x);               // Get the possible values for x.
   Partition<Term> P;               // The partition of instances
                                    // representing the symmetry relation.
   Terms T;                         // Set of cofactors
   foreach ( Value c in D )         // For each value generate a cofactor
      T.insert(substitute(t,x,c));  // and put into T.
   symRelA(T,P);                    // Compute an approximate symmetry
                                    // relation for x in t.
   Boolean u,v = false;             // Taut. and unsat. cofactors.
   foreach (Term tc in P.reps())    // For each representative for the
   {                                // equivalence classes
       Result r = searchA(tc);      // search in a reduced problem.
       if ( r == SAT ) return SAT;  // If a cofactor is SAT so is t.
       if ( r == UNSAT ) u = true;  // At least one UNSAT.
       if ( r == TAUT ) v = true;   // At least one TAUT.
   }                                // if not SAT continue search.
   if ( u && !v) return UNSAT;      // All values have been searched.
   if ( !u && v) return TAUT;       // All cofactors TAUT or UNSAT?
   return SAT;                      // Both TAUT and UNSAT cofactors.
}
```

Algorithm 5 Combining Approximate Symmetry Relation

```
Boolean symRelA(Set<Term> T, Partition<Term>& P)
{
   if ( symRelA_1(T,P) )
     return true;
   foreach ( i = 2 .. n )
   {
       Partition<Term> Q;
       Boolean result = symRelA_i(P.reps(),Q);
       P.merge(Q);
       if (result)
          return true;
   }
   return false;
}
```

10.2.6 Computing Approximate Symmetry Relations

Since finding the maximal symmetry relation for a variable $\mathbf{x}$ in a bitvector term t might be impractical, different approximate symmetry relations are used instead.

As shown above the problem of finding a symmetry relation can be solved by subsequently considering two different cofactors of a term and solve the permutation equivalence problem for them. Since this is a hard problem and may be infeasible, we use the sufficient criteria for permutation equivalence of terms to compute approximate symmetry relations, which are represented by partitions of the set of cofactor terms considered. It will be shown that they can be combined to find better approximations. Thus a better approximation of the maximal symmetry relation can be computed subsequently using different sufficient criteria for permutation equivalence. RTL-BMC-problems can then be reduced along the lines of Lemma 10.17.

Let in the following t be a bitvector term and let $\mathbf{x}$ be a variable occurring in t. For two values $\mathbf{a}, \mathbf{b} \in Dom(\mathbf{x})$ to show that $\mathbf{a} \sim_{t,\mathbf{x}} \mathbf{b}$, the problem to be solved is finding a variable renaming π such that $t[\mathbf{x}/\mathbf{a}]$ and $t[\mathbf{x}/\mathbf{b}]$ are permutation equivalent in the extended bitvector algebra $\mathcal{A}_{\mathcal{BV}}$,i.e.

$$\exists \pi \in Sym(Var(t) \backslash \{\mathbf{x}\}) : \mathcal{A}_{\mathcal{BV}} \models t[\mathbf{x}/\mathbf{a}] = t[\mathbf{x}/\mathbf{b}]\pi$$

Lemma 10.24
Let $\sim$ be an equivalence relation on terms which implies permutation equivalence. Let T be the set of all cofactor terms of t w.r.t. $\mathbf{x}$. Then $\sim$ defines a symmetry relation for the values of $\mathbf{x}$ in t where $t[\mathbf{x}/\mathbf{a}] \sim t[\mathbf{x}/\mathbf{a}]$ implies $\mathbf{a} \sim_{t,\mathbf{x}} \mathbf{b}$.

Example 10.25

1. The simplest equivalence relation on a set of terms is identity. Identity is an equivalence relation, and it defines a symmetry relation. However, this symmetry relation is trivial for cofactor terms, since two cofactor terms $t[\mathbf{x}/\mathbf{a}]$ and $t[\mathbf{x}/\mathbf{b}]$ always differ in the positions where $\mathbf{x}$ has been replaced with $\mathbf{a}$ and $\mathbf{b}$, respectively.

2. If two cofactor terms are identical after renaming of variables they are permutation equivalent, i.e. if they are syntactically symmetric. However the induced symmetry relation is again trivial, since different cofactor terms differ in the positions of removed variables and are therefore never structurally symmetric.

3. We may employ some semantics of $\mathcal{A}_{\mathcal{BV}}$ captured in a set of identities E with $\mathcal{A}_{\mathcal{BV}} \models E$. Then $\approx_E$ defines a symmetry relation. Let for example s_2, $s_2[\mathbf{x}/\mathbf{0}]$, $s_2[\mathbf{x}/\mathbf{1}]$ be bitvector terms as below.

$$\begin{aligned} s_2 &\equiv \text{ite}(\mathbf{x}, \mathbf{y}_{[3]}, \mathbf{y}_{[3]}) + (\text{ite}(\neg\mathbf{x}, (011), (111)) + ((100) \ll \mathbf{x})) \\ s_2[\mathbf{x}/\mathbf{0}] &\equiv \text{ite}(\mathbf{0}, \mathbf{y}_{[3]}, \mathbf{y}_{[3]}) + (\text{ite}(\neg\mathbf{0}, (011), (111)) + ((100) \ll \mathbf{0})) \\ s_2[\mathbf{x}/\mathbf{1}] &\equiv \text{ite}(\mathbf{1}, \mathbf{y}_{[3]}, \mathbf{y}_{[3]}) + (\text{ite}(\neg\mathbf{1}, (011), (111)) + ((100) \ll \mathbf{1})) \end{aligned}$$

   Then $s_2[\mathbf{x}/\mathbf{0}]$ and $s_2[\mathbf{x}/\mathbf{1}]$ are equivalent which can be shown using constant propagation.

4. If two cofactor terms $t[\mathbf{x}/\mathbf{a}]$ and $t[\mathbf{x}/\mathbf{b}]$ are structurally identical modulo some set of identities E, then $\mathbf{a}$ and $\mathbf{b}$ are symmetrical values for $\mathbf{x}$ in t. The equivalence relation $\approx_{ES}$ is therefore a symmetry relation. Combining the relations $\approx_E$ and $\approx_S$, for example by first using rewriting on cofactor terms and then checking for syntactic symmetries yields an approximation of $\approx_{ES}$ which is also a symmetry relation for cofactor terms. For example the cofactor terms in Eq. (10.5) and Eq. (10.8) are symmetrical by this argument.

5. Last (and least) the semantic equivalence relation on $\mathcal{A}$ is mentioned here. If for two cofactor terms $\mathcal{A} \models t[x/a] = t[x/b]$, they are obviously permutation equivalent and the induced equivalence relation on cofactor terms is a symmetry relation. For the bitvector algebra $\mathcal{A}_{\mathcal{BV}}$, the decision problem of whether $s = t$ holds in $\mathcal{A}_{\mathcal{BV}}$ is $Co\text{-}NP$-complete. Therefore, in general it is hard to decide whether $t[x/a] = t[x/b]$ holds in $\mathcal{A}_{\mathcal{BV}}$. However, there is a good chance that $t[x/a]$ and $t[x/b]$ are structurally similar. Then the problem can be solved using equivalence checking technology already used in equivalence checking tools [DH02].

10.2.7 A Comprehensive Example

The following example illustrates that approximate symmetry relations are in fact useful to reduce RTL-BMC-problems, and most important, for finding errors in designs.

Example 10.26
Let the previously verified memory design from 8.2.1 be used within another design that stores some specific data in this memory. Let the new design have additional inputs $\mathbf{d0}_{[16]}$, $\mathbf{d1}_{[16]}$, and $\mathbf{d2}_{[16]}$, each of width 16. The data input $\mathbf{di}$ of the memory depends on the new inputs $\mathbf{d0}$, $\mathbf{d1}$, $\mathbf{d2}$, and the address input $\mathbf{a}$. This dependency is specified by the term f, with

$$f \equiv \text{ite}((\mathbf{a\%2}) = \mathbf{0}, \mathbf{d0}, \mathbf{d1}) \wedge \text{ite}((\mathbf{a\%3}) = \mathbf{0}, \mathbf{d0}, \text{ite}((\mathbf{a\%3}) = \mathbf{1}, \mathbf{d1}, \mathbf{d2}))$$

(Here % denotes the modulo operator, and $\wedge$ bitwise conjunction of two BVs.) Depending on the address $\mathbf{a}$, f is the bitwise conjunction of two terms t_0, t_1, with

$$\begin{aligned} t_0 &\equiv \text{ite}((\mathbf{a\%2}) = \mathbf{0}, \mathbf{d0}, \mathbf{d1}) \\ t_1 &\equiv \text{ite}((\mathbf{a\%3}) = \mathbf{0}, \mathbf{d0}, \text{ite}((\mathbf{a\%3}) = \mathbf{1}, \mathbf{d1}, \mathbf{d2})) \end{aligned}$$

If $\mathbf{a}$ is even, i.e. $(\mathbf{a\%2}) = \mathbf{0}$, then $t_0 = \mathbf{d0}$, otherwise $t_0 = \mathbf{d1}$. If $(\mathbf{a\%3}) = \mathbf{0}$ then $t_1 = \mathbf{d0}$, otherwise, if $(\mathbf{a\%3}) = \mathbf{1}$, then $t_1 = \mathbf{d1}$, otherwise $t_1 = \mathbf{d2}$, The structure of the intended composite design is depicted in Figure 10.1. For some reason the implementation differs

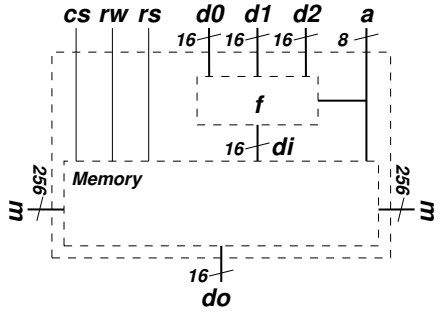


Figure 10.1: Memory Used to Store Specific Data

from this intention in that f' is used instead of f, with

$$f' \equiv \text{ite}((\mathbf{a\%3} = \mathbf{0}), \mathbf{d0}, \text{ite}((\mathbf{a\%2}) = \mathbf{1}, \mathbf{d1}, \mathbf{d2})) \wedge \text{ite}((\mathbf{a\%2}) = \mathbf{0}, \mathbf{d0}, \mathbf{d1})$$

The term f' is the conjunction of the terms t'_1, and t_0 (defined above), where

$$t'_1 \equiv \text{ite}((\mathbf{a\%3} = \mathbf{0}), \mathbf{d0}, \text{ite}((\mathbf{a\%2}) = \mathbf{1}, \mathbf{d1}, \mathbf{d2}))$$

Consider the subterms t_0, t_1 and t'_1 occurring in the specification $f = t_0 \wedge t_1$ and in the implementation $f' = t'_1 \wedge t_0$. Note that the designer made a little mistake in the implementation (f'). The two parts $t0$ and $t1$ of the conjunction are interchanged, which is ok, but the condition of the second multiplexer on the left-hand-side (t'_1) of the conjunction is changed from $(\mathbf{a\%3}) = \mathbf{1}$ into $(\mathbf{a\%2}) = \mathbf{1}$.

We want to verify the correctness of the write behavior of the composite design using RTL-BMC. It is to be expected that it is not correctly implemented. We want to find an approximate symmetry relation for the variable $\mathbf{a_t}$ in the resulting RTL-BMC-problem and reduce the problem size. The specification for the write operation is:

$$\begin{aligned} p_{write2} &\equiv (\mathbf{rs_t} = \mathbf{0} \wedge \mathbf{cs_t} = \mathbf{1} \wedge \mathbf{rw_t} = \mathbf{1}) \Rightarrow \text{read}(\mathbf{m_{t+1}}, \mathbf{a_t}) = f_\mathsf{t} \\ f_\mathsf{t} &\equiv \text{ite}((\mathbf{a_t\%2}) = \mathbf{0}, \mathbf{d0_t}, \mathbf{d1_t}) \wedge \text{ite}((\mathbf{a_t\%3}) = \mathbf{0}, \mathbf{d0_t}, \text{ite}((\mathbf{a_t\%3}) = \mathbf{1}, \mathbf{d1_t}, \mathbf{d2_t})) \end{aligned}$$

The implementation of the transition term for $\mathbf{m}$ in the composite design using f' leads to:

$$\begin{aligned} \mathbf{m_{t+1}} = \; & (\ \text{ite}(\mathbf{rs_t} = \mathbf{1}, \mathbf{0}, (\text{ite}(\mathbf{cs_t} = \mathbf{1} \wedge \mathbf{a_t} = \mathbf{0} \wedge \mathbf{rw_t} = \mathbf{1}, f'_\mathsf{t}, \mathbf{m255_t}))) \\ & \otimes \text{ite}(\mathbf{rs_t} = \mathbf{1}, \mathbf{0}, (\text{ite}(\mathbf{cs_t} = \mathbf{1} \wedge \mathbf{a_t} = \mathbf{1} \wedge \mathbf{rw_t} = \mathbf{1}, f'_\mathsf{t}, \mathbf{m254_t}))) \\ & \otimes \cdots \\ & \otimes \text{ite}(\mathbf{rs_t} = \mathbf{1}, \mathbf{0}, (\text{ite}(\mathbf{cs_t} = \mathbf{1} \wedge \mathbf{a_t} = \mathbf{255} \wedge \mathbf{rw_t} = \mathbf{1}, f'_\mathsf{t}, \mathbf{m0_t}))) \\ &) \\ f'_\mathsf{t} \equiv \; & \text{ite}((\mathbf{a_t\%3}) = \mathbf{0}, \mathbf{d0_t}, \text{ite}((\mathbf{a_t\%2}) = \mathbf{1}, \mathbf{d1_t}, \mathbf{d2_t})) \wedge \text{ite}((\mathbf{a_t\%2}) = \mathbf{0}, \mathbf{d0_t}, \mathbf{d1_t}) \end{aligned}$$

The resulting RTL-BMC-problem is then:

$$
\begin{aligned}
p'_{write2} \equiv \quad & (\mathbf{rs_t} = \mathbf{0} \wedge \mathbf{cs_t} = \mathbf{1} \wedge \mathbf{rw_t} = \mathbf{1}) \Rightarrow \text{read}(\\
& \quad (\text{ite}(\mathbf{rs_t} = \mathbf{1}, \mathbf{0}, (\text{ite}(\mathbf{cs_t} = \mathbf{1} \wedge \mathbf{a_t} = \mathbf{0} \wedge \mathbf{rw_t} = \mathbf{1}, f'_t, \mathbf{m255_t}))) \\
& \quad\quad \otimes \text{ite}(\mathbf{rs_t} = \mathbf{1}, \mathbf{0}, (\text{ite}(\mathbf{cs_t} = \mathbf{1} \wedge \mathbf{a_t} = \mathbf{1} \wedge \mathbf{rw_t} = \mathbf{1}, f'_t, \mathbf{m254_t}))) \\
& \quad\quad \otimes \cdots \\
& \quad\quad \otimes \text{ite}(\mathbf{rs_t} = \mathbf{1}, \mathbf{0}, (\text{ite}(\mathbf{cs_t} = \mathbf{1} \wedge \mathbf{a_t} = \mathbf{255} \wedge \mathbf{rw_t} = \mathbf{1}, f'_t, \mathbf{m0_t}))) \\
& \quad\quad), \mathbf{a_t}) = f_t
\end{aligned}
$$

To find an approximate symmetry relation for a_t in p'_{write2}, we first generate all 256 cofactor terms for a_t in p'_{write2}. Applying constant propagation and comparing them does not give us any information, since the resulting terms all are different. The first 6 cofactor terms after applying constant propagation are shown below:

$$
\begin{aligned}
i_0 \equiv \quad & (\mathbf{rs_t} = \mathbf{0} \wedge \mathbf{cs_t} = \mathbf{1} \wedge \mathbf{rw_t} = \mathbf{1}) \Rightarrow \\
& \quad \mathbf{d0_t} \wedge \mathbf{d0_t} = \text{ite}(\mathbf{rs_t} = \mathbf{1}, \mathbf{0}, (\text{ite}(\mathbf{cs_t} = \mathbf{1} \wedge \mathbf{rw_t} = \mathbf{1}, \mathbf{d0_t} \wedge \mathbf{d0_t}, \mathbf{m0_t}))) \\
i_1 \equiv \quad & (\mathbf{rs_t} = \mathbf{0} \wedge \mathbf{cs_t} = \mathbf{1} \wedge \mathbf{rw_t} = \mathbf{1}) \Rightarrow \\
& \quad \mathbf{d1_t} \wedge \mathbf{d1_t} = \text{ite}(\mathbf{rs_t} = \mathbf{1}, \mathbf{0}, (\text{ite}(\mathbf{cs_t} = \mathbf{1} \wedge \mathbf{rw_t} = \mathbf{1}, \mathbf{d1_t} \wedge \mathbf{d1_t}, \mathbf{m1_t}))) \\
i_2 \equiv \quad & (\mathbf{rs_t} = \mathbf{0} \wedge \mathbf{cs_t} = \mathbf{1} \wedge \mathbf{rw_t} = \mathbf{1}) \Rightarrow \\
& \quad \mathbf{d0_t} \wedge \mathbf{d2_t} = \text{ite}(\mathbf{rs_t} = \mathbf{1}, \mathbf{0}, (\text{ite}(\mathbf{cs_t} = \mathbf{1} \wedge \mathbf{rw_t} = \mathbf{1}, \mathbf{d2_t} \wedge \mathbf{d0_t}, \mathbf{m2_t}))) \\
i_3 \equiv \quad & (\mathbf{rs_t} = \mathbf{0} \wedge \mathbf{cs_t} = \mathbf{1} \wedge \mathbf{rw_t} = \mathbf{1}) \Rightarrow \\
& \quad \mathbf{d1_t} \wedge \mathbf{d0_t} = \text{ite}(\mathbf{rs_t} = \mathbf{1}, \mathbf{0}, (\text{ite}(\mathbf{cs_t} = \mathbf{1} \wedge \mathbf{rw_t} = \mathbf{1}, \mathbf{d0_t} \wedge \mathbf{d1_t}, \mathbf{m3_t}))) \\
i_4 \equiv \quad & (\mathbf{rs_t} = \mathbf{0} \wedge \mathbf{cs_t} = \mathbf{1} \wedge \mathbf{rw_t} = \mathbf{1}) \Rightarrow \\
& \quad \mathbf{d0_t} \wedge \mathbf{d1_t} = \text{ite}(\mathbf{rs_t} = \mathbf{1}, \mathbf{0}, (\text{ite}(\mathbf{cs_t} = \mathbf{1} \wedge \mathbf{rw_t} = \mathbf{1}, \mathbf{d2_t} \wedge \mathbf{d0_t}, \mathbf{m4_t}))) \\
i_5 \equiv \quad & (\mathbf{rs_t} = \mathbf{0} \wedge \mathbf{cs_t} = \mathbf{1} \wedge \mathbf{rw_t} = \mathbf{1}) \Rightarrow \\
& \quad \mathbf{d1_t} \wedge \mathbf{d2_t} = \text{ite}(\mathbf{rs_t} = \mathbf{1}, \mathbf{0}, (\text{ite}(\mathbf{cs_t} = \mathbf{1} \wedge \mathbf{rw_t} = \mathbf{1}, \mathbf{d1_t} \wedge \mathbf{d1_t}, \mathbf{m5_t})))
\end{aligned}
$$

Now we consider structural symmetries of the reduced terms above. Instead of just stating the result we give an argument on the meta level here. An approximation of the symmetry relation can be constructed by hand as follows. Because of the modulo operation, the term t_0 evaluates to **d0** and **d1** with periodicity of 2. The terms t_1 and t'_1 change with periodicity 3 and 6 respectively. Because the lowest common multiple of 2, 3 and 6 is 6, there are 6 possible combinations of variables **d0**, **d1** and **d2**. The variables $\mathbf{m0_t}, \mathbf{m1_t}, \ldots, \mathbf{m255_t}$ occur only once in each of the instances. This is shown in Table 10.3. Renaming all the variables $\mathbf{ma_t}$ in the reduced terms i_a with $\mathbf{m0_t}$. (Rename $\mathbf{m1_t}$ in i_1, $\mathbf{m2_t}$ in i_2, ..., and $\mathbf{m255_t}$ in i_{255} with $\mathbf{m0_t}$) we get a symmetry relation according to partition A below.

$$
\begin{aligned}
A = \{ \quad & \{\mathbf{0}, \mathbf{6}, \mathbf{12}, \ldots, \mathbf{252}\}, \\
& \{\mathbf{1}, \mathbf{7}, \mathbf{13}, \ldots, \mathbf{253}\}, \\
& \{\mathbf{2}, \mathbf{8}, \mathbf{14}, \ldots, \mathbf{254}\}, \\
& \{\mathbf{3}, \mathbf{9}, \mathbf{15}, \ldots, \mathbf{255}\}, \\
& \{\mathbf{4}, \mathbf{10}, \mathbf{16}, \ldots, \mathbf{250}\}, \\
& \{\mathbf{5}, \mathbf{11}, \mathbf{17}, \ldots, \mathbf{251}\} \quad \}
\end{aligned}
$$

W.l.o.g. the first 6 terms are considered further. The terms i_0 and i_1 are syntactically symmetrical, e.g. rename $\mathbf{d1} \rightarrow \mathbf{d0}$, and $\mathbf{m0} \rightarrow \mathbf{m1}$ in i_1. The same holds for i_2, i_3. (Rename $\mathbf{d1} \rightarrow \mathbf{d0}$, $\mathbf{d0} \rightarrow \mathbf{d2}$, $\mathbf{m2} \rightarrow \mathbf{m3}$ in i_3!) The cofactor terms i_4 and i_5 are not

Table 10.3: Possible Combination of Variables

$\mathbf{a}_\mathsf{t}$	$t_{0,\mathsf{t}}$	$t_{1,\mathsf{t}}$	$t'_{1',\mathsf{t}}$	$\mathbf{ma}_\mathsf{t}$
0	$\mathbf{d0}_\mathsf{t}$	$\mathbf{d0}_\mathsf{t}$	$\mathbf{d0}_\mathsf{t}$	$\mathbf{m0}_\mathsf{t}$
1	$\mathbf{d1}_\mathsf{t}$	$\mathbf{d1}_\mathsf{t}$	$\mathbf{d1}_\mathsf{t}$	$\mathbf{m1}_\mathsf{t}$
2	$\mathbf{d0}_\mathsf{t}$	$\mathbf{d2}_\mathsf{t}$	$\mathbf{d2}_\mathsf{t}$	$\mathbf{m2}_\mathsf{t}$
3	$\mathbf{d1}_\mathsf{t}$	$\mathbf{d0}_\mathsf{t}$	$\mathbf{d0}_\mathsf{t}$	$\mathbf{m3}_\mathsf{t}$
4	$\mathbf{d0}_\mathsf{t}$	$\mathbf{d1}_\mathsf{t}$	$\mathbf{d2}_\mathsf{t}$	$\mathbf{m4}_\mathsf{t}$
5	$\mathbf{d1}_\mathsf{t}$	$\mathbf{d2}_\mathsf{t}$	$\mathbf{d1}_\mathsf{t}$	$\mathbf{m5}_\mathsf{t}$
6	$\mathbf{d0}_\mathsf{t}$	$\mathbf{d0}_\mathsf{t}$	$\mathbf{d0}_\mathsf{t}$	$\mathbf{m6}_\mathsf{t}$
7	$\mathbf{d1}_\mathsf{t}$	$\mathbf{d1}_\mathsf{t}$	$\mathbf{d1}_\mathsf{t}$	$\mathbf{m7}_\mathsf{t}$
8	$\mathbf{d0}_\mathsf{t}$	$\mathbf{d2}_\mathsf{t}$	$\mathbf{d2}_\mathsf{t}$	$\mathbf{m8}_\mathsf{t}$
9	$\mathbf{d1}_\mathsf{t}$	$\mathbf{d0}_\mathsf{t}$	$\mathbf{d0}_\mathsf{t}$	$\mathbf{m9}_\mathsf{t}$
10	$\mathbf{d0}_\mathsf{t}$	$\mathbf{d1}_\mathsf{t}$	$\mathbf{d2}_\mathsf{t}$	$\mathbf{m10}_\mathsf{t}$
11	$\mathbf{d0}_\mathsf{t}$	$\mathbf{d2}_\mathsf{t}$	$\mathbf{d1}_\mathsf{t}$	$\mathbf{m11}_\mathsf{t}$
...	...	...	...	

syntactically symmetric to any other. The resulting approximate symmetry relation is A' below:

$$\begin{aligned} A' = \{ \quad & \{\mathbf{0}, \mathbf{1}, \mathbf{6}, \mathbf{7}, \ldots, \mathbf{252}, \mathbf{253}\}, \\ & \{\mathbf{2}, \mathbf{3}, \mathbf{8}, \mathbf{9}, \ldots, \mathbf{254}, \mathbf{255}\}, \\ & \{\mathbf{4}, \mathbf{10}, \mathbf{16}, \ldots, \mathbf{250}\}, \\ & \{\mathbf{5}, \mathbf{11}, \mathbf{17}, \ldots, \mathbf{251}\} \qquad \} \end{aligned}$$

There are just 4 equivalence classes and thus just 4 terms (e.g. i_0, i_2, i_4 and i_5) have to be considered. Here i_4 and i_5 are invalid.

10.3 Extended Symmetry Reduction

Now the symmetry reduction scheme described before is extended to make it more powerful by combining symmetry relations on different variables. First the limitations of the approach described before are illustrated. Then it is shown how extended symmetry relations can be used for reduction and how symmetry relations on different variables can be combined to extended symmetry relations. Finally this idea is transformed into an algorithm which can be used for preprocessing in RTL-BMC.

10.3.1 Limitations of the Basic Reduction Scheme

The symmetry relations found for a variable depends on the variables that have been removed before. This is due to the following fact.

Let t be a bitvector term and let $\mathbf{x}, \mathbf{y} \in Var(t)$ be variables occurring in t. Let $\sim$ be an equivalence relation on terms inducing an approximate symmetry relation on cofactor terms. (Take for example $\approx_E$, $\approx_S$ or $\approx_{ES}$.) Furthermore let $\mathbf{a}_1, \mathbf{a}_2 \in Dom(\mathbf{x})$, and let $\mathbf{b}_1, \mathbf{b}_2 \in Dom(\mathbf{y})$ be values for $\mathbf{x}$ and $\mathbf{y}$ in t, respectively.

1. If $t[\mathbf{x}/\mathbf{a}_1] \sim t[\mathbf{x}/\mathbf{a}_2]$, and $t[\mathbf{y}/\mathbf{b}_1] \sim t[\mathbf{y}/\mathbf{b}_2]$, then $\mathbf{a}_1 \sim_{t,\mathbf{x}} \mathbf{a}_2$ and $\mathbf{b}_1 \sim_{t,\mathbf{y}} \mathbf{b}_2$ follows. However, this does not imply that $t[\mathbf{x}/\mathbf{a}_1][\mathbf{y}/\mathbf{b}_1] \sim t[\mathbf{x}/\mathbf{a}_1][\mathbf{y}/\mathbf{b}_2]$ holds, and hence $\mathbf{b}_1 \sim_{t[\mathbf{x}/\mathbf{a}_1],\mathbf{y}} \mathbf{b}_2$ cannot be concluded. Alg. 4 will reduce the problem to two cofactor, regardless in which order the cofactor terms are considered. (We will see later that this can be avoided.)

2. Conversely, if $t[\mathbf{x}/\mathbf{a}_1] \not\sim t[\mathbf{x}/\mathbf{a}_2]$, and $t[\mathbf{y}/\mathbf{b}_1] \not\sim t[\mathbf{y}/\mathbf{b}_2]$, then $t[\mathbf{x}/\mathbf{a}_1][\mathbf{y}/\mathbf{b}_1] \sim t[\mathbf{x}/\mathbf{a}_1]$ $[\mathbf{y}/\mathbf{b}_2]$ may still hold. This case leads to deficiencies since not all symmetries are found, again regardless in which order the cofactor are considered.

3. Finally, it is possible that $t[\mathbf{x}/\mathbf{a}_1] \not\sim t[\mathbf{x}/\mathbf{a}_2]$, and $t[\mathbf{y}/\mathbf{b}_1] \sim t[\mathbf{y}/\mathbf{b}_2]$, but $t[\mathbf{x}/\mathbf{a}_1][\mathbf{y}/\mathbf{b}_1]$ $\sim t[\mathbf{x}/\mathbf{a}_1][\mathbf{y}/\mathbf{b}_2]$. This case may lead to deficiencies depending on the order in which the cofactors are considered.

Therefore the number of cofactor terms finally considered by Alg. 4 is larger than necessary, and may depend on the order, in which the variables are selected. Consider the following example which illustrates this.

Example 10.27
Let E be a set of identities as below.

$$E = \{\text{ite}(\mathbf{c}, \mathbf{a}, \mathbf{a}) = \mathbf{a},\ \text{ite}(\mathbf{0}, \mathbf{a}, \mathbf{b}) = \mathbf{b},\ \text{ite}(\mathbf{1}, \mathbf{a}, \mathbf{b}) = \mathbf{a}\}$$

1. Consider the bitvector term t

$$t \equiv \text{ite}(\mathbf{x}, \text{ite}(\mathbf{y}, \mathbf{w}_1, \mathbf{w}_2), \text{ite}(\mathbf{z}, \mathbf{w}_3, \mathbf{w}_4))$$

   Then $\mathbf{1}$ and $\mathbf{0}$ are symmetrical values for $\mathbf{x}$ in t, as well as for $\mathbf{y}$, since $t[\mathbf{x}/\mathbf{1}] \approx_{ES} t[\mathbf{x}/\mathbf{0}]$, and $t[\mathbf{y}/\mathbf{1}] \approx_{ES} t[\mathbf{y}/\mathbf{0}]$:

$$\begin{array}{rcl} E \vdash t[\mathbf{x}/\mathbf{1}] & = & \text{ite}(\mathbf{y}, \mathbf{w}_1, \mathbf{w}_2) \\ E \vdash t[\mathbf{x}/\mathbf{0}] & = & \text{ite}(\mathbf{z}, \mathbf{w}_3, \mathbf{w}_4) \\ E \vdash t[\mathbf{y}/\mathbf{1}] & = & \text{ite}(\mathbf{x}, \mathbf{w}_1, \text{ite}(\mathbf{z}, \mathbf{w}_3, \mathbf{w}_4)) \\ E \vdash t[\mathbf{y}/\mathbf{0}] & = & \text{ite}(\mathbf{x}, \mathbf{w}_2, \text{ite}(\mathbf{z}, \mathbf{w}_3, \mathbf{w}_4)) \end{array}$$

   Since both pairs of cofactor terms are equivalent, Alg. 4 will reduce the problem to one cofactor, regardless of the order the cofactor terms are considered. But $\mathbf{1}$ and $\mathbf{0}$ are neither symmetrical values for $\mathbf{x}$ in $t[\mathbf{y}/\mathbf{1}]$ nor for $\mathbf{x}$ in $t[\mathbf{y}/\mathbf{0}]$, since:

$$\begin{array}{rcl} E \vdash t[\mathbf{y}/\mathbf{1}][\mathbf{x}/\mathbf{1}] & = & \mathbf{w}_1 \\ E \vdash t[\mathbf{y}/\mathbf{1}][\mathbf{x}/\mathbf{0}] & = & \text{ite}(\mathbf{z}, \mathbf{w}_3, \mathbf{w}_4) \\ E \vdash t[\mathbf{y}/\mathbf{0}][\mathbf{x}/\mathbf{1}] & = & \mathbf{w}_2 \\ E \vdash t[\mathbf{y}/\mathbf{0}][\mathbf{x}/\mathbf{0}] & = & \text{ite}(\mathbf{z}, \mathbf{w}_3, \mathbf{w}_4) \end{array}$$

   Therefore two cofactors remain in both cases.

2. Consider the bitvector term t'.

$$t' \equiv \text{ite}(\mathbf{x}, \text{ite}(\mathbf{y}, \mathbf{w}_1, \text{ite}(\mathbf{x}, \mathbf{w}_2, \mathbf{w}_3)), \mathbf{w}_4)$$

Then $t[\mathbf{x}/\mathbf{1}] \neq_{ES} t[\mathbf{x}/\mathbf{0}]$, and $t[\mathbf{y}/\mathbf{1}] \neq_{ES} t[\mathbf{y}/\mathbf{0}]$ as shown below:

$$\begin{array}{rcl} E \vdash t'[\mathbf{x}/\mathbf{1}] & = & \text{ite}(\mathbf{y}, \mathbf{w}_1, \mathbf{w}_2) \\ E \vdash t'[\mathbf{x}/\mathbf{0}] & = & \mathbf{w}_3 \\ E \vdash t'[\mathbf{y}/\mathbf{1}] & = & \text{ite}(\mathbf{x}, \mathbf{w}_1, \mathbf{w}_4) \\ E \vdash t'[\mathbf{y}/\mathbf{0}] & = & \text{ite}(\mathbf{x}, \text{ite}(\mathbf{x}, \mathbf{w}_2, \mathbf{w}_3), \mathbf{w}_4) \end{array}$$

But the cofactors terms $t'[\mathbf{y}/\mathbf{1}][\mathbf{x}/\mathbf{1}]$, $t'[\mathbf{y}/\mathbf{1}][\mathbf{x}/\mathbf{0}]$, $t'[\mathbf{y}/\mathbf{0}][\mathbf{x}/\mathbf{1}]$ and $t'[\mathbf{y}/\mathbf{0}][\mathbf{x}/\mathbf{0}]$ are all pairwise equivalent. The cofactors terms $t'[\mathbf{x}/\mathbf{1}][\mathbf{y}/\mathbf{1}]$, $t'[\mathbf{x}/\mathbf{1}][\mathbf{y}/\mathbf{0}]$, $t'[\mathbf{x}/\mathbf{0}][\mathbf{y}/\mathbf{1}]$ and $t'[\mathbf{x}/\mathbf{0}][\mathbf{y}/\mathbf{0}]$ are also all pairwise equivalent. If Alg. 4 chooses to reduce t first w.r.t. $\mathbf{y}$, the cofactors are not equivalent and it will continue considering them separately such that 2 cofactor terms remain. The same holds if t is reduced first w.r.t. $\mathbf{x}$, independently of the order chosen. However, it would be desirable to end up with one remaining cofactor only.

3. Consider the bitvector term t''

$$t' \equiv \text{ite}(\mathbf{x}, \text{ite}(\mathbf{y}, \mathbf{w}_1, \mathbf{w}_2), \mathbf{w}_4)$$

Then $t''[\mathbf{x}/\mathbf{1}] \neq_{ES} t''[\mathbf{x}/\mathbf{0}]$, but $t''[\mathbf{x}/\mathbf{1}] \approx_{ES} t''[\mathbf{x}/\mathbf{0}]$

$$\begin{array}{rcl} E \vdash t'[\mathbf{x}/\mathbf{1}] & = & \text{ite}(\mathbf{y}, \mathbf{w}_1, \mathbf{w}_2) \\ E \vdash t'[\mathbf{x}/\mathbf{0}] & = & \mathbf{w}_4 \\ E \vdash t'[\mathbf{y}/\mathbf{1}] & = & \text{ite}(\mathbf{x}, \mathbf{w}_1, \mathbf{w}_4) \\ E \vdash t'[\mathbf{y}/\mathbf{0}] & = & \text{ite}(\mathbf{x}, \mathbf{w}_2, \mathbf{w}_4) \end{array}$$

Now depending on the order in which the cofactors are considered, there are finally either one or two cofactors that have to be treated further.

10.3.2 Using Extended Symmetry Relations

Instead of considering the remaining cofactors for one variable separately, they can be left together, removing the next variable in all of them and computing an extended symmetry relation.

Example 10.28
In the example above the extended symmetry relations computed for (x, y) in t, t' and t'' are $\sim_P$, $\sim_{P'}$ and $\sim_{P''}$ given by the partitions P, P', and P'' of $\mathbb{B}^2$ below.

$$\begin{array}{rcl} P & = & \{\{00, 01\}, \{10, 11\}\} \\ P' & = & \{\{00, 01, 10, 11\}\} \\ P'' & = & \{\{00, 01, 10, 11\}\} \end{array}$$

Now, independently of the variable order, in the two latter cases only one cofactor remains to be checked due to Lemma 10.18.

Still the result is not completely satisfactory, as in the example above, still two cofactors remain to be considered. Next we will show that it is possible to join these cases and to consider only one class.

10.3.3 Combining Symmetry Relations on Different Variables

Consider a Boolean function f in variables x and y (and possibly further variables). Suppose we know that $\sim_{f,x}$ and $\sim_{f,y}$ are symmetry relations and $0 \sim_{f,x} 1$ and $0 \sim_{f,y} 1$ (as for t in the example above.) Then we know the following:

$$\begin{array}{lllll} f = 1 & \Leftrightarrow & f_x = 1 & \Leftrightarrow & f_{\bar{x}} = 1 \\ f = 1 & \Leftrightarrow & f_y = 1 & \Leftrightarrow & f_{\bar{y}} = 1 \end{array}$$

It follows that $f_{xy} = f_{yx}$, $f_{\bar{x}y} = f_{y\bar{x}}$, $f_{x\bar{y}} = f_{\bar{y}x}$, $f_{\bar{x}\bar{y}} = f_{\bar{y}\bar{x}}$, and we conclude:

$$f = 1 \Leftrightarrow f_{xy} = 1$$

Generalizing this idea to bitvector functions yields the following result.

Lemma 10.29
Let $f : \mathbb{B}^{n_1} \times \ldots \times \mathbb{B}^{n_m} \rightarrow \mathbb{B}^{n_f}$ be a bitvector function in variables $X = \{\mathbf{x}_1, \ldots, \mathbf{x}_m\}$. Let $X_k \subseteq X$ with $X_k = \{\mathbf{x}_1, \ldots, \mathbf{x}_k\}$, Let $P_1, \ldots, P_k$ be partitions of $Dom(\mathbf{x}_1), \ldots Dom(\mathbf{x}_k)$ such that $\sim_{P_1}, \ldots, \sim_{P_k}$ are symmetry relations for f in $\mathbf{x}_1, \ldots, \mathbf{x}_k$ respectively. Let $R_1, \ldots, R_k$ be sets of representatives for each equivalence class, respectively, then

$$f = 1 \;\Leftrightarrow\; \forall \mathbf{a}_1 \in R_1, \ldots, \forall \mathbf{a}_k \in R_k : f_{\mathbf{x}_1 = \mathbf{a}_1 \cdots \mathbf{x}_k = \mathbf{a}_k} = 1 \tag{10.10}$$

(Note that the equivalence relation $\sim_D$ on $D := Dom(\mathbf{x}_1) \times \cdots \times Dom(\mathbf{x}_k)$ defined as $(\mathbf{a}_1, \ldots, \mathbf{a}_k) \sim_D (\mathbf{b}_1, \ldots, \mathbf{b}_k)$ iff $\mathbf{a}_1 \sim_{P_1} \mathbf{b}_1$, $\ldots$,$\mathbf{a}_k \sim_{P_1} \mathbf{b}_k$ is no longer an extended symmetry relation for $(\mathbf{x}_1, \ldots, \mathbf{x}_k)$ in f.)

Corollary 10.30
Let t be a bitvector term of sort 1 over a set of bitvector variables X. Let $\mathbf{x}, \mathbf{y} \in Var(t)$ be bitvector variables in t. Let $P = \{P_1, \ldots, P_k\}$, and $Q = \{Q_1, \ldots, Q_l\}$ be partitions of the domains of $\mathbf{x}$ and $\mathbf{y}$ such that $\sim_P$ and $\sim_Q$ are symmetry relations for $\mathbf{x}$, $\mathbf{y}$ in t, respectively. Let $A = \{p_1, \ldots, p_k\}$ and $B = \{q_1, \ldots, q_l\}$ be sets of values such that $p_i \in P_i$ and $q_j \in Q_j$, then

$$\mathcal{A}_{\mathcal{BV}} \models t = 1 \;\Leftrightarrow\; \forall a \in A \forall b \in B : \mathcal{A}_{\mathcal{BV}} \models t[x/a][y/b] = 1$$

10.3.4 A Practical Reduction Algorithm

Let t be a bitvector term of sort 1, for which $t = 1$ has to be shown. Instead of generating the cofactor terms for one variable in a term, and then finding a symmetry relation, and finally considering the variables in the remaining instances (as in Algorithm 4), the idea is now the following.

Assume, there is a complete decision procedure '`taut`' deciding for t, whether $t = 1$ holds. Further assume there is an algorithm '`symRelA`' to compute approximate symmetry relations. We want to use algorithm '`symRelA`' as preprocessing procedure for '`taut`'.

First a set of candidate variables X is selected from the variables occurring the term t. Obviously this set should be selected, such that algorithm '`symRelA`' is likely to find large symmetry relations for each variable[4]

[4]The heuristics used for experiments in the next chapter is very simple. It selects all so called 'for'-variables of the property as candidates. Thus it just leaves the problem to the user who has to provide the property in the appropriate syntax.

Then the approximate symmetry relation for each of the variables in X in t is computed. Now for each variable $x \in X$ the domain is reduced, such that it contains only one value for each equivalence class in the respective symmetry relation.

If all possible combinations of the remaining values for each variable in X the cofactors t' are considered and $t' = 1$ holds for each of them, then $t = 1$ holds in general as well, according to the corollary above. If for one combination of values $t' = 1$ doesn't hold, then, of course, $t = 1$ doesn't hold either.

Now, instead of generating all cofactor terms for each $x \in X$ at once, the variable with the smallest domain is selected first, all cofactor terms for this variable are generated, and its domain is reduced. Then the next variable is selected and the process is repeated for all remaining variables (always starting with t). Thus for each $x \in X$ the domains are reduced independently. Then we may branch on a variable (with the smallest remaining domain) and consider all resulting cofactor terms as new starting points for reduction, i.e. as new t's. Before branching on the next variable, we try to merge branches by checking cofactor terms from different branches for permutation equivalence, in order to use the extended symmetry relations implied. When the last variable has been removed, the remaining cofactor terms do not contain variables of X anymore. The remaining cofactor terms are then checked for constantness using algorithm '`taut`' individually.

The approach described above is used in the practical Algorithm 6 for tautology checking. It takes a term `tb` to be proven constantly 1 and a set of variables `X` which are suspected to have symmetrical values. It uses the Algorithms 7, 8, 9, and 5. First the map of variables to their reduced domains `VD` is initialized with the full domains for each variable in `X`. Then a set of reduced terms `F` is computed using `reduce` (Algorithm 7). These terms do not contain the variables `X` anymore, since they have been instantiated with domain values during the run of `reduce` (Algorithm 7). If any of these terms is not constant, then the term `tb` is not constant. If all of them are constant, so is `tb`. (Note that in the case that `tb` is not constantly 1, some additional bookkeeping on the cofactor terms allows to generate a counter-example as assignment for the variables in `tb`.) The Algorithms 7, 8, and 9 are explained below.

Algorithm 6 Practical SAT Using Approximate Symmetry Relations

```
Boolean search4 (Term tb, Variables X)
{
  map<Variable, Values> VD;          // Variables with reduced domains.
  foreach (Variable x in X)          // Initialize the map with
    VD[x]=Dom(x);                    // the full domains of variables.
  Terms TR;                          // Remaining terms to check.
  TR = reduce(tb, X, VD);            // Compute reduced terms of tb.
  foreach (Term tr in TR)            // For each remaining term
    if (taut(tr) == false)           // check if it is constantly 1.
       return false;                 // If one term is not, so is tb.
  return true;                       // If all cofactor terms are, tb=1.
}
```

Algorithm 7 computes the symmetry relation for each variable in `X` and reduces the domain of each variable accordingly. Then a variable with the smallest domain is selected

using Algorithm 8 and Algorithm 9 is used to split on this variable and reduce the remaining cofactor terms. Algorithm 8 selects a variable from a set of variables `X` which has

Algorithm 7 Reduce Number of Cofactor Terms

```
Terms reduce(Term t, Variables X, map<Variable, Values> VD)
{
  Terms T;                              // Set of remaining terms.
  if (X.empty())                        // If no variables are left,
    T.insert(t);                        // the remaining term is t
  else                                  // Otherwise:
  {
    foreach ( Variable x in X )         // Compute symmetry relation
    {                                   // for each variable.
      map<Term,Value> TC;               // Map cofactor terms to the value.
      Partition Term P;                 // Represents symmetry relation.
      foreach ( Value c in VD[x] )      // For each value in the reduced
      {                                 // domain for x
        Term tc;                        // generate a cofactor term.
        tc =  substitute(t,x,c);
        TC[tc] = c;                     // Remember corresponding value
        P.insert(tc);                   // which is a singleton first.
      }
      symRelAC(P);                      // Compute the symmetry relation.
      foreach ( Value c in VD[x] )      // If a value in the domain of x
        if ( c != TC[P.find(c)] )       // does not represent its equiv.
          VD[x].remove(c);              // class, then remove it from the
    }                                   // Domain of x.
    Variable x = selectVariable(X, VD); // Select a variable with small
    T = splitAndCombine(t, X, VD, x);   // domain, reduce each term.
  }
  return T;                             // Return the remaining terms.
}
```

a smallest domain in `VD`.

Algorithm 9 generates all cofactor terms for the remaining values of a variable `x` and reduces these terms using Algorithm 7. The remaining terms do not contain variables of `X` anymore. Before returning the resulting union of terms, the remaining terms from different branches are checked for symmetry again.

10.3.5 Input Variable Splitting

The following optimization has to be applied in conjunction with the extended symmetry reduction technique described above, to make the approach work practically.

Let t be a bitvector term of width 1, and let $\mathbf{x}$ be a bitvector variable of width w occurring in t. Let t' be the corresponding Boolean term, and let $x_1, \ldots, x_w$ be Boolean variables corresponding to $\mathbf{x}$. If t' has symmetrical value vectors for the vector of variables $(x_1, \ldots, x_v)$ with $v < w$, this fact can not be detected or exploited on the bitvector-level by the techniques described before. The problem is illustrated in the example below.

Algorithm 8 Select Variable With Smallest Domain

```
Variable selectVariable( Variables X, map<Variable, Values> VD )
{
  Variable x;
  unsigned mD = 0;
  foreach ( Variable y in X )
    if ( ( mD = 0 ) or
          ( VD[y].size() < mD ) )
    {
      mD = VD[y].size();
      x = y;
    }
  return x;
}
```

Algorithm 9 Split on Variable and Reduce Terms

```
Terms splitAndCombine( Term t, Variables X,
                       map<Variable, Values> VD, Variable x)
{
  Values Dx = VD[x];                  // Remaining domain of x.
  X.remove(x);                        // No longer consider x.
  VD.remove(x);
  map<Value, Terms> TC;               // Domain values and cofact. terms.
  foreach ( Value c in Dx )           // Generate a cofactor term for
  {                                   // each remaining value of x.
    Term tc = substitute(t,x,c);      // Reduce each term, and
    TC[c] = reduce(tc, X, VD);        // get the remaining terms
  }                                   // for each value.
  if ( Dx.size() == 1 )               // If there is just one value
    return TC[Dx.choose()];           // return the remaining term.
  Partition<Term> P;                  // Otherwise try to match terms
  foreach ( Value c in Dx )           // for different values of x.
    foreach ( Term tc in TC[c] )      // Initialize the Partition of
      P.insert(tc);                   // remaining terms and
  symRelAC(P);                        // compute a symmetry relation.
  return P.reps();                    // Only the representatives are
}                                     // to be considered.
```

Example 10.31
Consider the memory from 8.2.1. It has 256 memory cells of width 16. The vector of state variables is $\overline{s} = (\mathbf{m255}_{[16]}, \mathbf{m254}_{[16]}, \ldots, \mathbf{m1}_{[16]}, \mathbf{m0}_{[16]})$, where $\mathbf{mi}_{[16]}$ denotes the i'th memory cell. They could be represented by a single bitvector variable $\mathbf{m}_{[256\times16]}$ of width 256×16, where $\mathbf{m}_{[256\times16]} = \mathbf{m255}\otimes\cdots\otimes\mathbf{m1}\otimes\mathbf{m0}$. Then the output and transition functions can be represented by the following (extended) bitvector terms.

$$t_O \equiv \text{ite}(\mathbf{rs} = \mathbf{1}, \mathbf{0}_{[16]}, \text{ite}(\mathbf{cs} = \mathbf{1} \wedge \mathbf{rw} = \mathbf{0}, \text{read}(\mathbf{m}, \mathbf{a}), \mathbf{0}_{[16]}))$$

$$t_S \equiv \quad \text{ite}(\mathbf{rs} = \mathbf{1}, \mathbf{0}_{[256\times16]}, \text{ite}(\mathbf{cs} = \mathbf{1} \wedge \mathbf{rw} = \mathbf{1}, \text{write}(\mathbf{m}, \mathbf{a}, \mathbf{di}), \mathbf{m}))$$

Then the write property reads as follows:

$$\begin{aligned} p' \equiv \quad & \mathbf{rs_t} = \mathbf{0} \wedge \mathbf{cs_t} = \mathbf{1} \wedge \mathbf{rw_t} = \mathbf{1} \Rightarrow \\ & \mathbf{di_t} = \text{read}(\text{ite}(\mathbf{rs_t} = \mathbf{1}, \mathbf{0}_{[256\times16]}, \text{ite}(\mathbf{cs_t} = \mathbf{1} \wedge \mathbf{rw_t} = \mathbf{1}, \text{write}(\mathbf{m_t}, \mathbf{a_t}, \mathbf{di_t}), \mathbf{m_t})), \mathbf{a_t}) \end{aligned}$$

Now consider the following cofactors for $\mathbf{a_t} = \mathbf{0}$ and $\mathbf{a_t} = \mathbf{170}$.

$$\begin{aligned} p'[\mathbf{a_t}/\mathbf{0}] \equiv \quad & \mathbf{rs_t} = \mathbf{0} \wedge \mathbf{cs_t} = \mathbf{1} \wedge \mathbf{rw_t} = \mathbf{1} \Rightarrow \\ & \mathbf{di_t} = \text{read}(\text{ite}(\mathbf{rs_t} = \mathbf{1}, \mathbf{0}_{[256\times16]}, \text{ite}(\mathbf{cs_t} = \mathbf{1} \wedge \mathbf{rw_t} = \mathbf{1}, \\ & \qquad \text{write}(\mathbf{m_t}, \mathbf{0}, \mathbf{di_t}), \mathbf{m_t})), \mathbf{0}) \\ p'[\mathbf{a_t}/\mathbf{170}] \equiv \quad & \mathbf{rs_t} = \mathbf{0} \wedge \mathbf{cs_t} = \mathbf{1} \wedge \mathbf{rw_t} = \mathbf{1} \Rightarrow \\ & \mathbf{di_t} = \text{read}(\text{ite}(\mathbf{rs_t} = \mathbf{1}, \mathbf{0}_{[256\times16]}, \text{ite}(\mathbf{cs_t} = \mathbf{1} \wedge \mathbf{rw_t} = \mathbf{1}, \\ & \qquad \text{write}(\mathbf{m_t}, \mathbf{170}, \mathbf{di_t}), \mathbf{m_t})), \mathbf{170}) \end{aligned}$$

Applying constant propagation and normalization as before yields:

$$\begin{aligned} p''[\mathbf{a_t}/\mathbf{0}] \equiv \quad & \mathbf{rs_t} = \mathbf{0} \wedge \mathbf{cs_t} = \mathbf{1} \wedge \mathbf{rw_t} = \mathbf{1} \Rightarrow \\ & \mathbf{di_t} = \text{ite}(\mathbf{rs_t} = \mathbf{1}, \mathbf{0}_{[16]}, \text{ite}(\mathbf{cs_t} = \mathbf{1} \wedge \mathbf{rw_t} = \mathbf{1}, \mathbf{di_t}), \mathbf{m_t}[\mathbf{0}, \mathbf{16}]) \\ p''[\mathbf{a_t}/\mathbf{170}] \equiv \quad & \mathbf{rs_t} = \mathbf{0} \wedge \mathbf{cs_t} = \mathbf{1} \wedge \mathbf{rw_t} = \mathbf{1} \Rightarrow \\ & \mathbf{di_t} = \text{ite}(\mathbf{rs_t} = \mathbf{1}, \mathbf{0}_{[16]}, \text{ite}(\mathbf{cs_t} = \mathbf{1} \wedge \mathbf{rw_t} = \mathbf{1}, \mathbf{di_t}), \mathbf{m_t}[\mathbf{2720}, \mathbf{16}]) \end{aligned}$$

The normalized cofactor terms above are not syntactically symmetrical as they only differ in the subterms $\mathbf{m_t}[\mathbf{0}, \mathbf{16}]$ and $\mathbf{m_t}[\mathbf{2720}, \mathbf{16}]$ which are not syntactically symmetrical.

In the example above the problem can be solved by 'remembering' that $\mathbf{m_t}[\mathbf{0}, \mathbf{16}] = \mathbf{m0}$ and $\mathbf{m_t}[\mathbf{2720}, \mathbf{16}] = \mathbf{m170}$ and replacing them accordingly. In general the problem is solved by the following approach.

1. First cofactor terms are normalized pushing extractions down (towards variables) using the additional rewrite rules in Tables B.7 and B.8.

2. Then extractions of variables are replaced with fresh variables as in the example above. Doing so one has to make sure that overlaps are recognized and resolved accordingly. This is done by partitioning the the variables along the borders of extractions and introducing a fresh variable for each chunk. This does not change the represented Boolean function.

Using the extended symmetry reduction described in this section, experiments have been conducted on relevant examples, which are described in the next chapter.

Chapter 11

Application to Real Designs

The reduction algorithm described before has been integrated into an RTL-BMC-framework and was implemented prototypical. The size of RTL-BMC-problems is reduced before a Boolean SAT problem is generated. The whole framework has been implemented in C++, (from RTL front-end, Property reader, variable reduction, Boolean encoding, down to the counterexample output). For two large industrial Verilog designs, experiments have been run for 4 properties and show the potential of the reduction technique.

11.1 RtProp – The Experimental Platform

For evaluating the ideas presented in this thesis as well as an experimental platform for RTL-based BIMC the program *RtProp* has been developed. It takes a register transfer state machine (RSM) and an interval-temporal-logic (ITL) property as well as an optional clocking scheme as basic input.

The RSM is a constructive representation of a digital circuit as a Mealy machine on the register transfer level according to Definition 8.1. RSMs can be generated automatically from Verilog HDL descriptions using a commercial tool called *verilogRTL2rsm*. The optional clocking scheme allows to automatically compute a synchronous model of the design. The given ITL-property is also taken as input by the commercial BMC tool *gateprop* which makes it easy to try RtProp on relevant industrial examples. The property is first translated into a basic form removing syntactic sugar, such that it is an expression over input output and state variables of the design annotated with time points.

The output and transition functions of the design as well as property are represented by bitvector terms of extended syntax according to Definition 6.27. Bitvector terms are always represented by fully collapsed term graphs, maximally sharing common subexpressions. The normalization techniques described in Section 7.2 such as constant propagation are built into the term generation process and are implicitly applied when terms are constructed.

From design and property a bitvector term of sort 1 is constructed which is constantly 1 iff the property holds in all states of the design as described in Chapter 8.

Bitvector terms can be translated into vectors of Boolean terms as usual. Boolean terms can be checked for constantness using a commercial composite prover incorporating state of the art BDD and SAT techniques. (For a similar approach see [PK00].) If

counterexamples are generated they are automatically translated back to the RT level and presented to the user in textual form.

Different reduction techniques are implemented and can be applied to the problem before solving it on the Boolean level. For normalization a prototypical rewrite system was implemented. As graph automorphism engine an implementation of [McK81], optimized for sparse graphs, was integrated. Currently the following reductions are implemented.

1. Rewrite heuristics to be applied on top of the built-in normalization can be specified in form of rule specifications contained in an additional file. Two different rewrite strategies (top-down and bottom-up rewriting) can be selected and are combined. Rule specifications can be selected for application in either or both strategies. Rewriting is implemented as term graph rewriting on fully collapsed term graphs such that common subexpressions are only treated once. During top-down (bottom-up) rewriting subterms are normalized in reverse topological (topological) order until all subterms are in normal form.

2. Symmetry reduction can be performed for bitvector variables of a BMC-problem. Currently, candidate variables for reduction are selected within the property (using so called 'for'-variables of ITL, as described later). Different strategies can be selected by options, for example whether rewriting should be applied to the whole bitvector term before generating cofactor terms or not. During symmetry reduction equivalence classes for the values of these selected variables are subsequently improved. First simple heuristics are tried, if no more reduction is achieved more sophisticated and more expensive methods are applied.

All steps described above (from reading the input files, through generating the problem, applying reduction and giving a counterexample if appropriate) are seamlessly integrated into the tool and require no assistance by the user once the program has been started.

However, RtProp is also able to write textual problem representations at different stages of this process for offline analysis. Currently two bitvector and one Boolean term format are supported. (All of which are proprietary.) This allows for example the use of other reduction techniques as [Joh01] whose results are manually feed back into the design and property.

11.2 Experimental Results on Symmetry Reduction

Experiments on reduction of RTL-BMC-problems by virtue of symmetrical values have been conducted on some instructive examples from this thesis and on two industrial designs and their most important properties.

11.2.1 How to Read the Result Tables

All experiments were run on a 800 MHz Pentium 3 with 512 MB RAM under Linux 2.4.4. All reported times are CPU seconds, where times reported as < 1 are closer to 0s than to 1s. The reduction heuristics and the rewrite rules were the same on all experiments.

Consider Table 11.1. In the first column the problem is described by the name of design and property (Name), the number of time frames the property spans over (#TFs), and the sizes of the domains of the for-variables to be factored out (VD). For example in the third line of Table 11.1 the first column the design is 'RegFile', and the property is 'write3' which involves two time frames and two for-variables with domains 4 and 2, respectively.

In the second column the results for proving the problem without reduction are shown. The number of variables (#Vars) and and-gates (#Gates) of the and-inverter-graph representing the problem on the Boolean level are given as a rough guideline for the size of the problem. The main information is the time needed for proving the problem (T.Prov). Note that there is no strong correspondence between the size of the problem and its difficulty. However, problems tend to get heavier with growing sizes.

The third column shows the results of the reduction. The total time needed for solving the problem (T.tot.), i.e. for the reduction (T.red.) and for proving the remaining cofactors (T.prov.).

Information about the remaining cofactors can be found in the last column, i.e. the number of cofactors left to be proven after reduction (#Left), the individual size of a typical cofactor in terms of variables (#Vars) and the number of and-gates (#Gates), and the time needed to prove one cofactor (T.prov).

11.2.2 Examples from this Thesis

To show the relation between the small examples from this thesis and industrial cases, experiments have been run for the register file from Example 5.6, and the memory described in Section 8.2.1.

Example 11.1 (Register File)
The write property for register file from Example 5.6 has been checked in different variations. The initial property shown in Table 8.1 was modified such that it contains so called 'for'-variables in VLI. These 'for'-variables are taken as candidates for cofactor expansion by RtProp. Two different versions of the property are shown in Fig. 11.1. Thus none, one (`ci`), and two variables (`ci`, `it`) are considered for reduction when proving the three properties, respectively. The experimental results shown in Table 11.1 indicate that the resulting problem is almost trivial in all cases, and that the reduction succeeded to reduce all for-variables. (I.e. just one cofactor was left to be proven.)

Example 11.2 (256×16-Memory)
The memory from Section 8.2.1 can be parameterized both in the number of memory cells as well as in the number of bits per cell. The number bits per cell has been fixed to 16. Experiments have been run for configurations with 4, 16, 64, 256, and 512 cells. The Verilog-HDL source code of the memory is shown in Fig. 11.2. From the RSM representations of each configuration (`clk`-synchronous) models were generated and the write properties given in Fig. 11.3 have been checked for each of them. The variables `ax` and `ax`, `dx` have been selected for reduction, respectively. The experimental results are shown in Table 11.1.

First consider the instances for `write1` shown on the left of Figure 11.3 containing one for-variable. Reduction of the only for-variable succeeded in all cases leaving a trivial

Write Property with one 'for' Variable	Write Property with two 'for'-Variables

`theorem write2;`
`for: ci = 0..3;`
`freeze: it = i@t;`
`assume:`
`  at t:  c == ci;`
`  at t:  w == 1'b1;`
`prove:`
`  at t+1:`
`  m[ci] == it;`
`  at t:`
`  o == m[ci];`
`end theorem;`
	`theorem write3;`
	`for: ci = 0..3,`
	`  it = 0..1;`
	`assume:`
	`  at t:  c == ci;`
	`  at t:  i == it;`
	`  at t:  w == 1'b1;`
	`prove:`
	`  at t+1:`
	`  m[ci] == it;`
	`  at t:`
	`  o == m[ci];`
	`end theorem;`

Figure 11.1: Variations of the Write Property of the Register-File with 'for'-Variables

problem to solve even for the 512 cell case, where the prove time for the non-reduced problem increased considerably. However, the reduction time was too high to see a positive overall effect.

Now consider the test cases for `write2` shown on the right of Figure 11.3. Here two instead of one variable were reduced. The reduction times are comparable with `write1`. However, the reduced problems were even solved by the rewriting procedure leaving nothing for the Boolean prover. For the larger example the overall runtime was better than without reduction. For both properties the reduction time was mainly due to unsuccessful calls of the automorphism procedure. For the larger examples the considered graphs got relatively large (some hundred thousand nodes). Due to an implementation deficiency the procedure was unnecessarily called several times with identical graphs. This problem will be resolved in later versions.

Table 11.1: Experimental Results for Register File and Memory

Property			**Original Problem**			**Reduced Problem**			**Cofactors**			
Name	#TFs	VD	#Vars	#Gates	**T.prov**	**T.total**	T.red	T.prov	#Left	#Vars	#Gates	T.prov
RegFile_write	2	0	< 20	< 1000	0.01	0.01	0.00	0.01	1	< 20	< 1000	0.01
RegFile_write2	2	4	< 20	< 1000	< 0.01	0.03	0.03	< 0.01	1	< 20	< 1000	< 0.01
RegFile_write3	2	4/2	< 20	< 1000	< 0.01	0.02	0.02	< 0.01	1	< 20	< 1000	< 0.01
Memory2_write1	2	4	82	371	0.01	0.03	0.03	< 0.01	1	< 20	< 1000	< 0.01
Memory4_write1	2	16	276	1355	0.03	0.11	0.11	< 0.01	1	< 20	< 1000	< 0.01
Memory6_write1	2	64	1046	5291	0.14	0.84	0.84	< 0.01	1	< 20	< 1000	< 0.01
Memory8_write1	2	256	4120	21035	1.34	26.32	26.32	< 0.01	1	< 20	< 1000	< 0.01
Memory9_write1	2	512	8217	42027	181.00	228.00	228.00	< 0.01	1	< 20	< 1000	< 0.01
Memory2_write2	2	16/4	87	405	0.01	0.11	0.11	0.00	1	0	1	0.00
Memory4_write2	2	16/16	281	1389	0.03	0.22	0.22	0.00	1	0	1	0.00
Memory6_write2	2	16/64	1051	5325	0.17	1.05	1.05	0.00	1	0	1	0.00
Memory8_write2	2	16/256	4125	21069	113.00	26.92	26.92	0.00	1	0	1	0.00
Memory9_write2	2	16/512	8222	42061	478.00	227.00	227.00	0.00	1	0	1	0.00

```
`define MEM_WIDTH 16
`define MEM_DEPTH 8
`define MEM_CELLS 256

module memory8 (cs, rw, rs, a, di, do, clk);

   input cs;
   input rw;
   input rs;
   input [`MEM_DEPTH-1:0] a;
   input [`MEM_WIDTH-1:0] di;
   input                  clk;
   output [`MEM_WIDTH-1:0] do;
   reg [`MEM_WIDTH-1:0]    m [`MEM_CELLS-1:0];
   reg [`MEM_WIDTH-1:0]    o;

   integer                 i;

   always @(posedge clk)
      begin
         if (rs)
            begin
               for (i = 0; i<=`MEM_CELLS-1 ; i = i+1)
                  begin
                     m[i]=`MEM_WIDTH'd0;
                  end // for (i = 0; i<='MEM_CELLS-1 ; i = i+1)
               o = `MEM_WIDTH'd0;
            end
         else   // if (clk)
            if (cs)
               begin
                  if (rw)
                     m[a]=di;
                  else
                     o = m[a];
               end
            else
               if (! rw)
                  o = `MEM_WIDTH'd0;
      end // always @ (posedge clk or negedge reset)
      assign do = o;

endmodule // memory8
```

Figure 11.2: VERILOG-HDL Code of Memory

```
theorem write1;

for:
        ax = 0..MEM_CELLS-1;
freeze:
        dit = di@t;
assume:
        at t: rs == 1'b0;
        at t: cs == 1'b1;
        at t: rw == 1'b1;
        at t:  a == ax;
prove:
        at t+1:  m[ax] == dit;
end theorem;
```

```
theorem write2;

for:
        ax = 0..MEM_CELLS-1,
        dx = 0..MEM_WIDTH-1;
freeze:
        dixt = di[dx]@t;
assume:
        at t: rs == 1'b0;
        at t: cs == 1'b1;
        at t: rw == 1'b1;
        at t:  a == ax;
prove:
        at t+1:  m[ax][dx] == dixt;
end theorem;
```

Figure 11.3: Write Properties for Memory in VERILOG-ITL

11.2.3 Interrupt Arbiter

The first industrial design considered is a priority interrupt controller (ITC) of a 16 Bit synthesizable micro controller core (C166S). The ITC is an arbiter deciding which of the currently active interrupt sources should be served next by the CPU. In this arbitration scheme the CPU is the slave and its service is requested by some of up to 128 interrupt nodes (requesting masters). Different interrupt nodes may have different priorities between 0 and 127. Requesting interrupt nodes with higher priorities are served before lower priority nodes. The priority of each node is programmable. Furthermore each node may be enabled or disabled. The arbiter produces the arbitration result every 4 clock cycles. The arbiter selects the enabled and requesting interrupt node with the highest priority. It indicates the winning interrupt node requesting and its priority to the CPU. If the priority of this interrupt node is higher then the priority of the currently running program in the CPU, the program is suspended (interrupted) and the program (interrupt survive routine) belonging to the requesting interrupt node is started.

The design is specified by about 600 lines of Verilog code. The number of interrupt nodes is parameterizable from 2 to 128 in steps of powers of 2. The verification task is to ensure that the request of a slave is granted if it has the highest priority and is enabled. The first property is symmetric, since it should be irrelevant which slave has the highest priority (maximum). There are 4 properties, one for each of the configurations with 4, 8, 16 and 32 interrupt nodes each. The properties which were developed during a verification project run over 7 time steps. Making them suitable for reduction was not necessary since they already contained the for-variables needed for specifying the reduction candidates in RtProp. Each property has one variable with domain size 4, 8, 16 and 32, respectively, which are place holders for the winning interrupts.

The experimental results are shown in Table 11.2 under `Arbiter_active_i_4` through `Arbiter_active_i_32`. In the first three configurations the selected variables have been eliminated by symmetry reduction in the 3 first cases. In the last case the reduction procedure ran out of memory during rewriting most likely because of a blow up of the

hash table for normal forms. (This is an implementation issue which could not be resolved easily and would have required a lot of implementation effort.) In all cases the time needed for reduction was to large to get a positive overall effect. The long reduction times are solely due to the normalization procedure. The particular problem was that a rules for the maximum function could not be specified since such an operator is missing in the implementation. As result it had to be modeled by other bitvector operations (ite and <) which destroyed the symmetry present in the maximum function. Two other variations

Table 11.2: Experimental Results for Interrupt Arbiter

Property			**Original Problem**			**Reduced Problem**			**Cofactors**			
Name	#TFs	VD	#Vars	#Gates	**T.prov**	**T.total**	T.red	T.prov	#Left	#Vars	#Gates	T.prov
Arbiter_active_i_4	7	4	136	3486	2	7	6	1	1	134	3205	1
Arbiter_active_i_8	7	8	249	6399	5	65	64	1	1	246	5806	1
Arbiter_active_i_16	7	16	474	12231	15	1684	1680	4	1	470	11014	4
Arbiter_active_i_32	7	32	923	23895	49	-	MemEx	-	-	-	-	-
Arbiter_spec_active_4	1	4	38	600	< 1	2	1	1	2	36	336	< 1
Arbiter_spec_active_8	1	8	75	1264	8	4	3	1	2	72	740	1
Arbiter_spec_active_16	1	16	148	2592	27	12	9	3	2	144	1548	1
Arbiter_spec_active_32	1	32	293	5248	87	48	37	11	2	288	3164	6
Arbiter_1111_active_4	1	4	38	580	1	2	1	< 1	2	36	397	< 1
Arbiter_1111_active_8	1	8	75	1212	2	5	5	1	2	72	861	< 1
Arbiter_1111_active_16	1	16	148	2476	9	20	18	2	2	144	1789	1
Arbiter_1111_active_32	1	32	293	5004	35	98	90	8	2	288	3645	4

of the ITC (`Arbiter_spec`, `Arbiter_1111`) were designed as functional models. Both of them implement the same functionality as the original design, however, within one clock cycle instead of 4. The difference between them is in the structure of the function implementing the maximum level of all requesting and enabled interrupt nodes (chain vs. balanced tree and bitvector comparison vs. bit comparison). The experimental results are also shown in Table 11.2. Again the reduction times dominated the prove times, and time spent during reduction was mainly due to rewriting. However, in many cases proving the problems with reduction was faster than without reduction.

11.2.4 Tag-RAM of a CPU Core's Memory Cache

The second industrial design considered is a Tag-RAM as they are commonly used in memory caches. Its general purpose is briefly described here.

Systems containing a CPU require memory to store program and data. The amount of memory used may be quite large. However, not all memory is equal in terms of speed, power and space consumption as well as price. The principle of locality states that the memory locations nearby the current location are most likely to be accessed next, and memory locations accessed recently are most likely to be accessed again soon. Therefore the address space of the CPU is usually partitioned into blocks of words, and an address is divided into a block address and a word address. To avoid deficiencies while keeping the cost of the system low the principle of locality suggests to cache recently used memory locations and to organize the memory in a hierarchy of increasingly larger, slower, and cheaper cache memories. A cache contains some but not all blocks of the main memory in its cache lines. If the CPU accesses a certain memory location the system tries the closest location to the CPU first (cache lookup).

The block address is divided into the tag (some of the MSBs) and the index (some of the LSBs of the block address). The cache consists of a memory containing the actual

data (data store) and the Tag-RAM. It stores the data belonging to a block in a cache line of the data store under the index of the block address. The Tag-RAM identifies which block from main memory is currently stored in each cache line. The values stored in the Tag-RAM determine whether a cache lookup results in a hit or a miss. If the tag under the index is the same as the tag of the current block address then a there is a hit otherwise a miss. If a cache hit occurred the block can be found under the index in the data store of the cache.

There are different ways of organizing caches, associative, direct mapped and block set associative. The design considered here is a 256x55-bit 2 way set associative Tag-RAM. It consists of about 500 lines of Verilog source code and could not be scaled. It contains a lot of control logic to organize the data transfers and some arithmetic for determining whether a hit of miss occurred. It was verified that the read and write operations work correctly and the content of the tag memory does not change if the memory is not enabled. The properties are parameterizable in the number of memory cells, between 2 and 256. The properties are specified over 8 time steps. Up to 6 variables have been selected for elimination. Experimental results are shown in Table 11.3. The advantages of the reduction approach are tremendous for all three properties. Only the last lines for each property are of significance in practice since they represent the problems resulting from the original properties. The other lines represent instances where the properties have been scaled down, leaving parts of the original problem uncovered. However, they are still useful to see how the reduction procedure scales up.

The first property shown is `cs` which ensures that the memory content does not change unwanted. For the configurations proving this property for 2 to 32 addresses (address input values) and 2 to 32 memory cells all cofactors were collapsed to a single one which was proven almost in no time. In the other cases not all cofactors were identified leaving 64, 128 and 256 small problems. The reduction time increased with the number of cofactors, except the transition from 128 to 256. Here the reduction procedure used a different strategy due to the large number of cofactors (It gave up earlier on some reductions). Proving the property without reduction required more than three hours CPU-time. Using reduction this time was reduced to about 7 minutes.

The situation is comparable for the other properties. The last property has been proven for the first time using reduction (however, spending 14 hours CPU time). The remaining cofactors are often symmetrical for different configurations since just the property was scaled up.

Note that proving many of the reduced properties was intractable for BIMC at the time this thesis was started. Only due to improvements on the SAT-solver implementation using newest results [MMZ+01] the ratio changed somewhat in favor of not reducing the problem before solving it. However, this effect was not strong enough. Still the reduction approach leads to a speedup by one to two orders of magnitude in relevant cases, proving a property intractable for the standard approach even now.

11.2.5 Interpretation of the Results

Proving some of the industrial cases has been intractable for the commercial BMC-tool gateprop at the time this thesis was started. Due to the tremendous improvements made on Boolean SAT solvers in the mean time this situation changed. Properties or the first

Table 11.3: Experimental Results Tag-RAM

Property			**Original Problem**			**Reduced Problem**			**Cofactors**			
Name	#TFs	VD	#Vars	#Gates	**T.prov**	**T.tot.**	T.red	T.prov	#Left	#Vars	#Gates	T.prov
Tag-RAM_cs_2	4	2/2	3565	10575	9	2	1	< 1	1	92	1642	< 1
Tag-RAM_cs_4	4	4/4	3567	13662	8	2	2	< 1	1	92	1642	< 1
Tag-RAM_cs_8	4	8/8	3569	19833	14	3	2	< 1	1	92	1642	< 1
Tag-RAM_cs_16	4	16/16	3571	32172	40	4	4	< 1	1	92	1642	< 1
Tag-RAM_cs_32	4	32/32	3573	56847	136	7	7	< 1	1	92	1642	< 1
Tag-RAM_cs_64	4	64/64	3569	106170	562	838	821	17	64	92	1642	< 1
Tag-RAM_cs_128	4	128/128	7090	201520	2522	1627	1594	33	128	92	1642	< 1
Tag-RAM_cs_256	4	256/256	14139	328886	11385	443	110	333	256	3563	8427	1
Tag-RAM_read_2	5	2/2/2/2	14299	330864	74	12	8	3	2	3703	9496	2
Tag-RAM_read_4	5	2/2/4/4	14301	330870	84	12	9	3	2	3703	9496	2
Tag-RAM_read_8	5	2/2/8/8	14303	330876	110	13	9	3	2	3703	9496	2
Tag-RAM_read_16	5	2/2/16/16	14305	330882	196	13	10	3	2	3703	9496	2
Tag-RAM_read_32	5	2/2/32/32	14307	330888	320	14	11	3	2	3703	9496	2
Tag-RAM_read_64	5	2/2/64/64	14309	330894	714	17	14	3	2	3703	9496	2
Tag-RAM_read_128	5	2/2/128/128	14311	330900	1626	24	20	3	2	3703	9496	2
Tag-RAM_read_256	5	2/2/256/256	14297	330842	4806	472	378	94	61	3703	9496	2
Tag-RAM_write_2	8	2/2/2/2/256	28375	1098440	1582	250	206	44	1	28331	329237	44
Tag-RAM_write_4	8	2/2/4/4/256	28377	1098446	1938	252	208	44	1	28331	329237	44
Tag-RAM_write_8	8	2/2/8/8/256	28379	1098452	2182	253	208	44	1	28331	329237	44
Tag-RAM_write_16	8	2/2/16/16/256	28381	1098458	1991	257	213	44	1	28331	329237	44
Tag-RAM_write_32	8	2/2/32/32/256	28383	1098464	2676	258	213	45	1	28331	329237	45
Tag-RAM_write_64	8	2/2/64/64/256	28385	1098470	20724	266	222	44	1	28331	329237	44
Tag-RAM_write_128	8	2/2/128/128/256	28387	1098476	MemEx	251	239	12	1	28286	329083	12
Tag-RAM_write_256	8	2/2/256/256/256	28373	1098418	MemEx	51116	39466	11650	250	28331	329237	46

industrial example can now be proven in a matter of seconds. Although for the second example the situation improved there is still no success in some cases. (Note that the experimental results reflect the current state-of-the-art after these improvements!)

Using the reduction techniques introduced in this thesis the RTL-BMC-approach was successful in more cases than before. It is to be noted that in almost all cases the time for proving the reduced problem has been neglectable. The most time was spent during reduction. The reduction time was mainly due to the normalization. The reason is that the rewrite system is very prototypical and leads to a blow up of the memory in some examples due to deficiencies in the implementation.

The time to find automorphisms was neglegtable in many examples where the number of nodes in the CUGs ranged between several thousand up to about $100,000$. In the memory example from this thesis up to 1.5 million nodes have been generated were the automorphism procedure took very long.

For the Tag-RAM the advantages of the reduction approach are tremendous even using the current prototypical implementation. Due to current limitations of the normalization procedure not all equivalences could be found for larger examples. However the verification succeeded in these cases as well, since the remaining cofactors were small enough.

The current implementation of the rewrite system as well as the reduction heuristics leave a lot of room for improvement. (Some ideas are sketched in the next chapter.)

Chapter 12

Conclusion and Perspective

12.1 Summary

The sustained trend towards integrating more and more functionality into systems on a chip requires continuous improvement of verification methodologies and underlying algorithms. Property checking, and in particular bounded model checking (BMC), have been proven useful for functional verification of block sized hardware designs. Still, the verification of some classes of hardware designs, enjoying regular substructures or complex arithmetic data paths, is difficult and often intractable. For regular designs, this is mainly due to individual treatment of symmetrical parts of the search space by backtrack search procedures used. One method of tackling these deficiencies is to exploit the structure of design and property on the Register Transfer Level (RTL).

This work describes a new approach for reduction of property checking problems on the RTL. It provides a framework for Bounded Interval Model Checking (BIMC) preserving the problem inherent structure. It allows functional verification of block sized hardware designs of regular nature for previously intractable instances. New preprocessing techniques for the underlying backtrack search algorithms, reduce their complexity. The preprocessing is based on eliminating symmetrical parts of the search space beforehand. This is done by removing symmetrical values from variables in the problem description.

In particular the following approach has been taken. In the first part, an overview of the ideas and techniques and their interrelation has been provided on the bitlevel. The second part extends them to the RTL.

Expanding the algebraic framework for Boolean functions to many-sorted algebras provides the basis for dealing with bitvector functions and their representations as bitvector terms. First basic bitvector algebras, along with a small but functionally complete set of identities generating them, have been provided. Their extension by defined operations allows matching the syntax of a particular hardware description language such as Verilog. Compact representations of bitvector terms, by fully collapsed term graphs, provide the framework for implementing subsequent reduction techniques.

Comparing bitvector functions is a prerequisite for BIMC on the RTL. This work provides suitable methods, using standard decision procedures, for comparing and reducing Boolean functions. Extensions of syntactic preprocessing approaches for Boolean functions by many-sorted term rewriting allow for semi-decision procedures treating subalgebras of the bitvector algebra syntactically. In particular constant-folding, constant-propagation

and normalization for bitvector terms, lead to practical syntactic preprocessing techniques operating on the RTL.

The extension of property checking for sequential circuits on the block-level by BIMC to the RTL allows the application of above techniques for problem reduction. This extension is based on the constructive representation of transition and output functions of Mealy machines by bitvector terms. Using algorithms for comparing Boolean functions makes the approach complete.

To enable verification designs of regular nature, a symmetry reduction method, based on symmetrical values, has been developed. The problem of detecting symmetrical values has been reduced to the permutation equivalence problem of Boolean functions. Advantages for verification, due to syntactic exploitation of self similarities in the function representation, have been outlined. The extension of this basic approach to the many-variable case as symmetrical value vectors, is the foundation for the extension to symmetry reduction of RTL-BMC-problems by exploitation of symmetrical values of bitvector variables.

In order to augment the approach as described, methods for solving the permutation equivalence problem for Bitvector functions have been developed. Several sufficient criteria are provided, namely syntactic symmetry, syntactic symmetry modulo identities, and variations for special classes of identities, such as identities describing commutativity or associativity and commutativity only. Decision and semi-decision procedures for these criteria based on bitvector term rewriting, isomorphism of suitably labeled DAGs and combinations thereof have been worked out. Extensions, treating sets of terms together, increase their efficiency.

Basic Symmetry reduction techniques for RTL-BMC, based on symmetrical values in bitvector functions, have been described. Their extension to symmetrical value vectors allows treatment of sets of bitvector variables together. Symmetry relations capture the correlation between the values of a single bitvector variable. Approximate symmetry relations are refined by combinations of semi-decision procedures for permutation equivalence of bitvector functions. After studying the limitations of the basic reduction scheme, the concept of symmetry relations has been enhanced by combining symmetry relations for different bitvector variables.

A practical reduction algorithm using this approach has been described. It was implemented within the experimental RTL-BMC-tool *RtProp*. Its application to real world property checking instances reveals significant performance advantages. Due to the reduction techniques described in this thesis, *RtProp* is able to succeed verifying previously intractable industrial designs, such as a Tag-RAM for a memory cache in a SoC design.

12.2 Future Work

Despite the successful application of the results of this thesis, it is only a small step. More research is required for maintaining formal verification technology applicable to ever larger designs. There is still much room for improvement in the results presented here. Some possible extensions and future uses are outlined below.

12.2.1 Verification

Among other work this thesis has led to the integration of word level representations for hardware designs within commercial formal verification tools currently developed at Infineon Technologies.

The representation of verification problems by bitvector terms allows the use of word-level solvers. However, current implementations often don not cover the whole bitvector algebra, either due to limitations of their expressiveness or due to systematic deficiencies. Often they are either well suited for data-path or control oriented designs. Combining them, for example by extending the approach described in [PK00] to the RTL, should be investigated. Other ideas include translation of bitvector equivalence problems onto the Boolean level by abstraction techniques [BGV99, PRSS99] and step-wise refinement on the occurence of unjustified counter-examples. This approach requires investigation of conflict analysis procedures on the RTL, which are needed to do sensible refinements of abstractions.

Rewriting bitvector terms could be used for combinational equivalence checking of RTL- vs. netlist-designs, e.g. by heuristically analyzing the likely original RTL-structure of the netlist by reverse engineering and rewriting the RTL-design into a similar structure. Alternatively a similar netlist could be generated and compared, e.g. by trying different low level architectures for arithmetic operations. This thesis provides a framework for such manipulations on the RTL. This kind of equivalence checking method would also benefit from methods for solving the permutation equivalence problem, as it is equivalent to the latch correspondence problem encountered in combinational equivalence checking.

Good heuristics for permutation equivalence also favor a direct comparison of RTL- vs. RTL-designs, e.g. after structural changes have been applied to the RTL-code. Additionally, normalization techniques provided here can be used to make them structurally more similar.

Finally, the framework provided here may serve as a starting point for comparison of behavioral models from system design with implementations on the RTL.

12.2.2 Symmetry Detection

The automorphism approach to detect symmetries can also be applied for finding symmetrical variables. This way symmetry breaking constraints can be deduced. This approach has been tried by the author and works well on artificial examples such as the pigeon-hole-problem or n-queens, but it does not work well for relevant designs from industrial property checking encountered so far. This is mainly due to a lack of such symmetries in these examples.

Extending the signature based approach [Moh99] to the RTL could lead to new semi-decision procedures for permutation equivalence of bitvector functions which could readily be integrated into our approach. Other work on symmetry detection, such as [PB97] for structural symmetry, or [AP01] for syntactic symmetry modulo commutativity, might be integrated as well. Conversely, our symmetry detection methods might be useful in different domains, such as word-level decision diagrams, extending the work of [SMMD99]. Combining symmetry detection on the RTL with bitlevel methods, could resolve deficiencies due to type incompatibility, currently solved only partly by input variable splitting.

The symmetry approach described here could also be extended to hierarchical designs, treating modules as entities whose internal symmetries are computed before hand, and captured by identities, thus abstracting them by a new bitvector operation.

12.2.3 Problem Reduction

Currently our reduction approach suffers from deficiencies in the prototypical implementation of the rewrite system. Compiling the rules into C++ and using them as a library would lead to a much improved performance while keeping the same flexibility for testing different rules. Extracting the now built-in rules from the current system and joining them with the configurable part would make the complete rewrite system accessible to automatic methods for establishing termination and confluence, as well as completion.

Combining our approach with other reduction techniques such as Johannsen:2001 is possible already today, however practical experience is still missing.

Currently the symmetry reduction approach is based on co-implication of cofactor terms. It can be relaxed to implication such that $f = 1$ iff $(f|_{x=0} = 1 \wedge (f|_{x=0} = 1 \Rightarrow f|_{x=1} = 1))$, which allows checking $f|_{x=0} = 1$ instead of $f = 1$. Using the permutation equivalence approach this yields $f = 1$ iff $(f|_{x=0} = 1 \wedge \exists \pi : (f|_{x=0} = 1 \Rightarrow \pi(f|_{x=1}) = 1))$. Given a set of identities E such an implication could be established syntactically by E-matching of cofactor-terms, i.e. $E \models t = 1$ iff $E \models t[x/0] = 1 \wedge \exists \sigma : E \models t[x/0]\sigma = t[x/1]$, where σ is a substitution of variables in $t[x/0]$ with subterms of $t[x/1]$. The same idea should be extendable to the RTL. Note that the base case with a degenerated substitution $\sigma \in Sym(Var(t))$, fixing x, is equivalent to the our technique, as σ is a variable renaming. It is easy to see that σ is a unifier of $t[x/0]$ and $t[x/1]$ above.

Finally, other approaches reducing models and properties individually, as done in in symbolic model checking, should be investigated for BMC on the RTL.

Part III

Appendix

Appendix A

Miscellaneous Proofs

A.1 Proof of Lemma 6.22 on the Functional Completeness of Bitvector Terms

We will show that any bitvector function can be expressed as concatenation of disjunctions (sums) of minterms over bit projections of bitvector variables. Obviously f can be expressed as concatenation of its components, i.e. for $f_i := f[i]_{[n_f]}$:

$$\begin{aligned} f(\mathbf{x}_1, \ldots, \mathbf{x}_k) &= (f_{n_f-1} \otimes \cdots \otimes f_0)(\mathbf{x}_1, \ldots, \mathbf{x}_k) \\ &= f_{n_f-1}(\mathbf{x}_1, \ldots, \mathbf{x}_k) \otimes \cdots \otimes f_0(\mathbf{x}_1, \ldots, \mathbf{x}_k) \\ &= (f_{n_f-1}(\mathbf{x}_1, \ldots, \mathbf{x}_k), \ldots, f_0(\mathbf{x}_1, \ldots, \mathbf{x}_k)) \end{aligned}$$

For $\mathbf{a} \in \mathbb{B}^{n_1} \times \cdots \times \mathbb{B}^{n_k}$ with $\mathbf{a} = (a_1, \ldots, a_k)$ let $f^1_{\mathbf{a}}$ be be the characteristic function for $\mathbf{a}$, i.e.

$$f^1_{\mathbf{a}}(\mathbf{x}_1, \ldots, \mathbf{x}_k) = \begin{cases} 0 & : (\mathbf{x}_1, \ldots, \mathbf{x}_k) \neq \mathbf{a} \\ 1 & : (\mathbf{x}_1, \ldots, \mathbf{x}_k) = \mathbf{a} \end{cases}$$

For variable indices $1 \leq i \leq m$ and bit indices $0 \leq k < n_i$ let $p_{i,k} : \mathbb{B}^{n_i} \to \mathbb{B}$ denote the k'th bit of the i'th variable or its negation, such that:

$$p_{i,k}(\mathbf{x}_i) = \begin{cases} \neg(\mathbf{x}_i[k]_{[n_i]}) & : a_i[k]_{[n_i]} = 0 \\ \mathbf{x}_i[k]_{[n_i]} & : a_i[k]_{[n_i]} = 1 \end{cases}$$

Then $f^1_{\mathbf{a}}$ can be written as minterm, i.e.:

$$\begin{aligned} f^1_{\mathbf{a}}(\mathbf{x}_1, \ldots, \mathbf{x}_k) &= p_{1,n_1-1}(\mathbf{x}_1) \wedge \cdots \wedge p_{1,0}(\mathbf{x}_1) \wedge \\ &\quad \vdots \\ &\quad p_{m,n_k-1}(\mathbf{x}_k) \wedge \cdots \wedge p_{m,0}(\mathbf{x}_k) \end{aligned}$$

Then the component functions f_i of f can be written as disjunctions (sums) of minterms, i.e.

$$f_i(\mathbf{x}_1, \ldots, \mathbf{x}_k) = \bigvee_{\mathbf{a} \in \mathbb{B}^{n_1} \times \cdots \wedge \mathbb{B}^{n_k}} f_i(\mathbf{a}) \wedge f^1_{\mathbf{a}}(X)$$

where conjuncts for which $f_i(\mathbf{a}) = 0$ holds are removed from the disjunction. The function f is then expressed as concatenation of the components f_i as above. ■

A.2 Proof of Lemma 6.24 on the Completeness of Bitvector Identities

W.l.o.g. let s and t have width n. Obviously $[s]_{\approx_{E_{\mathcal{BV}}}} = [t]_{\approx_{E_{\mathcal{BV}}}}$ iff $s \approx_{E_{\mathcal{BV}}} t$ iff ${}^n\mathcal{T}(X)/_{\approx_{E_{\mathcal{BV}}}} \models s = t$. We will simply write $s = t$ instead of $[s]_{\approx_{E_{\mathcal{BV}}}} = [t]_{\approx_{E_{\mathcal{BV}}}}$ in the following.

Since the algebra of bitvector functions is a model of $E_{\mathcal{BV}}$, from $s = t$ it follows that $\phi(s) = \phi(t)$, since ϕ is a homomorphism.

Conversely we have to show that $\phi(s) = \phi(t)$ implies $s = t$. This is equivalent to show that $s \neq t$ implies $\phi(s) \neq \phi(t)$, which is shown below.

1. Assume that the $s \neq t$ and $\phi(s) = \phi(t)$ is true for some s and t.

2. Using the bitvector identities any term can be written in CDNF. If u is a term and u' is u in CDNF, obviously $u = u'$ holds. Therefore it is safe to assume that s and t are in CDNF. Then s and t have the form $s_{n-1} \otimes \cdots \otimes s_0$ and $t_{n-1} \otimes \cdots \otimes t_0$ and it holds that $s_i = s[i]$ and $t_i = t[i]$. W.l.o.g. we can further assume that $t[i]$ and $s[i]$ are written as disjunction of minterms in normal form w.r.t. remark 6.21 (which can be achieved by further applying the Boolean identities).

3. Since $s \equiv (s_{n-1} \otimes \cdots \otimes s_0) \neq (t_{n-1} \otimes \cdots \otimes t_0) \equiv t$, there must be some i with $0 \leq i < n$ such that $s_i \neq t_i$.

4. Since ϕ is a homomorphism $(\phi(s)[n-1]) \otimes \cdots \otimes (\phi(s)[0]) = (\phi(t)[n-1]) \otimes \cdots \otimes (\phi(t)[0])$ and for all i we have $\phi(s[i]) = \phi(s)[i] = \phi(t)[i] = \phi(t[i])$.

5. From the first part of this lemma and $s_i = s[i]$ and $t[i] = t_i$ it follows that $\phi(s_i) = \phi(s[i]) = \phi(t[i]) = \phi(t_i)$.

6. Let ϕ' denote a function mapping bitvector functions to bitvector terms according to the last lemma.

7. Then from $\phi(s_i) = \phi(t_i)$ it follows that $\phi'\phi(s_i) \equiv \phi'\phi(t_i)$.

8. But since s_i and t_i are in minterm normal form, $\phi'\phi$ must be the identity function and we have $s_i \equiv \phi'\phi(s_i) \equiv \phi'\phi(t_i) \equiv t_i$. In particular we get $s_i = t_i$ in contradiction to the assumption, hence $\phi(s) \neq \phi(t)$.

A.3 Proof of Lemma 9.14 on Syntactic Symmetry by TSG-Isomorphism

Let ϕ be an isomorphism s.t. $\underline{\triangledown} G_s = \phi(\underline{\triangledown} G_t)$. First we construct a variable renaming π from ϕ. Then we show that π is a syntactic symmetry.

1. First some facts about ϕ are stated:

   (a) ϕ is a bijective mapping from the underlying DAG (N_t, E_t) to (N_s, E_s), i.e. it is an isomorphism between them.

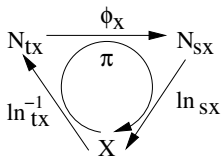


Figure A.1: Construction of Variable Renaming from Isomorphism

(b) For nodes of ∇G_t labeled with function symbols, the labeling does not change under ϕ, i.e. $\forall n_t \in N_t : ln_t(n_t) \in F \Rightarrow ln_t(n_t) = ln_s(\phi(n_t))$.

(c) For nodes labeled with variables the sort does not change under ϕ, i.e. $\forall n_t \in N_t : ln_t(n_t) \in X \Rightarrow sort(ln_t(n_t)) = sort(ln_s(\phi(n_t)))$.

(d) The edge labels do not change under ϕ, i.e. $\forall e_t \in E_t : le_t(e_t) = le_s(\phi(e_t))$.

2. Now we construct a variable permutation from ϕ.

   (a) Let $N_{sx} := \{n_s \in N_s : ln_s(n_s) \in X\}$, and $N_{tx} := \{n_t \in N_t : ln_t(n_t) \in X\}$ be the sets of nodes of ∇G_s and ∇G_t which are labeled with variables. Then the functions $ln_{sx} : N_{sx} \to X$ and $ln_{tx} : N_{tx} \to X$ with $ln_{sx} := ln_s|_X$ and $ln_{tx} := ln_t|_X$ are bijective, since all variable nodes are shared. Hence $\phi_X : N_{tx} \to N_{sx}$ with $\phi_X := \phi|_{N_{tx}}$ is also bijective. The situation is depicted in Fig. A.1.

   (b) Let $\pi : X \to X$ be a mapping with $\pi(x) := ln_{sx}(\phi_X(ln_{tx}^{-1}(x)))$. Then π is a variable permutation, since ln_{sx} and ln_{tx}^{-1} are bijections respecting sorts. (Then $\pi^{-1}(x) = ln_{tx}(\phi_X^{-1}(ln_{sx}^{-1}(x)))$.)

3. It remains to be shown that $s \equiv t\pi$. First a fully collapsed term graph $\nabla G_t''$ is constructed from ∇G_t such that $term(\nabla G_t'') \equiv t\pi$. Then we show that $\phi(\nabla G_t'') = \nabla G_s$.

   (a) Let $\nabla G_t'' = (N_t, E_t, ln_t'', le_t)$ with $ln_t'' : N_t \to X$ such that

   $$ln_t''(n) = \begin{cases} \pi(ln_t(n)) & : n \in N_{tx} \\ ln_t(n) & : n \in N_t \backslash N_{tx} \end{cases}$$

   Obviously $\nabla G_t''$ is fully collapsed, and $term(\nabla G_t'') \equiv t\pi$.

   (b) Now consider $\phi(\nabla G_t'') = (\phi N_t, \phi E_t, ln_t'', le_t)$. By construction of ϕ as isomorphism between $\underline{\nabla} G_t$ and $\underline{\nabla} G_s$, $\phi N_t = N_s$, $\phi E_t = E_s$, and for all $e_t \in E_t$, $le_t(e_t) = le_s(\phi(e_t))$. For $n_t \in N_t$ there are two cases to be considered to show that $ln_t''(n_t) = ln_s(\phi(n_t))$:

      i. For $n_t \in N_t \backslash N_{tx} : ln_t''(n) = ln_t(n) = ln_s(\phi(n_t))$ by construction of ϕ.

      ii. For $n_t \in N_{tx} : ln_t''(n) = \pi(ln_t(n_t)) = ln_{sx}(\phi_X(ln_{tx}^{-}1(ln_t(n_t)))) = ln_s(\phi(n))$.

      Hence ϕ is an isomorphism between $(\nabla G_t'')$ and ∇G_s.

Now we have $s \equiv term(\nabla G_s) \equiv term(\phi(\nabla G_t'')) \equiv term(\nabla G_{t\pi}) \equiv t\pi$, which proves one part of the conjecture.

Conversely, for s and t let $\pi \in Sym(X)$ be a variable permutation such that $s \equiv t\pi$. Then by definition, $\triangledown G_s$ and $\triangledown G_{t\pi}$ are isomorphic. W.l.o.g. let them be identical. We show that $\underline{\triangledown} G_s$ and $\underline{\triangledown} G_t$ are isomorphic by constructing an isomorphism from π.

1. Consider the terms t and s:
   (a) They have the same set of positions, i.e. $Pos(t) = Pos(s)$.
   (b) Variables at the same position have the same sort, in particular for a variable position $p_x \in Pos(t)$ with $t|_{p_x} = x$ (the subterm of t at position p_x) we have $s|_{p_x} = \pi(x)$.
   (c) The function symbols at each non variable position are identical, and for $p_u \in Pos(t)$ with $t|_{p_u} = u$ we have $s|_{p_u} = u\pi$.
2. Now consider the fully collapsed term graphs $\triangledown G_t = (N_t, E_t, ln_t, le_t)$, and (w.l.o.g.) $\triangledown G_{t\pi} = \triangledown G_s = (N_s, E_s, ln_s, le_s)$.
   (a) Let $\eta_t : Pos(t) \to N_t$ and $\eta_s : Pos(s) \to N_s$ be functions mapping each position p in t and s to a node in $\triangledown G_t$ and $\triangledown G_t$ representing $t|_p$ and $s|_p$, respectively.
   (b) η_t and η_s are injective, since each node in $\triangledown G_t$ and $\triangledown G_s$ represents a subterm of t and s, respectively.
   (c) Since $\triangledown G_t$ and $\triangledown G_s$ are fully collapsed, for each two positions $p_1, p_2 \in Pos(t) = Pos(s)$ we have $\eta_t(p_1) = \eta_t(p_2) \Leftrightarrow t|_{p_1} \equiv t|_{p_2}$ and $\eta_s(p_1) = \eta_s(p_2) \Leftrightarrow s|_{p_1} \equiv s|_{p_2}$. Therefore, η_t and η_s are unique, and for each position p there are unique nodes $n_t \in N_t$ and $n_s \in N_s$ such that $term(n_t) = t|_p$ and $term(n_s) = s|_p$ with $n_t = \eta_t(p)$ and $n_s = \eta_s(p)$, respectively. Hence $|N_t| = |N_s|$.
   (d) Since $\triangledown G_t$ and $\triangledown G_s$ are term graphs, the out degree of nodes representing the same non variable subterm position are the same, since they are labeled with the same function symbol, i.e. $\delta(\eta_t(p_u)) = \delta(\eta_s(p_u))$. The edge labels of edges starting in $\eta_t(p_u)$ and $\eta_s(p_u)$ are identical for the same reason, i.e. for all $1 \leq i \leq \delta(\eta_t(p_u))$, $le_t((\eta_t(p_u), \eta_t(p_u)[i])) = le_s((\eta_s(p_u), \eta_s(p_u)[i]))$.
3. Let $\phi : N_t \to N_s$ such that for all positions $p \in Pos(t)$, $\phi(\eta_t(p)) = \eta_s(p)$. ϕ is well defined, since η_t and η_s are unique injective functions. Since $|N_t| = |N_s|$, ϕ is one-to-one (and hence bijective).
4. Now consider the application of ϕ to $\underline{\triangledown} G_t = (N_t, E_t, ln'_t, le_t)$. We will show that $\phi(\underline{\triangledown} G_t) = \underline{\triangledown} G_s = (N_s, E_s, ln'_s, le_s)$, where le'_t and le'_s are defined as usual. Obviously $\phi N_t = N_s$, $\phi E_t = E_s$, and for all $e_t \in E_t$, $le_t(e_t) = le_s(\phi(e_t))$. We have to show that for all $n_t \in N_t$, $ln'_t(n_t) = ln'_s(\phi(n_t))$:
   (a) For each non-variable position p_u in t, s, we have $ln_t(\eta_t(p_u)) = ln_s(\eta_s(p_u)) = ln_s(\phi(\eta_t(p_u)))$. Hence for all nodes n_t in N_t with $ln'_t(n_t) \notin X$, $ln'_t(n_t) = ln'_s(\phi(n_t))$.
   (b) For each variable position p_x in t, s with $t|_{p_x} = x$ we have $s|_{p_x} = \pi(x)$ and $ln_t(\eta_t(p_x)) = \pi^{-1}(ln_s(\eta_s(p_x))) = \pi^{-1}(ln_s(\phi(\eta_t(p_x))))$. Hence for all nodes n_t in N_t with $ln'_t(n_t) \in X$, $ln'_t(n_t) = \pi^{-1}(ln'_s(\phi(n_t)))$. Since $sort(x) = sort(\pi(x))$ we have $ln'_t(n_t) = ln'_s(\phi(n_t))$.

Hence ϕ is an isomorphism between $\underline{\triangledown} G_t$ and $\underline{\triangledown} G_s$.

■

A.4 Proof of Lemma 9.21 on Syntactic Symmetry Modulo Permutative Identities by Isomorphism of PTSGs

We construct a variable renaming π such that $t\pi \equiv \pi(term(\triangledown G_t)) \equiv term(\triangledown G_{t\pi}) \approx_{E_C} term(\phi(\triangledown G_{t\pi})) \equiv term(\triangledown G_s) \equiv s$.

1. Since ϕ is an isomorphism from $\overline{\underline{\triangledown}} G_t$ to $\overline{\underline{\triangledown}} G_s$ we have $\phi N_t = N_s$, $\phi E_t = E_s$. Let π be a variable renaming constructed from ϕ along the lines of the proof for lemma 9.14 such that $\pi(term(\triangledown G_t)) \equiv term(\triangledown G_{t\pi})$. W.l.o.g. let $\triangledown G_{t\pi} = (N_t, E_t, ln_{t\pi}, le_t)$ with

$$ln_{t\pi}(n) = \begin{cases} ln_t(n) & : ln_t(n) \notin X \\ \pi(ln_t(n)) & : ln_t(n) \in X \end{cases}$$

2. For all nodes n in N_t we have $ln_{t\pi}(n) = ln_s(\phi(n))$ by construction of $ln_{t\pi}$. Now we show that $term(\triangledown G_{t\pi}) \approx_{E_C} term(\phi(\triangledown G_{t\pi}))$ by induction over the structure of (N_t, E_t). Let $n \in N_t$ be a node.

   IB: Let $rk(n) = 0$. Since there are no outgoing edges from leave nodes we have $term(\triangledown G_{t\pi}|_n) = ln_{t\pi}(n)$ and $term(\phi(\triangledown G_{t\pi})|_{\phi(n)}) = ln_s(\phi(n))$. Since $ln_{t\pi}(n) = ln_s(\phi(n))$, we have $term(\triangledown G_{t\pi}|_n) \approx_{E_C} term(\phi(\triangledown G_{t\pi})|_{\phi(n)})$.

   IH: For all nodes n with $rk(n) \leq k$ let $term(\triangledown G_{t\pi}|_n) \approx_{E_C} term(\phi(\triangledown G_{t\pi})|_{\phi(n)})$.

   IS: Let $rk(n) = k + 1$. Since $k + 1 > 0$, n is labeled with a function symbol $f \in F$. Let f be of arity $m = \delta(n) > 0$, and $ln_{t\pi}(n) = ln_t(n) = ln_s(\phi(n)) = f$. Then $term(\triangledown G_{t\pi}|_n) \equiv f(t_1, \ldots, t_m)$ with $t_i = term(\triangledown G_{t\pi}|_{n[i]})$, and $term(\phi(\triangledown G_{t\pi})|_{\phi(n)}) \equiv f(s_1, \ldots, s_m)$ with $s_i = term(\phi(\triangledown G_{t\pi})|_{\phi(n[i])})$ $(1 \leq i \leq \delta(n))$. There are two cases to be considered.

   $f \notin F_C$: For all edges $e = (n, n[i])$ originating in n we have $le_t(e) = le'_t(e) = le'_s(\phi(e)) = le_s(e)$. Then $\phi(n_t[i]) = \phi(n_t)[i]$. Therefore, we have $s_i = term(\phi(\triangledown G_{t\pi})|_{\phi(n[i])}) = term(\phi(\triangledown G_{t\pi})|_{\phi(n)[i]})$. Since $rk(n[i]) \leq k$, by induction hypothesis it follows that $s_i \approx_{E_C} t_i$, and hence $f(t_1, \ldots, t_n) \approx_{E_C} f(s_1, \ldots, s_n)$.

   $f \in F_C$: For all edges $e = (n, n[i])$ originating in n we have $\pi'(le_t(e)) = le_s(e)$ for some $\pi' \in Sym(\{1, \ldots, m\})$, since $\phi(e) = (\phi(n), \phi(n[i]))$. Let t_i and s_i be as above, then π' extends to $\{s_1, \ldots, s_m\}$ with $\pi'(s_i) = s_{\pi'(i)} = term(\phi(\triangledown G_{t\pi})|_{\phi(n[\pi'(i)])})$. By induction hypothesis it follows that $f(t_1, \ldots, t_n) \approx_{E_C} f(\pi'(s_1), \ldots, \pi'(s_m))$. Since f is a commutativity function symbol we have $f(t_1, \ldots, t_n) \approx_{E_C} f(s_1, \ldots, s_m)$.

   Since ϕ maps the root of $\overline{\underline{\triangledown}} G_t$ to the root of $\overline{\underline{\triangledown}} G_s$ we have $term(\triangledown G_{t\pi}) \approx_{E_C} term(\phi(\triangledown G_{t\pi}))$.

3. Finally we show that $term(\phi(\nabla G_{t\pi})) \equiv term(\nabla G_s)$ by proving (by induction) that $\phi(\nabla G_{t\pi})$ and ∇G_s are isomorphic. By construction of ϕ the sets of nodes and edges are the same i.e. $\phi N_t = N_s$, and $\phi E_t = E_s$. The node labels are identical, i.e. $ln_{t\pi}(\phi(n)) = ln_s(\phi(n))$ as shown above. Let $n \in N_s$ be a node.

   IB: Let $rk(n) = 0$ then n is a leaf and the subgraphs $\phi(\nabla G_{t\pi})|_n$ and $\nabla G_s|_n$ only consist of node n with identical label. Then they are trivially isomorphic and $term(\phi(\nabla G_{t\pi})|_n) \equiv term(\nabla G_s|_n)$.

   IH: For $rk(n) \leq k$ let $\phi(\nabla G_{t\pi})|_n$ and $\nabla G_s|_n$ be isomorphic.

   IS: Let $rk(n) = k+1$. Let $f = ln_s(n)$ and let $\delta(n) = m$. By construction $\phi(\nabla G_{t\pi})|_n$ and $\nabla G_s|_n$ can only differ in edge labels. Let E_n be the set of all edges starting in n. Let $ln_s(n) \notin F_C$. Then by construction of $\phi(\nabla G_{t\pi})$ for all edge e in E_n, $le_t\phi^{-1}(e) = le_s(e)$. Then by induction $term(\phi(\nabla G_{t\pi})|_n) \equiv term(\nabla G_s|_n)$.

   Let $ln_s(n) \in F_C$. First consider all pairs of target nodes $n[i]$, $n[j]$ $(1 \leq i, j \leq m)$ of these edges. If for $i \neq j$ we have $n[i] \neq n[j]$, then the subgraphs $\phi(\nabla G_{t\pi})|_{n[i]}$ and $\phi(\nabla G_{t\pi})|_{n[j]}$ cannot be isomorphic. The same holds for $\nabla G_s|_{n[i]}$ and $\nabla G_s|_{n[j]}$. If for $i \neq j$, we have $n[i] = n[j]$, then they are all isomorphic and are shared. (Both because ∇G_s and $\phi(\nabla G_{t\pi})$ are fully collapsed term graphs.) Therefore $le_t\phi^{-1}(e_i) \neq le_t\phi^{-1}(e_j)$. Hence, the labels of edges can only differ in the same group of multi edges ending in the same node. Therefore $\phi(\nabla G_{t\pi})|_n$ and $\nabla G_s|_n$ are isomorphic, and $term(\phi(\nabla G_{t\pi})|_n) \equiv term(\nabla G_s|_n)$.

Since the roots of $\phi(\nabla G_{t\pi})$ and ∇G_s are the same, they are isomorphic which proves the conjecture.

■

A.5 Proof for Lemma 9.40 on LDAG and CUG Automorphisms

The proof is constructive. Let within this proof φ denote the extension of ϕ to edges, i.e. $\varphi : E \to E$, and let φ' denote the extension of ϕ' to E', i.e. $\varphi' : E' \to E'$.

Let $\phi \in Aut(G)$. Let $\phi' : N' \to N'$ be as below.

$$\phi'(n') = \begin{cases} \phi(n') & : n' \in N \\ \varphi(n') & : n' \in E \end{cases}$$

To show that ϕ' is an automorphism of G' we have to show, that

1. ϕ' is a bijection,
2. ϕ' is a CUG-morphism, i.e. it is compatible with the graph structure $(\forall e' \in E' : \varphi'(e') \in E')$, and
3. ϕ' preserves the coloring of G'.

1. If $\phi \in Aut(G)$ then $\phi' : N' \to N'$ is well defined and it is a bijection, since $N \bigcap E = \emptyset$, and ϕ, φ are bijections.

2. We have to show that $\forall e' \in E' : \varphi'(e') \in E'$. Let $e' = (n, e)$ with $n \in N$ and $e \in E$. Then $\varphi'(e') = \varphi'(n, e) = (\phi'(n), \phi'(e)) = (\phi(n), \varphi(e))$. There are two cases to consider: 2a: $s(e) = n$, and 2b: $t(e) = n$.

   (a) If $e = (n, n')$, i.e. $n = s(e)$, then $\phi(n) = \phi(s(e)) = s(\varphi(e))$ (since $f \in Aut(G)$). Let $\varphi(e) = e_2 \in E$, hence $s(e_2) \in N$ and by definition $(s(e_2), e_2) \in E'$ and $(s(e_2), e_2) = (\phi(n), \varphi(e))$.

   (b) If $e = (n', n)$, i.e. $n = t(e)$, then $\phi(n) = \phi(t(e)) = t(\varphi(e))$ (since $f \in Aut(G)$). Let $\varphi(e) = e_2 \in E$, hence $t(e_2) \in N$ and $(\phi(n), \varphi(e)) = (t(e_2), e_2) \in E'$.

3. We have to show that $\forall e' = (n, e) \in E' : c(\phi'(n)) = c(n) \ \wedge \ c(\phi'(e)) = c(e)$.

   First consider $c(\phi'(n))$. By definition $c(n) = 0 \times ln(n) \times rk(n)$. Since $\phi'(n) \in N$ and $\phi'(n) = \phi(n)$, $c(\phi'(n)) = 0 \times ln(\phi(n)) \times rk(\phi(n))$. Since f is an automorphism of G, $ln(\phi(n)) = ln(n)$, and by lemma 6.29 $rk(\phi(n)) = rk(n)$, hence $c(\phi'(n)) = c(n)$.

   Now consider $c(\varphi'(e))$. Let $c(e) = 1 \times le(e) \times rk(t(e))$. Since $\varphi'(e) \in E$ and $\varphi'(e) = \varphi(e)$, $c(\varphi'(e)) = 1 \times le(\varphi(e)) \times rk(t(\varphi(e)))$. Since f is an automorphism of G, $le(\varphi(e)) = le(e)$. Again there are two cases:

   (a) If $e = (n, n')$, i.e. $t(e) = n'$, then $t(\varphi(e)) = \phi(n')$ and $rk(\phi(n')) = rk(n')$, by lemma 6.29

   (b) If $e = (n', n)$, i.e. $t(e) = n$, then $t(\varphi(e)) = \phi(n)$ and $rk(\phi(n)) = rk(n)$, as above.

   Hence $rk(t(\varphi(e))) = rk(t(e))$ and thus $c(\varphi'(e)) = c(e)$, and therefore φ' is color preserving.

From the results above it follows that $\phi \in Aut(G) \Rightarrow \phi' \in Aut(G')$.

Conversely, let $\phi' \in Aut(G')$. Let $\phi : N \to N$ with $\phi(n) := \phi'|_N(n)$ and $\varphi : E \to E$ with $\varphi(e) := \phi'|_E(e)$. To show that $\phi \in Aut(G)$ we have to show that 1. ϕ is bijective, 2. ϕ is compatible with the graph structure, i.e. $\phi \circ s = s \circ \varphi$ and $\phi \circ t = t \circ \varphi$, and 3. ϕ preserves labels of nodes and edges, i.e. $ln(\phi(n)) = ln(n)$ and $le(\varphi(e)) = le(e)$.

1. If $\phi' \in Aut(G')$ then $\phi'|_N : N \to N$ and $\phi'|_E : E \to E$, since ϕ' is color preserving and E and N never share colors. Hence ϕ, φ are bijections, and thus also f is bijective.

2. Let $e \in E$, with $s(e) = n$ and $t(e) = n'$.

$$
\begin{aligned}
e \in E \quad & \text{iff } (s(e), e) \in E' \text{ and } (t(e), e) \in E', \\
& \qquad\qquad\qquad\qquad\qquad\qquad \text{(by definition of } E') \\
& \text{iff } \varphi'((s(e), e)) \in E' \text{ and } \varphi'((t(e), e)) \in E', \\
& \qquad\qquad\qquad\qquad\qquad\qquad \text{(since } \phi' \in Aut(G')) \\
& \text{iff } (\phi'(s(e)), \phi'(e)) \in E' \text{ and } (\phi'(t(e)), \phi'(e)) \in E', \\
& \qquad\qquad\qquad\qquad\qquad\qquad \text{(by definition of } f') \\
& \text{iff } (\phi(n), \varphi(e)) \in E' \text{ and } (\phi(n'), \varphi(e)) \in E' \\
& \qquad\qquad\qquad\qquad\qquad\qquad \text{(by definition of } \phi,\ \varphi).
\end{aligned}
$$

   Since $E' := \{(s(e), e) : e \in E\} \cup \{(t(e), e) : e \in E\}$, follows $\varphi(e) \in E$ and either of the following must hold:

(a) $s(\varphi(e)) = \phi(n')$ and $t(\varphi(e)) = \phi(n)$, or

(b) $s(\varphi(e)) = \phi(n)$ and $t(\varphi(e)) = \phi(n')$.

Since $\phi' \in Aut(G')$ is color preserving, $c(n) = c(\phi(n))$ and $c(n') = c(\phi(n'))$, which implies $rk(n) = rk(\phi(n))$ and $rk(n') = rk(\phi(n'))$. Since $\forall e \in E : rk(s(e)) > rk(t(e))$, also $rk(s(\varphi(e))) > rk(t(\varphi(e)))$. Therefore $s(\varphi(e)) = \phi(n)$ and $t(\varphi(e)) = \phi(n')$. Hence $\phi(s(e)) = s(\varphi(e))$ and $\phi(t(e)) = t(\varphi(e))$.

3. Let $n \in N$, then $c(n) = 0 \times ln(n) \times rk(n)$. Since ϕ' is an automorphism of G', it is color preserving, i.e. $c(n) = c(\phi'(n))$. Since $\phi'|_N = \phi$, $\phi'(n) = \phi(n)$, thus $0 \times ln(n) \times rk(n) = c(n) = c(\phi'(n)) = 0 \times ln(\phi(n)) \times rk(\phi(n))$, and hence $ln(n) = ln(\phi(n))$.

   Let $e \in E$, then $c(e) = 1 \times le(e) \times rk(t(e))$. Since ϕ' is an automorphism of G', it is color preserving, i.e. $c(e) = c(\varphi'(e))$. Since $\varphi'|_E = \varphi$, $\varphi'(e) = \varphi(e)$, thus $1 \times le(e) \times rk(t(e)) = c(e) = c(\varphi'(e)) = 1 \times le(\varphi(e)) \times rk(t(\varphi(e)))$, and hence $le(e) = le(\varphi(e))$.

■

Appendix B

Extended Bitvector Algebra

B.1 Syntax

The syntax of the extended bitvector terms is derived from the signatures of the extended bitvector operations shown in Table B.1.

B.2 Semantics

The semantics of the extended bitvector operations is given by bitvector functions as below. The intended semantics of the more complex bitvector operations is visualized in Table B.2.

Let the Boolean operations $\neg$ (Negation), $\vee$ (Or), and $\wedge$ (And) over $\mathbb{B}$ be defined as usual. To convert bitvectors into naturals and vice versa, functions nat_n are used, where $nat((a_{n-1} \cdots a_0))_n$ denotes the integer value of the bitvector $(a_{n-1} \cdots a_0)$, with

$$\begin{aligned} nat_n : \mathbb{B}^n &\rightarrow \mathbb{N}_0 \\ (a_{n-1}, \ldots, a_0) &\mapsto \textstyle\sum_{i=0}^{n-1} a_i 2^i \end{aligned}$$

Converting a natural into a bitvector of width n is done by means of the functions vec_n, defined below:

$$\begin{aligned} vec_n : \mathbb{N}_0 &\rightarrow \mathbb{B}^n \\ a &\mapsto (a_{n-1}, \ldots, a_0) \ : \ nat_n((a_{n-1}, \ldots, a_0)) \equiv a \ mod \ 2^n \end{aligned}$$

To implicitly convert a natural a into a bitvector of width n, we use $(a_{n-1}, \ldots, a_0) := nat(a)_{[n]}$. Now conversion between bitvectors and naturals can be done very conveniently.

The interpretation of the bitvector function symbols is defined as follows.

1. Constants: For $c_{[n]} \in \mathbb{B}^n$: $[\![\mathbf{c}_{[n]}]\!]^{\mathcal{A}_{\mathcal{BV}}} = \mathbf{c}_{[n]}$ (Example: $[\![011001]\!]^{\mathcal{A}_{\mathcal{BV}}} = 011001$)

2. Concatenation:

$$\begin{aligned} [\![\otimes_{[n][m][n+m]}]\!]^{\mathcal{A}_{\mathcal{BV}}} : \mathbb{B}^n \times \mathbb{B}^m &\rightarrow \mathbb{B}^{n+m} \\ [\![\otimes_{[n][m][n+m]}]\!]^{\mathcal{A}_{\mathcal{BV}}}(\mathbf{a}_{[n]}, \mathbf{b}_{[m]}) &\mapsto (a_{n-1} \cdots a_0 b_{m-1} \cdots b_0) \end{aligned}$$

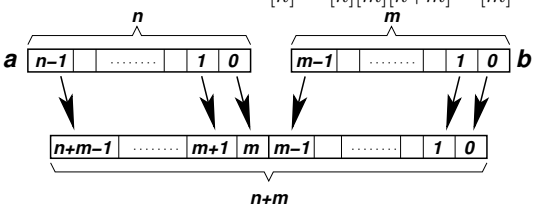
(Example: $[\![101 \otimes 100]\!]^{\mathcal{A}_{\mathcal{BV}}} = 101100$)

Table B.1: Extended Bitvector Signature Specification $\Sigma_{\mathcal{BV}} : F_{\mathcal{BV}} \to \mathbb{N}^+$

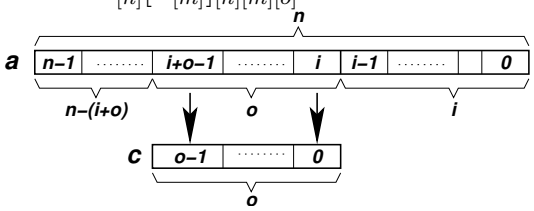
Name of operator	$F_{\mathcal{BV}}$	$\Sigma_{\mathcal{BV}}$ (with $n, m, o \in \mathbb{N}$)
Constants of width n	$\mathbb{B}^n$	$\mathbb{B}^n : \quad [n]$
Bitwise negation	$\neg\cdot$	$\neg_{[n][n]} : [n] \to [n]$
Unary minus	$-\cdot$	$-_{[n][n]} : [n] \to [n]$
Unary bitwise conjunction	$\wedge\cdot$	$\wedge_{[n][1]} : [n] \to [1]$
Unary bitwise disjunction	$\vee\cdot$	$\vee_{[n][1]} : [n] \to [1]$
Unary bitwise anti valence	$\oplus\cdot$	$\oplus_{[n][1]} : [n] \to [1]$
Zero extend	$\mathrm{ze}_{[m]}\cdot$	$\mathrm{ze}_{[n][m]} : [n] \to [m]$
Sign extend	$\mathrm{se}_{[m]}\cdot$	$\mathrm{se}_{[n][m]} : [n] \to [m]$
Implication	$\cdot \Rightarrow \cdot$	$\Rightarrow_{[1][1][1]}: [1] \times [1] \to [1]$
Binary bitwise conjunction	$\cdot \wedge \cdot$	$\wedge_{[n][n][n]} : [n] \times [n] \to [n]$
Binary bitwise disjunction	$\cdot \vee \cdot$	$\vee_{[n][n][n]} : [n] \times [n] \to [n]$
Binary bitwise anti valence	$\cdot \oplus \cdot$	$\oplus_{[n][n][n]} : [n] \times [n] \to [n]$
Binary addition	$\cdot + \cdot$	$+_{[n][n][n]} : [n] \times [n] \to [n]$
Binary multiplication	$\cdot * \cdot$	$*_{[n][n][n]} : [n] \times [n] \to [n]$
Division	$\cdot / \cdot$	$/_{[n][n][n]} : [n] \times [n] \to [n]$
Modulo	$\cdot \% \cdot$	$\%_{[n][n][n]} : [n] \times [n] \to [n]$
Rotate right	$\cdot \circlearrowright \cdot$	$\circlearrowright_{[n][m][n]}: [n] \times [m] \to [n]$
Rotate left	$\cdot \circlearrowleft \cdot$	$\circlearrowleft_{[n][m][n]}: [n] \times [m] \to [n]$
Shift right	$\cdot \gg \cdot$	$\gg_{[n][m][n]}: [n] \times [m] \to [n]$
Shift left	$\cdot \ll \cdot$	$\ll_{[n][m][n]}: [n] \times [m] \to [n]$
Equality	$\cdot = \cdot$	$=_{[n][n][1]}: [n] \times [n] \to [1]$
Less	$\cdot < \cdot$	$<_{[n][n][1]}: [n] \times [n] \to [1]$
Concatenation	$\cdot \otimes \cdot$	$\otimes_{[n][m][n+m]} : [n] \times [m] \to [n+m]$
Slice	$\cdot[\cdot]_{[o]}$	$[]_{[n][m][o]} : [n] \times [m] \to [o]$
Multiplex read	$\mathrm{read}(\cdot,\cdot)_{[o]}$	$\mathrm{read}_{[n][m][o]} : [n] \times [m] \to [o]$
If-then-else	$\mathrm{ite}(\cdot,\cdot,\cdot)$	$\mathrm{ite}_{[1][n][n][n]} : \mathbb{B} \times [n] \times [n] \to [n]$
Multiplex write	$\mathrm{write}(\cdot,\cdot,\cdot)$	$\mathrm{write}_{[n][m][o][n]} : [n] \times [m] \times [o] \to [n]$
m-ary bitwise conjunction	$\wedge(\cdot,\cdot,\ldots,\cdot)$	$\wedge_{m*[n][n]} : [n]^m \to [n]$
m-ary bitwise disjunction	$\vee(\cdot,\cdot,\ldots,\cdot)$	$\vee_{m*[n][n]} : [n]^m \to [n]$
m-ary bitwise antivalence	$\oplus(\cdot,\cdot,\ldots,\cdot)$	$\oplus_{m*[n][n]} : [n]^m \to [n]$
m-ary addition / Sum	$\sum(\cdot,\cdot,\ldots,\cdot)$	$\sum_{m*[n][n]} : [n]^m \to [n]$
m-ary multiplication	$\prod(\cdot,\cdot,\ldots,\cdot)$	$\prod_{m*[n][n]} : [n]^m \to [n]$

Table B.2: Graphical Representation of Bitvector Operations

Concatenation: $\mathbf{a}_{[n]} \otimes_{[n][m][n+m]} \mathbf{b}_{[m]}$

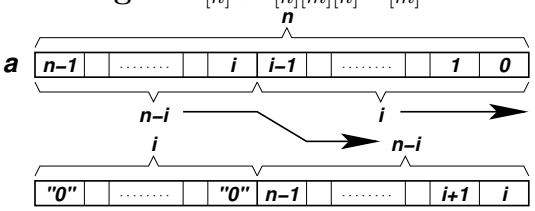


Slice: $\mathbf{a}_{[n]}[\mathbf{b}_{[m]}]_{[n][m][o]}$



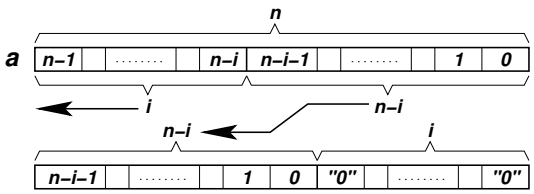
$(i = nat_m(\mathbf{b}_{[m]}))$

Shift right: $\mathbf{a}_{[n]} \gg_{[n][m][n]} \mathbf{b}_{[m]}$



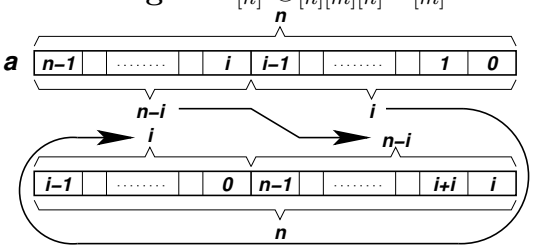
$(i = nat_{[m]}(\mathbf{b}_{[m]}))$

Shift left: $\mathbf{a}_{[n]} \ll_{[n][m][n]} \mathbf{b}_{[m]}$



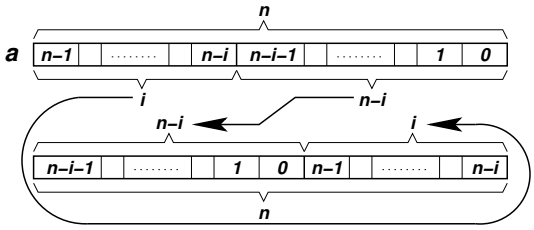
$(i = nat_{[m]}(\mathbf{b}_{[m]}))$

Rotate right: $\mathbf{a}_{[n]} \circlearrowright_{[n][m][n]} \mathbf{b}_{[m]}$



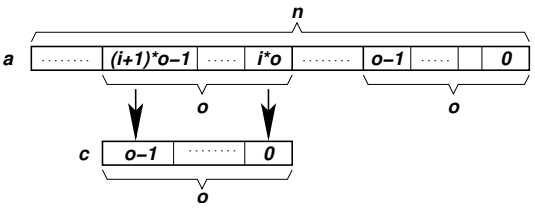
$(i = nat_{[m]}(\mathbf{b}_{[m]})\%n)$

Rotate left: $\mathbf{a}_{[n]} \circlearrowleft_{[n][m][n]} \mathbf{b}_{[m]}$



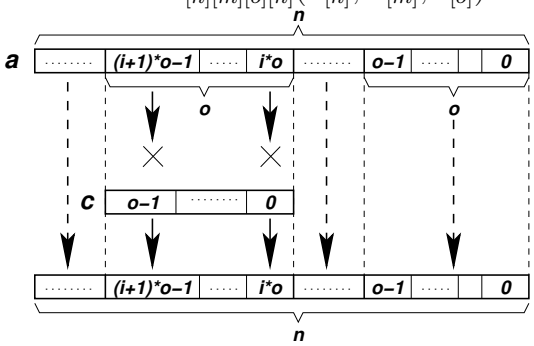
$(i = nat_{[m]}(\mathbf{b}_{[m]})\%n)$

Read: $\text{read}_{[n][m][o]}(\mathbf{a}_{[n]}, \mathbf{b}_{[m]})$



$(i = nat_m(\mathbf{b}_{[m]}), j \in \mathbb{N}_o)$

Write: $\text{write}_{[n][m][o][n]}(\mathbf{a}_{[n]}, \mathbf{b}_{[m]}, \mathbf{c}_{[o]})$



$(i = nat_m(\mathbf{b}_{[m]}))$

3. Slice:

$$\begin{array}{ll} [\![[\cdot]_{[n][m][o]}]\!]^{\mathcal{A}_{\mathcal{BV}}} : \mathbb{B}^n \times \mathbb{B}^m & \to \mathbb{B}^o \\ [\![[\cdot]_{[n][m][o]}]\!]^{\mathcal{A}_{\mathcal{BV}}}(\mathbf{a}_{[n]}, \mathbf{b}_{[m]}) & \mapsto (c_{o-1} \cdots c_0) \\ & \quad \text{with } c_j = \begin{cases} a_{i+j} & : i+j < n \\ 0 & : i+j \geq n \end{cases} \\ & \quad (i = nat_m(\mathbf{b}_{[m]}), j \in \mathbb{N}_o) \end{array}$$

(Example: $[\![101110[011]_{[2]}]\!] = 01$)

4. Bitwise negation:

$$\begin{array}{ll} [\![\neg_{[n][n]}]\!]^{\mathcal{A}_{\mathcal{BV}}} : \mathbb{B}^n & \to \mathbb{B}^n \\ [\![\neg_{[n][n]}]\!]^{\mathcal{A}_{\mathcal{BV}}}(\mathbf{a}_{[n]}) & \mapsto \neg a_{n-1} \cdots \neg a_0 \end{array}$$

(Example: $[\![\neg 011001]\!] = 100110$)

5. Unary minus:

$$\begin{array}{ll} [\![-_{[n][n]}]\!]^{\mathcal{A}_{\mathcal{BV}}} : \mathbb{B}^n & \to \mathbb{B}^n \\ [\![-_{[n][n]}]\!]^{\mathcal{A}_{\mathcal{BV}}}(\mathbf{a}_{[n]}) & \mapsto (\neg_{[n][n]}\mathbf{a}_{[n]}) +_{[n][n][n]} \mathbf{1}_{[n]} \end{array}$$

(Example: $[\![-101100]\!] = 010100$)

6. Unary bitwise conjunction:

$$\begin{array}{ll} [\![\wedge_{[n][1]}]\!]^{\mathcal{A}_{\mathcal{BV}}} : \mathbb{B}^n & \to \mathbb{B} \\ [\![\wedge_{[n][1]}]\!]^{\mathcal{A}_{\mathcal{BV}}}(\mathbf{a}_{[n]}) & \mapsto a_{n-1} \wedge \cdots \wedge a_1 \wedge a_0 \end{array}$$

(Example: $[\![\wedge 01111]\!] = 0$)

7. Unary bitwise disjunction:

$$\begin{array}{ll} [\![\vee_{[n][1]}]\!]^{\mathcal{A}_{\mathcal{BV}}} : \mathbb{B}^n & \to \mathbb{B} \\ [\![\vee_{[n][1]}]\!]^{\mathcal{A}_{\mathcal{BV}}}(\mathbf{a}_{[n]}) & \mapsto a_{n-1} \vee \cdots \vee a_1 \vee a_0 \end{array}$$

(Example: $[\![\vee 001010]\!] = 1$)

8. Unary bitwise antivalence:

$$\begin{array}{ll} [\![\oplus_{[n][1]}]\!]^{\mathcal{A}_{\mathcal{BV}}} : \mathbb{B}^n & \to \mathbb{B} \\ [\![\oplus_{[n][1]}]\!]^{\mathcal{A}_{\mathcal{BV}}}(\mathbf{a}_{[n]}) & \mapsto a_{n-1} \oplus \cdots \oplus a_1 \oplus a_0 \end{array}$$

(Example: $[\![\oplus 010101]\!] = 1$)

9. Zero extend:

$$\begin{array}{ll} [\![\mathrm{ze}_{[n][m]}]\!]^{\mathcal{A}_{\mathcal{BV}}} : \mathbb{B}^n & \to \mathbb{B}^m \\ [\![\mathrm{ze}_{[n][m]}]\!]^{\mathcal{A}_{\mathcal{BV}}}(\mathbf{a}_{[n]}) & \mapsto \begin{cases} \mathbf{0}_{[m-n]} \otimes \mathbf{a}_{[n]} & : m > n \\ \mathbf{a}_{[n]}[\mathbf{0}_{[m]}]_{[n][m][m]} & : m \leq n \end{cases} \end{array}$$

(Example: $[\![\mathrm{ze}_{[6]}1001]\!] = 001001$)

10. Sign Extend:

$$\begin{aligned} [\![\mathrm{se}_{[n][m]}]\!]^{\mathcal{A}_{\mathcal{BV}}} : \mathbb{B}^n &\to \mathbb{B}^m \\ [\![\mathrm{se}_{[n][m]}]\!]^{\mathcal{A}_{\mathcal{BV}}}(\mathbf{a}_{[n]}) &\mapsto \begin{cases} \mathbf{b}_{[m-n]} \otimes \mathbf{a}_{[n]} & : m > n, (\forall i \in \mathbb{B}_{m-n} : b_{i-1} = a_{n-1}) \\ \mathbf{a}_{[n]}[\mathbf{0}_{[m]}]_{[n][m][m]} & : m \leq n \end{cases} \end{aligned}$$

(Example:$[\![\mathrm{se}_{[6]}1001]\!] = 111001$)

11. Binary bitwise conjunction:

$$\begin{aligned} [\![\wedge_{[n][n][n]}]\!]^{\mathcal{A}_{\mathcal{BV}}} : \mathbb{B}^n \times \mathbb{B}^n &\to \mathbb{B}^n \\ [\![\wedge_{[n][n][n]}]\!]^{\mathcal{A}_{\mathcal{BV}}}(\mathbf{a}_{[n]}, \mathbf{b}_{[n]}) &\mapsto (a_{n-1} \wedge b_{n-1})(a_{n-2} \wedge b_{n-2}) \cdots (a_0 \wedge b_0) \end{aligned}$$

(Example: $[\![01011 \wedge 11101]\!] = 01001$)

12. Binary bitwise disjunction:

$$\begin{aligned} [\![\vee_{[n][n][n]}]\!]^{\mathcal{A}_{\mathcal{BV}}} : \mathbb{B}^n \times \mathbb{B}^n &\to \mathbb{B}^n \\ [\![\vee_{[n][n][n]}]\!]^{\mathcal{A}_{\mathcal{BV}}}(\mathbf{a}_{[n]}, \mathbf{b}_{[n]}) &\mapsto (a_{n-1} \vee b_{n-1})(a_{n-2} \vee b_{n-2}) \cdots (a_0 \vee b_0) \end{aligned}$$

(Example: $[\![01001 \vee 11101]\!] = 11101$)

13. Binary bitwise antivalence:

$$\begin{aligned} [\![\oplus_{[n][n][n]}]\!]^{\mathcal{A}_{\mathcal{BV}}} : \mathbb{B}^n \times \mathbb{B}^n &\to \mathbb{B}^n \\ [\![\oplus_{[n][n][n]}]\!]^{\mathcal{A}_{\mathcal{BV}}}(\mathbf{a}_{[n]}, \mathbf{b}_{[n]}) &\mapsto (a_{n-1} \oplus b_{n-1})(a_{n-2} \oplus b_{n-2}) \cdots (a_0 \oplus b_0) \end{aligned}$$

(Example: $[\![01001 \oplus 11101]\!] = 10100$)

14. Implication:

$$\begin{aligned} [\![\Rightarrow_{[1][1][1]}]\!]^{\mathcal{A}_{\mathcal{BV}}} : \mathbb{B} \times \mathbb{B} &\to \mathbb{B} \\ [\![\Rightarrow_{[1][1][1]}]\!]^{\mathcal{A}_{\mathcal{BV}}}(\mathbf{a}_{[1]}, \mathbf{b}_{[1]}) &\mapsto \neg a_{[1]} \vee b_{[1]} \end{aligned}$$

(Example: $[\![1 \Rightarrow 0]\!] = 0$)

15. Binary addition:

$$\begin{aligned} [\![+_{[n][n][n]}]\!]^{\mathcal{A}_{\mathcal{BV}}} : \mathbb{B}^n \times \mathbb{B}^n &\to \mathbb{B}^n \\ [\![+_{[n][n][n]}]\!]^{\mathcal{A}_{\mathcal{BV}}}(\mathbf{a}_{[n]}, \mathbf{b}_{[n]}) &\mapsto vec_n(nat_n(\mathbf{a}_{[n]}) + nat_n(\mathbf{b}_{[n]})) \end{aligned}$$

(Example: $[\![11010 + 10101]\!] = 01111$)

16. Binary multiplication:

$$\begin{aligned} [\![*_{[n][n][n]}]\!]^{\mathcal{A}_{\mathcal{BV}}} : \mathbb{B}^n \times \mathbb{B}^n &\to \mathbb{B}^n \\ [\![*_{[n][n][n]}]\!]^{\mathcal{A}_{\mathcal{BV}}}(\mathbf{a}_{[n]}, \mathbf{b}_{[n]}) &\mapsto vec_n(nat_n(\mathbf{a}_{[n]}) * nat_n(\mathbf{b}_{[n]})) \end{aligned}$$

(Example: $[\![00101 * 00011]\!] = 01111$)

17. Division:

$$\begin{aligned}
[\![/_{[n][n][n]}]\!]^{\mathcal{ABV}} : \mathbb{B}^n \times \mathbb{B}^n &\to \mathbb{B}^n \\
[\![/_{[n][n][n]}]\!]^{\mathcal{ABV}}(\mathbf{a}_{[n]}, \mathbf{b}_{[n]}) &\mapsto \begin{cases} vec_n(nat_n(\mathbf{a}_{[n]})/nat_n(\mathbf{b}_{[n]})) & : nat_n(\mathbf{b}_{[n]}) > 0 \\ \underbrace{1 \cdots 1}_{n} & : nat_n(\mathbf{b}_{[n]}) = 0 \end{cases}
\end{aligned}$$

(Example: $[\![10000/00011]\!] = 00101$)

18. Modulo:

$$\begin{aligned}
[\![\%_{[n][n][n]}]\!]^{\mathcal{ABV}} : \mathbb{B}^n \times \mathbb{B}^n &\to \mathbb{B}^n \\
[\![\%_{[n][n][n]}]\!]^{\mathcal{ABV}}(\mathbf{a}_{[n]}, \mathbf{b}_{[n]}) &\mapsto \begin{cases} vec_n(nat_n(\mathbf{a}_{[n]})\%nat_n(\mathbf{b}_{[n]})) & : nat_n(\mathbf{b}_{[n]}) > 0 \\ \mathbf{0}_{[n]} & : nat_n(\mathbf{b}_{[n]}) = 0 \end{cases}
\end{aligned}$$

(Example: $[\![10000\%00011]\!] = 00001$)

19. Shift right:

$$\begin{aligned}
[\![\gg_{[n][m][n]}]\!]^{\mathcal{ABV}} : \mathbb{B}^n \times \mathbb{B}^m &\to \mathbb{B}^n \\
[\![\gg_{[n][m][n]}]\!]^{\mathcal{ABV}}(\mathbf{a}_{[n]}, \mathbf{b}_{[m]}) &\mapsto \begin{cases} \mathbf{a}_{[n]} & : i = 0 \\ \mathbf{0}_{[i]} \otimes \mathbf{a}_{[n]}[\mathbf{b}_{[m]}]_{[n-i]} & : 0 < i < n \\ \mathbf{0}_{[n]} & : i \geq n \end{cases} \\
& \qquad (i = nat_{[m]}(\mathbf{b}_{[m]}))
\end{aligned}$$

(Example: $[\![000111 \gg 0010]\!] = 000001$)

20. Shift left:

$$\begin{aligned}
[\![\ll_{[n][m][n]}]\!]^{\mathcal{ABV}} : \mathbb{B}^n \times \mathbb{B}^m &\to \mathbb{B}^n \\
[\![\ll_{[n][m][n]}]\!]^{\mathcal{ABV}}(\mathbf{a}_{[n]}, \mathbf{b}_{[m]}) &\mapsto \begin{cases} \mathbf{a}_{[n]} & : i = 0 \\ \mathbf{a}_{[n]}[\mathbf{0}_{[1]}]_{[n-i]} \otimes \mathbf{0}_{[i]} & : 0 < i < n \\ \mathbf{0}_{[n]} & : i \geq n \end{cases} \\
& \qquad (i = nat_{[m]}(\mathbf{b}_{[m]}))
\end{aligned}$$

(Example: $[\![000111 \ll 0010]\!] = 011100$)

21. Rotate right:

$$\begin{aligned}
[\![\circlearrowright_{[n][m][n]}]\!]^{\mathcal{ABV}} : \mathbb{B}^n \times \mathbb{B}^m &\to \mathbb{B}^n \\
[\![\circlearrowright_{[n][m][n]}]\!]^{\mathcal{ABV}}(\mathbf{a}_{[n]}, \mathbf{b}_{[m]}) &\mapsto \begin{cases} \mathbf{a}_{[n]} & : i = 0 \\ \mathbf{a}_{[n]}[\mathbf{0}_{[0]}]_{[i]} \otimes \mathbf{a}_{[n]}[vec_n(i)]_{[n-i]} & : 0 < i < n \end{cases} \\
& \qquad (i = nat_{[m]}(\mathbf{b}_{[m]})\%n)
\end{aligned}$$

(Example: $[\![000111 \circlearrowright 0010]\!] = 110001$)

22. Rotate left:

$$\begin{aligned}
[\![\circlearrowleft_{[n][m][n]}]\!]^{\mathcal{ABV}} : \mathbb{B}^n \times \mathbb{B}^m &\to \mathbb{B}^n \\
[\![\circlearrowleft_{[n][m][n]}]\!]^{\mathcal{ABV}}(\mathbf{a}_{[n]}, \mathbf{b}_{[m]}) &\mapsto \begin{cases} \mathbf{a}_{[n]} & : i = 0 \\ \mathbf{a}_{[n]}[\mathbf{0}_{[n]}]_{[n-i]} \otimes \mathbf{a}_{[n]}[vec_n(n-i)]_{[i]} & : 0 < i < n \end{cases} \\
& \qquad (i = nat_{[m]}(\mathbf{b}_{[m]})\%n)
\end{aligned}$$

(Example: $[\![000111 \circlearrowleft 0010]\!] = 011100$)

23. Equal:

$$\begin{aligned} [\![=_{[n][n][1]}]\!]^{\mathcal{AB}\mathcal{V}} : \mathbb{B}^n \times \mathbb{B}^n &\to \mathbb{B} \\ [\![=_{[n][n][1]}]\!]^{\mathcal{AB}\mathcal{V}}(\mathbf{a}_{[n]}, \mathbf{b}_{[n]}) &\mapsto \begin{cases} 1 & : \forall i \in \mathbb{N}_n : a_i = b_i \\ 0 & : \exists i \in \mathbb{N}_n : a_i \neq b_i \end{cases} \end{aligned}$$

(Example: $[\![000111 = 001111]\!] = 0$)

24. Less:

$$\begin{aligned} [\![<_{[n][n][1]}]\!]^{\mathcal{AB}\mathcal{V}} : \mathbb{B}^n \times \mathbb{B}^n &\to \mathbb{B} \\ [\![<_{[n][n][1]}]\!]^{\mathcal{AB}\mathcal{V}}(\mathbf{a}_{[n]}, \mathbf{b}_{[n]}) &\mapsto \begin{cases} 1 & : nat_n(\mathbf{a}_{[n]}) < nat_n(\mathbf{b}_{[n]}) \\ 0 & : nat_n(\mathbf{a}_{[n]}) \geq nat_n(\mathbf{b}_{[n]}) \end{cases} \end{aligned}$$

(Example: $[\![000111 < 001000]\!] = 1$)

25. Multiplex read:

$$\begin{aligned} [\![\mathrm{read}_{[n][m][o]}]\!]^{\mathcal{AB}\mathcal{V}} : \mathbb{B}^n \times \mathbb{B}^m &\to \mathbb{B}^o \\ [\![\mathrm{read}_{[n][m][o]}]\!]^{\mathcal{AB}\mathcal{V}}(\mathbf{a}_{[n]}, \mathbf{b}_{[m]}) &\mapsto c_{o-1} \cdots c_0 \\ &\text{with } c_j = \begin{cases} a_{i*o+j-1} & : i + j \leq n \\ 0 & : i + j > n \end{cases} \\ &\qquad (i = nat_m(\mathbf{b}_{[m]}), j \in \mathbb{N}_o) \end{aligned}$$

(Example: $[\![\mathrm{read}_{[3]}(000.111.011.001,\ 011)]\!] = 111$)

26. If-then-else[1]:

$$\begin{aligned} [\![\mathrm{ite}_{[1][n][n][n]}]\!]^{\mathcal{AB}\mathcal{V}} : \mathbb{B} \times \mathbb{B}^n \times \mathbb{B}^n &\to \mathbb{B}^n \\ [\![\mathrm{ite}_{[1][n][n][n]}]\!]^{\mathcal{AB}\mathcal{V}}(\mathbf{c}_{[1]}, \mathbf{a}_{[n]}, \mathbf{b}_{[n]}) &\mapsto \begin{cases} \mathbf{a}_{[n]} & : \mathbf{c}_{[1]} =_{[1]} \mathbf{1}_{[1]} \\ \mathbf{b}_{[n]} & : \mathbf{c}_{[1]} =_{[1]} \mathbf{0}_{[1]} \end{cases} \end{aligned}$$

(Example: $[\![\mathrm{ite}(0, 101, 010)]\!] = 010$)

27. Multiplex-write:

$$\begin{aligned} [\![\mathrm{write}_{[n][m][o][n]}]\!]^{\mathcal{AB}\mathcal{V}} : \mathbb{B}^n \times \mathbb{B}^m \times \mathbb{B}^o &\to \mathbb{B}^n \\ [\![\mathrm{write}_{[n][m][o][n]}]\!]^{\mathcal{AB}\mathcal{V}}&(\mathbf{a}_{[n]}, \mathbf{b}_{[m]}, \mathbf{c}_{[o]}) \\ \mapsto \begin{cases} \mathbf{a}_{[n]} & : n \leq o * i \\ \mathbf{c}_{[o]}[\mathbf{0}_{[o]}]_{[n-o*i]} \otimes \mathbf{a}_{[n]}[\mathbf{0}_{[n]}]_{[o*i]} & : n > o * i > n - o \\ \mathbf{a}_{[n]}[vec_n(o * (i+1))]_{[n-o*(i+1)]} \otimes \mathbf{c}_{[o]} \otimes \mathbf{a}_{[n]}[\mathbf{0}_{[n]}]_{[o*i]} & : n - o \geq o * i \geq o \\ \mathbf{a}_{[n]}[vec_n(o)]_{[n-o]} \otimes \mathbf{c}_{[o]} & : i = 0 \end{cases} \\ (i = nat_m(\mathbf{b}_{[m]})) \end{aligned}$$

(Example: $[\![\mathrm{write}(111.111.000.101,\ 11,\ 10)]\!] = 111.010.000.101$)

[1]Sometimes 'ite$(\cdot,\cdot,\cdot)$' will be denoted as '$\cdot ? \cdot : \cdot$'.

28. m-ary bitwise conjunction:

$$\begin{array}{rl} [\![\wedge_{m*[n][n]}]\!]^{\mathcal{A}_{\mathcal{BV}}} : \overbrace{\mathbb{B}^n \times \cdots \times \mathbb{B}^n}^{m} & \to \mathbb{B}^n \\ [\![\wedge_{m*[n][n]}]\!]^{\mathcal{A}_{\mathcal{BV}}}(a_{[n],1}, \ldots, a_{[n],m}) & \mapsto a_{[n],1} \wedge_{[n][n][n]} (a_{[n],2} \wedge_{[n][n][n]} (\cdots \\ & \quad (a_{[n],m-1} \wedge_{[n][n][n]} a_{[n],m}) \cdots)) \end{array}$$

(Example: $[\![\wedge_{3*[4][4]}(1100, 1100, 0110)]\!] = (0100)$)

29. m-ary bitwise disjunction:

$$\begin{array}{rl} [\![\vee_{m*[n][n]}]\!]^{\mathcal{A}_{\mathcal{BV}}} : \overbrace{\mathbb{B}^n \times \cdots \times \mathbb{B}^n}^{m} & \to \mathbb{B}^n \\ [\![\vee_{m*[n][n]}]\!]^{\mathcal{A}_{\mathcal{BV}}}(a_{[n],1}, \ldots, a_{[n],m}) & \mapsto a_{[n],1} \vee_{[n][n][n]} (a_{[n],2} \vee_{[n][n][n]} (\cdots \\ & \quad (a_{[n],m-1} \vee_{[n][n][n]} a_{[n],m}) \cdots)) \end{array}$$

(Example: $[\![\vee_{3*[4][4]}(1100, 1100, 0110)]\!] = (1110)$)

30. m-ary bitwise antivalence:

$$\begin{array}{rl} [\![\oplus_{m*[n][n]}]\!]^{\mathcal{A}_{\mathcal{BV}}} : \overbrace{\mathbb{B}^n \times \cdots \times \mathbb{B}^n}^{m} & \to \mathbb{B}^n \\ [\![\oplus_{m*[n][n]}]\!]^{\mathcal{A}_{\mathcal{BV}}}(a_{[n],1}, \ldots, a_{[n],m}) & \mapsto a_{[n],1} \oplus_{[n][n][n]} (a_{[n],2} \oplus_{[n][n][n]} (\cdots \\ & \quad (a_{[n],m-1} \oplus_{[n][n][n]} a_{[n],m}) \cdots)) \end{array}$$

(Example: $[\![\oplus_{3*[4][4]}(1100, 1100, 0110)]\!] = (0110)$)

31. m-ary addition / sum:

$$\begin{array}{rl} [\![\sum_{m*[n][n]}]\!]^{\mathcal{A}_{\mathcal{BV}}} : \overbrace{\mathbb{B}^n \times \cdots \times \mathbb{B}^n}^{m} & \to \mathbb{B}^n \\ [\![\sum_{m*[n][n]}]\!]^{\mathcal{A}_{\mathcal{BV}}}(a_{[n],1}, \ldots, a_{[n],m}) & \mapsto vec_n(\sum_{i=1}^{m} nat_n(a_{[n],i})) \end{array}$$

(Example: $[\![\sum_{3*[4][4]}(1100, 1100, 0110)]\!] = (1110)$)

32. m-ary multiplication / product:

$$\begin{array}{rl} [\![\prod_{m*[n][n]}]\!]^{\mathcal{A}_{\mathcal{BV}}} : \overbrace{\mathbb{B}^n \times \cdots \times \mathbb{B}^n}^{m} & \to \mathbb{B}^n \\ [\![\prod_{m*[n][n]}]\!]^{\mathcal{A}_{\mathcal{BV}}}(a_{[n],1}, \ldots, a_{[n],m}) & \mapsto vec_n(\prod_{i=1}^{m} nat_n(a_{[n],i})) \end{array}$$

(Example: $[\![\prod_{3*[4][4]}(1100, 1100, 0110)]\!] = (0000)$)

The meaning of some bitvector function symbols can be explained graphically as in table B.2.

B.3 Rewrite Rules

The rewrite rules used for constant folding, partial normalization and symmetry reduction are shown in the following Tables. (Let $norm(\mathbf{c}_{[m]}) = vec_{[\lceil ld(nat_m(\mathbf{c}_{[m]}))\rceil]}(nat_m(\mathbf{c}_{[m]}))$.)

The second and the third rule for ite in Table B.8 mean basically the following, $\neg(\text{ite}(\mathbf{x} < \mathbf{y}, \mathbf{x}, \mathbf{y})) \to max(\mathbf{x}, \mathbf{y})$, and $\neg(\text{ite}(\mathbf{x} < \mathbf{y}, \mathbf{y}, \mathbf{x})) \to min(\mathbf{x}, \mathbf{y})$, however, min and max are not part of the bitvector logic (yet). (The permutation term sort graphs for these constructs are then isomorphic.)

Table B.3: Constant propagation rules 1

Binary bitwise conjunction:
$\mathbf{x}_{[n]} \wedge \mathbf{0}_{[n]} \rightarrow \mathbf{0}_{[n]}$
$\mathbf{0}_{[n]} \wedge \mathbf{x}_{[n]} \rightarrow \mathbf{0}_{[n]}$
$\mathbf{x}_{[n]} \wedge (11\cdots1)_{[n]} \rightarrow \mathbf{x}_{[n]}$
$(11\cdots1)_{[n]} \wedge \mathbf{x}_{[n]} \rightarrow \mathbf{x}_{[n]}$
Binary bitwise disjunction:
$\mathbf{x}_{[n]} \vee \mathbf{0}_{[n]} \rightarrow \mathbf{x}_{[n]}$
$\mathbf{0}_{[n]} \vee \mathbf{x}_{[n]} \rightarrow \mathbf{x}_{[n]}$
$\mathbf{x}_{[n]} \vee (11\cdots1)_{[n]} \rightarrow (11\cdots1)_{[n]}$
$(11\cdots1)_{[n]} \vee \mathbf{x}_{[n]} \rightarrow (11\cdots1)_{[n]}$
Binary bitwise anti valence:
$\mathbf{x}_{[n]} \oplus \mathbf{0}_{[n]} \rightarrow \mathbf{x}_{[n]}$
$\mathbf{0}_{[n]} \oplus \mathbf{x}_{[n]} \rightarrow \mathbf{x}_{[n]}$
$\mathbf{x}_{[n]} \oplus (11\cdots1)_{[n]} \rightarrow \neg\mathbf{x}_{[n]}$
$(11\cdots1)_{[n]} \oplus \mathbf{x}_{[n]} \rightarrow \neg\mathbf{x}_{[n]}$
Addition:
$\mathbf{x}_{[n]} + \mathbf{0}_{[n]} \rightarrow \mathbf{x}_{[n]}$
$\mathbf{0}_{[n]} + \mathbf{x}_{[n]} \rightarrow \mathbf{x}_{[n]}$
Multiplication:
$\mathbf{x}_{[n]} * \mathbf{0}_{[n]} \rightarrow \mathbf{0}_{[n]}$
$\mathbf{0}_{[n]} * \mathbf{x}_{[n]} \rightarrow \mathbf{0}_{[n]}$
$\mathbf{x}_{[n]} * \mathbf{1}_{[n]} \rightarrow \mathbf{x}_{[n]}$
$\mathbf{1}_{[n]} * \mathbf{x}_{[n]} \rightarrow \mathbf{x}_{[n]}$
Division:
$\mathbf{x}_{[n]} / \mathbf{0}_{[n]} \rightarrow \mathbf{0}_{[n]}$
$\mathbf{0}_{[n]} / \mathbf{x}_{[n]} \rightarrow \mathbf{0}_{[n]}$
Modulo:
$\mathbf{x}_{[n]} \% \mathbf{0}_{[n]} \rightarrow \mathbf{0}_{[n]}$
$\mathbf{0}_{[n]} \% \mathbf{x}_{[n]} \rightarrow \mathbf{0}_{[n]}$
Equality:
$\mathbf{x}_{[1]} = \mathbf{1}_{[1]} \rightarrow \mathbf{x}_{[1]}$
$\mathbf{1}_{[1]} = \mathbf{x}_{[1]} \rightarrow \mathbf{x}_{[1]}$
$\mathbf{x}_{[1]} = \mathbf{0}_{[1]} \rightarrow \neg\mathbf{x}_{[1]}$
$\mathbf{0}_{[1]} = \mathbf{x}_{[1]} \rightarrow \neg\mathbf{x}_{[1]}$

Table B.4: Constant propagation rules 2

Rotate-Right:	
	$\mathbf{x}_{[n]} \circlearrowright \mathbf{0}_{[m]} \rightarrow \mathbf{x}_{[n]}$
$norm(\mathbf{c}_{[m+n]}\%vec_{m+n}(n)) \neq \mathbf{c}_{[m]} \Rightarrow$	
	$\mathbf{x}_{[n]} \circlearrowright \mathbf{c}_{[m]} \rightarrow \mathbf{x}_{[n]} \circlearrowright norm(c_{[m+n]}\%vec_{m+n}(n))$
Rotate-Left:	
	$\mathbf{x}_{[n]} \circlearrowleft \mathbf{0}_{[m]} \rightarrow \mathbf{x}_{[n]}$
$norm(\mathbf{c}_{[m+n]}\%vec_{m+n}(n)) \neq \mathbf{c}_{[m]} \Rightarrow$	
	$\mathbf{x}_{[n]} \circlearrowleft \mathbf{c}_{[m]} \rightarrow \mathbf{x}_{[n]} \circlearrowleft norm(\mathbf{c}_{[m+n]}\%vec_{m+n}(n))$
Shift-Right:	
	$\mathbf{x}_{[n]} \gg \mathbf{0}_{[m]} \rightarrow \mathbf{x}_{[n]}$
	$(nat_m(\mathbf{c}_{[m]}) >= n) \Rightarrow \mathbf{x}_{[n]} \gg \mathbf{c}_{[m]} \rightarrow \mathbf{0}_{[n]}$
Shift-Left:	
	$\mathbf{x}_{[n]} \ll \mathbf{0}_{[m]} \rightarrow \mathbf{x}_{[n]}$
	$(nat_m(\mathbf{c}_{[m]}) >= n) \Rightarrow \mathbf{x}_{[n]} \ll \mathbf{c}_{[m]} \rightarrow \mathbf{0}_{[n]}$
Less:	
	$(11\cdots1)_{[n]} < \mathbf{x}_{[n]} \rightarrow \mathbf{0}_{[1]}$
	$\mathbf{x}_{[n]} < \mathbf{0}_{[n]} \rightarrow \mathbf{0}_{[1]}$
Slice:	
	$\mathbf{x}_{[n]}[\mathbf{0}_{[m]}]_{[n]} \rightarrow \mathbf{x}_{[n]}$
	$(\mathbf{x}_{[n]}[\mathbf{a}_{[m]}]_{[o]})[\mathbf{b}_{[k]}]_{[l]} \rightarrow \mathbf{x}_{[n]}[vec(nat(\mathbf{a}) + nat(\mathbf{b}))]_{[l]}$
	$nat(\mathbf{a}) + o \leq k \Rightarrow (\mathbf{x}_{[n]} \otimes \mathbf{y}_{[k]})[\mathbf{a}]_{[o]} \rightarrow \mathbf{y}_{[k]}[\mathbf{a}]_{[o]}$
	$nat(\mathbf{a}) \geq k \Rightarrow (\mathbf{x}_{[n]} \otimes \mathbf{y}_{[k]})[\mathbf{a}]_{[o]} \rightarrow \mathbf{x}_{[n]}[\mathbf{a}]_{[o]}$
$nat(\mathbf{a}) < k \wedge nat(\mathbf{a}) + o > k \Rightarrow$	
	$(\mathbf{x}_{[n]} \otimes \mathbf{y}_{[k]})[\mathbf{a}]_{[o]} \rightarrow \mathbf{x}_{[n]}[\mathbf{0}_{[1]}]_{[o-k+nat(\mathbf{a})]}$
	$\otimes \mathbf{y}_{[k]}[\mathbf{a}]_{[k-nat(\mathbf{a})]}$
Multiplex read:	
	$\mathrm{read}(\mathbf{x}_{[n]}, \mathbf{c}_{[m]})_{[o]} \rightarrow \mathbf{x}_{[n]}[vec_n(nat_m(\mathbf{c}_{[m]}) * o)]_{[o]}$
If-Then-Else:	
	$\mathrm{ite}(\mathbf{1}_{[1]}, \mathbf{x}_{[n]}, \mathbf{y}_{[n]}) \rightarrow \mathbf{x}_{[n]}$
	$\mathrm{ite}(\mathbf{0}_{[1]}, \mathbf{x}_{[n]}, \mathbf{y}_{[n]}) \rightarrow \mathbf{y}_{[n]}$
	$\mathrm{ite}(\mathbf{x}_{[1]}, \mathbf{0}_{[1]}, \mathbf{x}_{[1]}) \rightarrow \mathbf{0}_{[1]}$
	$\mathrm{ite}(\mathbf{x}_{[1]}, \mathbf{x}_{[1]}, \mathbf{0}_{[1]}) \rightarrow \mathbf{x}_{[1]}$
	$\mathrm{ite}(\mathbf{x}_{[1]}, \mathbf{1}_{[1]}, \mathbf{x}_{[1]}) \rightarrow \mathbf{x}_{[1]}$
	$\mathrm{ite}(\mathbf{x}_{[1]}, \mathbf{x}_{[1]}, \mathbf{1}_{[1]}) \rightarrow \mathbf{1}_{[1]}$
	$\mathrm{ite}(\neg\mathbf{x}_{[1]}, \mathbf{0}_{[1]}, \mathbf{x}_{[1]}) \rightarrow \mathbf{x}_{[1]}$
	$\mathrm{ite}(\neg\mathbf{x}_{[1]}, \mathbf{x}_{[1]}, \mathbf{0}_{[1]}) \rightarrow \mathbf{0}_{[1]}$
	$\mathrm{ite}(\neg\mathbf{x}_{[1]}, \mathbf{1}_{[1]}, \mathbf{x}_{[1]}) \rightarrow \mathbf{1}_{[1]}$
	$\mathrm{ite}(\neg\mathbf{x}_{[1]}, \mathbf{x}_{[1]}, \mathbf{1}_{[1]}) \rightarrow \mathbf{x}_{[1]}$
Multiplex write:	
	$(\mathbf{a}_{[m]} = \mathbf{0}_{[m]}) \Rightarrow \mathrm{write}(\mathbf{x}_{[n]}, \mathbf{a}_{[m]}, \mathbf{z}_{[o]}) \rightarrow \mathbf{x}_{[n]}[vec_o(o)]_{[n-o]} \otimes \mathbf{z}_{[o]}$
	$(nat(\mathbf{a}_{[m]}) = n/o - 1) \Rightarrow \mathrm{write}(\mathbf{x}_{[n]}, \mathbf{a}_{[m]}, \mathbf{z}_{[o]}) \rightarrow \mathbf{z}_{[o]} \otimes \mathbf{x}_{[n]}[\mathbf{0}_{[o]}]_{[n-o]}$
	$(0 < nat(\mathbf{a}_{[m]}) < n/o - 1) \Rightarrow \mathrm{write}(\mathbf{x}_{[n]}, \mathbf{a}_{[m]}, \mathbf{z}_{[o]}) \rightarrow$
	$\mathbf{x}_{[n]}[vec_m((nat(\mathbf{a}_{[m]}) + 1) * o)]_{[n-(nat(\mathbf{a}_{[m]})+1)*o]} \otimes$
	$\mathbf{z}_{[o]} \otimes \mathbf{x}_{[n]}[\mathbf{0}_{[o]}]_{[nat(\mathbf{a}_{[m]})*o]}$

Table B.5: Normalization rules 1

Negation:
$\neg(\neg\mathbf{x}_{[n]}) \rightarrow \mathbf{x}_{[n]}$
Unary minus:
$-(-\mathbf{x}_{[n]}) \rightarrow \mathbf{x}_{[n]}$
$-(\mathbf{x}_{[n]}) \rightarrow (\neg(\mathbf{x}_{[n]})) + \mathbf{1}_{[n]}$
Unary bitwise conjunction:
$\wedge\mathbf{x}_{[1]} \rightarrow \mathbf{x}_{[1]}$
Unary bitwise disjunction:
$\vee\mathbf{x}_{[1]} \rightarrow \mathbf{x}_{[1]}$
Unary bitwise anti valence:
$\oplus\mathbf{x}_{[1]} \rightarrow \mathbf{x}_{[1]}$
Zero-Extend:
$\mathrm{ze}_{[n]}\mathbf{x}_{[n]} \rightarrow \mathbf{x}_{[n]}$
Sign-Extend:
$\mathrm{se}_{[n]}\mathbf{x}_{[n]} \rightarrow \mathbf{x}_{[n]}$
Implication:
$\mathbf{x}_{[1]} \Rightarrow \mathbf{y}_{[1]} \rightarrow \neg\mathbf{x}_{[1]} \vee \mathbf{y}_{[1]}$
Binary bitwise conjunction:
$\mathbf{x} \wedge \mathbf{x} \rightarrow \mathbf{x}$
$\mathbf{x}_{[n]} \wedge (\neg\mathbf{x}_{[n]}) \rightarrow \mathbf{0}_{[n]}$
$(\neg\mathbf{x}_{[n]}) \wedge \mathbf{x}_{[n]} \rightarrow \mathbf{0}_{[n]}$
$(\mathbf{y} <_{rko} \mathbf{x}) \Rightarrow \mathbf{x} \wedge \mathbf{y} \rightarrow \mathbf{y} \wedge \mathbf{x}$
Binary bitwise disjunction:
$\mathbf{x} \vee \mathbf{x} \rightarrow \mathbf{x}$
$\mathbf{x}_{[n]} \vee (\neg\mathbf{x}_{[n]}) \rightarrow (11\cdots1)_{[n]}$
$(\neg\mathbf{x}_{[n]}) \vee \mathbf{x}_{[n]} \rightarrow (11\cdots1)_{[n]}$
$(\mathbf{y} <_{rko} \mathbf{x}) \Rightarrow \mathbf{x} \vee \mathbf{y} \rightarrow \mathbf{y} \vee \mathbf{x}$
Binary bitwise anti valence:
$\mathbf{x}_{[n]} \oplus \mathbf{x}_{[n]} \rightarrow \mathbf{0}_{[n]}$
$\mathbf{x}_{[n]} \oplus (\neg\mathbf{x}_{[n]}) \rightarrow (11\cdots1)_{[n]}$
$(\neg\mathbf{x}_{[n]}) \oplus \mathbf{x}_{[n]} \rightarrow (11\cdots1)_{[n]}$
$(\mathbf{y} <_{rko} \mathbf{x}) \Rightarrow \mathbf{x} \oplus \mathbf{y} \rightarrow \mathbf{y} \oplus \mathbf{x}$

Table B.6: Normalization rules 2

Addition:	
	$\mathbf{x}_{[1]} + \mathbf{y}_{[1]} \rightarrow \mathbf{x}_{[1]} \oplus \mathbf{y}_{[1]}$
	$(\mathbf{y} <_{rko} \mathbf{x}) \Rightarrow \mathbf{x} + \mathbf{y} \rightarrow \mathbf{y} + \mathbf{x}$
Multiplication:	
	$(\exists s \in \mathbb{N}_n : 2^s = nat_n(c_{[n]})) \Rightarrow \mathbf{c}_{[n]} * \mathbf{x}_{[n]} \rightarrow \mathbf{x}_{[n]} \ll vec_n(s)$
	$(\exists s \in \mathbb{N}_n : 2^s = nat_n(c_{[n]})) \Rightarrow \mathbf{x}_{[n]} * \mathbf{c}_{[n]} \rightarrow \mathbf{x}_{[n]} \ll vec_n(s)$
	$\mathbf{x}_{[1]} * \mathbf{y}_{[1]} \rightarrow \mathbf{x}_{[1]} \wedge \mathbf{y}_{[1]}$
	$(\mathbf{y} <_{rko} \mathbf{x}) \Rightarrow \mathbf{x} * \mathbf{y} \rightarrow \mathbf{y} * \mathbf{x}$
Division:	
	$(\exists s \in \mathbb{N}_n : 2^s = nat_n(c_{[n]})) \Rightarrow \mathbf{x}_{[n]} / \mathbf{c}_{[n]} \rightarrow \mathbf{x}_{[n]} \gg vec_n(s)$
Modulo:	
(??)	$(\exists s \in \mathbb{N}_n : 2^s = nat_n(c_{[n]})) \Rightarrow \mathbf{x}_{[n]} \% \mathbf{c}_{[n]} \rightarrow \mathbf{x}_{[n]} \wedge (\neg(\mathbf{0}_{[s-1]}) \otimes \mathbf{0}_{[n-s+1]})$
Equality:	
	$\mathbf{x} = \mathbf{x} \rightarrow \mathbf{1}_{[1]}$
	$\mathbf{x} = \neg\mathbf{x} \rightarrow \mathbf{0}_{[1]}$
	$\neg\mathbf{x} = \mathbf{x} \rightarrow \mathbf{0}_{[1]}$
	$(\mathbf{y} <_{rko} \mathbf{x}) \Rightarrow \mathbf{x} = \mathbf{y} \rightarrow \mathbf{y} = \mathbf{x}$
Concatenation:	
	$\text{ite}(\mathbf{x}_{[1]}, \mathbf{yl}_{[n]}, \mathbf{zl}_{[n]}) \otimes \text{ite}(\mathbf{x}_{[1]}, \mathbf{yr}_{[m]}, \mathbf{zr}_{[m]}) \rightarrow \text{ite}(\mathbf{x}_{[1]}, \mathbf{yl}_{[n]} \otimes \mathbf{yr}_{[m]}, \mathbf{zl}_{[n]} \otimes \mathbf{zr}_{[m]})$
Slice:	
	$(\mathbf{a}_{[m]} \neq norm(\mathbf{a}_{[m]})) \Rightarrow \mathbf{x}_{[n]}[\mathbf{a}_{[m]}]_{[o]} \rightarrow \mathbf{x}_{[n]}[norm(\mathbf{a}_{[m]})]_{[o]}$
Multiplex read:	
	$\text{read}(\mathbf{x}_{[n]}, \mathbf{y}_{[m]})_{[1]} \rightarrow \mathbf{x}_{[n]}[\mathbf{y}_{[m]}]_{[1]}$
	$\text{read}(\text{write}(\mathbf{x}_{[n]}, \mathbf{y}_{[m]}, \mathbf{z}_{[o]})_{[n]}), \mathbf{y}_{[m]})_{[o]} \rightarrow \mathbf{z}_{[o]}$
If-Then-Else:	
	$\text{ite}(\mathbf{x}_{[1]}, \mathbf{y}_{[n]}, \mathbf{y}_{[n]}) \rightarrow \mathbf{y}_{[n]}$
Multiplex write:	
	$\text{write}(\mathbf{x}_{[n]}, \mathbf{y}_{[m]}, \text{read}(\mathbf{x}_{[n]}, \mathbf{y}_{[m]})_{[o]})_{[n]} \rightarrow \mathbf{x}_{[n]}$

Table B.7: Normalization rules 3

Negation:
$\neg(\mathbf{x}_{[n]} + \mathbf{y}_{[n]}) \rightarrow \neg\mathbf{x}_{[n]} + \neg\mathbf{y}_{[n]} + \mathbf{1}_{[n]}$
Unary bitwise conjunction:
$\wedge(\mathbf{x} \otimes \mathbf{y}) \rightarrow \wedge(\mathbf{x}) \wedge \wedge(\mathbf{y})$
Unary bitwise disjunction:
$\vee(\mathbf{x} \otimes \mathbf{y}) \rightarrow \vee(\mathbf{x}) \vee \vee(\mathbf{y})$
Unary bitwise anti valence:
$\oplus(\mathbf{x} \otimes \mathbf{y}) \rightarrow \oplus(\mathbf{x}) \oplus \oplus(\mathbf{y})$
Minus:
$-\mathbf{x}_{[n]} \rightarrow \neg\mathbf{x}_{[n]} + \mathbf{1}_{[n]}$
Addition:
$\mathbf{x}_{[n]} + \neg\mathbf{x}_{[n]} \rightarrow \mathbf{1}_{[n]}$
$\mathbf{a} + (\mathbf{x} + \mathbf{b}) \rightarrow (\mathbf{a} + \mathbf{b}) + \mathbf{x}$
$\mathbf{a} + (\mathbf{b} + \mathbf{x}) \rightarrow (\mathbf{a} + \mathbf{b}) + \mathbf{x}$
$(\mathbf{a} + \mathbf{x}) + \mathbf{b} \rightarrow (\mathbf{a} + \mathbf{b}) + \mathbf{x}$
$(\mathbf{x} + \mathbf{a}) + \mathbf{b} \rightarrow (\mathbf{a} + \mathbf{b}) + \mathbf{x}$
Multiplication:
$\mathbf{a} * (\mathbf{x} * \mathbf{b}) \rightarrow (\mathbf{a} * \mathbf{b}) * \mathbf{x}$
$\mathbf{a} * (\mathbf{b} * \mathbf{x}) \rightarrow (\mathbf{a} * \mathbf{b}) * \mathbf{x}$
$(\mathbf{a} * \mathbf{x}) * \mathbf{b} \rightarrow (\mathbf{a} * \mathbf{b}) * \mathbf{x}$
$(\mathbf{x} * \mathbf{a}) * \mathbf{b} \rightarrow (\mathbf{a} * \mathbf{b}) * \mathbf{x}$
Equality:
$\mathbf{x}_{[n+m]} = \mathbf{y}_{[n]} \otimes \mathbf{z}_{[m]} \rightarrow (x_{[n+m]}[vec(m)]_{[n]} = \mathbf{y}_{[n]}) \wedge (x_{[n+m]}[\mathbf{0}]_{[m]} = \mathbf{z}_{[m]})$
$\mathbf{y}_{[n]} \otimes \mathbf{z}_{[m]} = \mathbf{x}_{[n+m]} \rightarrow (\mathbf{y}_{[n]} = x_{[n+m]}[vec(m)]_{[n]}) \wedge (\mathbf{z}_{[m]} = x_{[n+m]}[\mathbf{0}]_{[m]})$
Shift & Rotate:
$(nat(\mathbf{a}) < n) \Rightarrow \mathbf{x}_{[n]} \ll \mathbf{a}_{[m]} \rightarrow \mathbf{x}_{[n]}[\mathbf{0}]_{[n-nat(\mathbf{a})]} \otimes \mathbf{0}_{[nat(\mathbf{a})]}$
$(nat(\mathbf{a}) < n) \Rightarrow \mathbf{x}_{[n]} \gg \mathbf{a}_{[m]} \rightarrow \mathbf{0}_{[nat(\mathbf{a})]} \otimes \mathbf{x}_{[n]}[\mathbf{0}]_{[n-nat(\mathbf{a})]}$
$(nat(\mathbf{a}) < n) \Rightarrow \mathbf{x}_{[n]} \circlearrowleft \mathbf{a}_{[m]} \rightarrow \mathbf{x}_{[n]}[\mathbf{0}]_{[n-nat(\mathbf{a})]} \otimes \mathbf{x}_{[n]}[\mathbf{a}]_{[nat(\mathbf{a})]}$
$(nat(\mathbf{a}) < n) \Rightarrow \mathbf{x}_{[n]} \circlearrowright \mathbf{a}_{[m]} \rightarrow \mathbf{x}_{[n]}[\mathbf{a}]_{[nat(\mathbf{a})]} \otimes \mathbf{x}_{[n]}[\mathbf{0}]_{[n-nat(\mathbf{a})]}$

Table B.8: Normalization rules 4

Ite:

$$\begin{aligned}
\text{ite}(\neg(\mathbf{x} < \mathbf{y}), \mathbf{x}, \mathbf{y})) &\rightarrow \text{ite}(\mathbf{y} < \mathbf{x}, \mathbf{x}, \mathbf{y}) \vee \text{ite}(\mathbf{x} < \mathbf{y}, \mathbf{y}, \mathbf{x}) \vee (\mathbf{x} \wedge \mathbf{y}) \\
\text{ite}(\neg(\mathbf{x} < \mathbf{y}), \mathbf{y}, \mathbf{x})) &\rightarrow \text{ite}(\mathbf{y} < \mathbf{x}, \mathbf{y}, \mathbf{x}) \vee \text{ite}(\mathbf{x} < \mathbf{y}, \mathbf{x}, \mathbf{y}) \vee (\mathbf{x} \wedge \mathbf{y})
\end{aligned}$$

Slice of logic:

$$\begin{aligned}
(\neg\mathbf{x})[\mathbf{y}]_{[o]} &\rightarrow \neg(\mathbf{x}[\mathbf{y}]_{[o]}) \\
(\mathbf{x} \wedge \mathbf{y})[\mathbf{a}]_{[o]} &\rightarrow (\mathbf{x}[\mathbf{a}]_{[o]}) \wedge (\mathbf{y}[\mathbf{a}]_{[o]}) \\
(\mathbf{x} \vee \mathbf{y})[\mathbf{a}]_{[o]} &\rightarrow (\mathbf{x}[\mathbf{a}]_{[o]}) \vee (\mathbf{y}[\mathbf{a}]_{[o]}) \\
(\mathbf{x} \oplus \mathbf{y})[\mathbf{a}]_{[o]} &\rightarrow (\mathbf{x}[\mathbf{a}]_{[o]}) \oplus (\mathbf{y}[\mathbf{a}]_{[o]} \\
\text{ite}(\mathbf{x}, \mathbf{y}, \mathbf{z})[\mathbf{a}]_{[o]} &\rightarrow \text{ite}(\mathbf{x}, \mathbf{y}[\mathbf{a}]_{[o]}, \mathbf{z}[\mathbf{a}]_{[o]})
\end{aligned}$$

Slice of Arithmetic:

$$\begin{aligned}
(\mathbf{x} + \mathbf{y})[\mathbf{0}]_{[o]} &\rightarrow \mathbf{x}[\mathbf{0}]_{[o]} + \mathbf{y}[\mathbf{0}]_{[o]} \\
(\mathbf{x} * \mathbf{y})[\mathbf{0}]_{[o]} &\rightarrow \mathbf{x}[\mathbf{0}]_{[o]} * \mathbf{y}[\mathbf{0}]_{[o]}
\end{aligned}$$

Slice of Zero Extend:

$$\begin{aligned}
(n \geq nat(\mathbf{a}) + o) &\Rightarrow (\text{ze}_{[k]}\mathbf{x}_{[n]})[\mathbf{a}_{[m]}]_{[o]} \rightarrow \mathbf{x}_{[n]}[\mathbf{a}_{[m]}]_{[o]} \\
(n \geq nat(\mathbf{a})) &\Rightarrow (\text{ze}_{[k]}\mathbf{x}_{[n]})[\mathbf{a}_{[m]}]_{[o]} \rightarrow \mathbf{0}_{[o]} \\
(nat(\mathbf{a}) = 0 \wedge k \geq 0) &\Rightarrow (\text{ze}_{[k]}\mathbf{x}_{[n]})[\mathbf{a}_{[m]}]_{[o]} \rightarrow \text{ze}_{[o]}\mathbf{x}_{[n]} \\
(nat(\mathbf{a}) < n < nat(\mathbf{a}) + o) &\Rightarrow (\text{ze}_{[k]}\mathbf{x}_{[n]})[\mathbf{a}_{[m]}]_{[o]} \rightarrow (\text{ze}_{[o]}\mathbf{x}_{[n]})[\mathbf{a}_{[m]}]_{[n-nat(\mathbf{a})]}
\end{aligned}$$

Slice of Sign Extend:

$$\begin{aligned}
(n \geq nat(\mathbf{a}) + o) &\Rightarrow (\text{se}_{[k]}\mathbf{x}_{[n]})[\mathbf{a}_{[m]}]_{[o]} \rightarrow \mathbf{x}_{[n]}[\mathbf{a}_{[m]}]_{[o]} \\
(n \geq nat(\mathbf{a})) &\Rightarrow (\text{se}_{[k]}\mathbf{x}_{[n]})[\mathbf{a}_{[m]}]_{[o]} \rightarrow \text{se}_{[o]}(\mathbf{x}_{[n]}[\mathbf{a}_{[m]}]_{[1]}) \\
(nat(\mathbf{a}) = 0 \wedge k \geq 0) &\Rightarrow (\text{se}_{[k]}\mathbf{x}_{[n]})[\mathbf{a}_{[m]}]_{[o]} \rightarrow \text{se}_{[o]}\mathbf{x}_{[n]} \\
(nat(\mathbf{a}) < n < nat(\mathbf{a}) + o) &\Rightarrow (\text{se}_{[k]}\mathbf{x}_{[n]})[\mathbf{a}_{[m]}]_{[o]} \rightarrow (\text{se}_{[o]}\mathbf{x}_{[n]})[\mathbf{a}_{[m]}]_{[n-nat(\mathbf{a})]}
\end{aligned}$$

Concatenation:

$$\begin{aligned}
\neg\mathbf{x} \otimes \neg\mathbf{y} &\rightarrow \neg(\mathbf{x} \otimes \mathbf{y}) \\
(\mathbf{x} \otimes \mathbf{a}) \otimes \mathbf{b} &\rightarrow \mathbf{x} \otimes (\mathbf{a} \otimes \mathbf{b}) \\
\mathbf{a} \otimes (\mathbf{b} \otimes \mathbf{x}) &\rightarrow (\mathbf{a} \otimes \mathbf{b}) \otimes \mathbf{x} \\
(nat(\mathbf{a}) = nat(\mathbf{a}) + l) \Rightarrow \mathbf{x}_{[n]}[\mathbf{a}]_{[o]} \otimes \mathbf{x}_{[n]}[\mathbf{b}]_{[l]} &\rightarrow \mathbf{x}_{[n]}[\mathbf{b}]_{[l+o]} \\
\neg\mathbf{x} \otimes \mathbf{a} &\rightarrow \neg(\mathbf{x} \otimes \neg\mathbf{a}) \\
\mathbf{a} \otimes \neg\mathbf{x} &\rightarrow \neg(\neg\mathbf{a} \otimes \mathbf{x}) \\
(\mathbf{x}_{[m]} + \mathbf{y}_{[m]}) \otimes \mathbf{a}_{[n]} &\rightarrow (\mathbf{x}_{[m]} \otimes \mathbf{0}_{[n]}) + (\mathbf{y}_{[m]} \otimes \mathbf{0}_{[n]}) + \text{ze}_{[m+n]}(\mathbf{a}_{[n]}) \\
(\mathbf{a}_{[n]} \neq \mathbf{0}_{[n]}) \Rightarrow \mathbf{x}_{[m]} \otimes \mathbf{a}_{[n]} &\rightarrow (\mathbf{x}_{[m]} \otimes \mathbf{0}_{[n]}) + \text{ze}_{[m+n]}\mathbf{a}_{[n]}
\end{aligned}$$

Appendix C

Basic Algorithms

C.1 Algorithmic Notation

Algorithms are denoted in pseudo code and in an object oriented fashion. The following data structures are used frequently throughout algorithms.

1. The data types `Variable`, `Value`, `Term`, and `Formula` represent variables, values, terms and formulas respectively.

2. The class `Set<T>` represents a set of objects of type `T`, and has the following basic operations:

   (a) `void Set<T>::insert(<T> t)`, which add elements to this set,

   (b) `void Set<T>::remove(<T> t)`, removes an element from this set,

   (c) `Boolean Set<T>::empty()` returns `true` if this set is empty, otherwise `false`, and

   (d) `Boolean Set<T>::find(<T> t)` returns true if `t` is in this set, otherwise `false`.

   (e) `<T> Set<T>::choose()` returns any element of this set at.

   (f) `Set<T> Set<T>::minus(Set<T> S)` returns this set, minus the set `S`.

3. The class `Map<S,T>` represents a map from objects of type `S` to objects of type `T`. The operations are:

   (a) `T Map<S,T>::get(<S> s)`, which returns the map entry for `s`,

   (b) `void Map<S,T>::add(<S> s, <T> t)`, which sets the entry for `s` to `t`,

   (c) `void Map<S,T>::remove(<S> s)`, which removes the entry for `s`.

   For a map `m` and objects `x` and `y` of type `S` and `T`, `m.get(x)` and `m.set(x,y)` are also denoted as `m[x]` and `m[x]=y`, respectively.

4. The following types are defined for readability:

```
typedef Set<Value> Values;
typedef Set<Value> Domain;
typedef Set<Term> Terms;
typedef Set<Formula> Formulas;
typedef Map<Variable,Value> Assignment;
```

C.2 The Partition Class

Equivalence relations play an important role within this thesis. They are naturally represented by partitions. Here the basic algorithms for representing and merging equivalence classes are provided.

The class `Partition<T>` is frequently used. It represents a partition of a set with elements of type `T`. It implicitly requires a total (well-founded) order on `T`. It is shown in Algorithm 10. The function `insert` (Algorithm 11) adds a singleton cell containing `t` to the partition, while `find` (Algorithm 12) returns the minimum cell representative (mcr) of the cell containing `t`. The function `unite` (Algorithm 14) merges the cells containing `s` and `t` and returns the mcr of the merged cell. The function `equivalent` (Algorithm 13) returns `true` iff `s` and `t` are in the same equivalence class, and the function `Set<T> Partition<T>::reps()` (Algorithm 15) returns the set of minimum cell representatives of the partition. (Note that using intrusive list pointers in `<T>` instead as a `Map<<T>,<T>>`, all these functions have linear complexity.) `Set<T> cell(<T> t)` (Algorithm 16) returns the cell containing `t`. The function `merge(Partition<T> Q)` (Algorithm 17) merges the partition with `Q`. If this partition is `P`, it computes `P` $\sqcup$ `Q`.

Algorithm 10 Partitioning of Objects w.r.t. a Total Order

```
class Partition<T>
{
  private:
    Set<T> S;                          // Set of elemets.
    Map< <T>, <T> > M;                 // Map of elements to smaller ones
                                       // of the same equivalence class.
  public:
    void insert(<T> t);                // Add singleton cell containing t.
    <T> find(<T> t);                   // Return representative of [t].
    <T> unite(<T> t1, <T> t2);         // Unite [t1] and [t2] and
                                       // return representative.
    Boolean equivalent(<T> t1,<T> t2); // Check whether [t1]=[t2].
    Set<T> reps();                     // Return set of (minimum) cell
                                       // representatives.
    Set<T> cell(<T> t);                // Return cell of t.
    void merge(Partition<T> Q);        // Merge this parition with Q.
}
```

Algorithm 11 Partition Insert

```
// Insert element into partition as trivial cell
void Partition<T>::insert(<T> t)
{
  S.insert(t);
  M[t] = t;
}
```

Algorithm 12 Partition Find

```
// Find representative of equivalence class for an object
<T> Partition<T>::find(<T> t)
{
  if (! S.find(t) )
    return t;
  <T> tn = M[t];
  while (tn != t)
  {
    t = tn;
    tn = M[t];
  }
  return t;
}
```

Algorithm 13 Partition Equivalent

```
// Check whether two objects are in the same partition.
<T> Partition<T>::equivalent(<T> t1, <T> t2)
{
  if ( find(t1) == find(t2) )
    return true;
  return false;
}
```

Algorithm 14 Partition Unite

```
// Unite equivalence classes of two objects and return new representative
<T> Partition<T>::unite(<T> t1, <T> t2)
{
  <T> u1 = find(t1);
  <T> u2 = find(t2);
  if (u1 < u2)
  {
    M[u2] = u1;
    return u1;
  }
  else if (u2 < u1)
  {
    M[u1] = u2;
    return u2;
  }
  return u1;
}
```

Algorithm 15 Partition Representatives

```
// Return set of minimum cell representatives
Set<T> Partition<T>::reps()
{
  Set <T> R;
  foreach (<T> t in S)
  {
    if (t == find(t))
      R.insert(t);
  }
  return R;
}
```

Algorithm 16 Partition Cell

```
Set<T> Partition<T>::cell(<T> t)
{
  Set <T> R;
  foreach (<T> s in S)
  {
    if (equiv(s,t))
      R.insert(s);
  }
  return R;
}
```

Algorithm 17 Partition Merge

```
void Partition<T>::merge(Partition<T> Q)
{
  foreach (<T> t in Q.reps())
  {
     foreach (r in Q.cell(t))
     {
        unite(r,t);
     }
  }
}
```

C.3 Equivalence Relations

If an algorithm `Boolean equiv(<T> ta, <T> tb)` deciding the equivalence of two objects `ta` and `tb` of type `<T>`, for example terms, then Algorithm 18 computes the induced equivalence relation for a given set `S` of objects of type `<T>` as partition of `T`. The algorithm

Algorithm 18 Equivalence Relation

```
Boolean EquivRel(Set<T> S, Partition<T>& P)
{
  foreach (<T> t in S )              // Initialize P to be
    P.insert({t});                   // the unit partition of S.
  Set <T> SS;                        // Set of treated t's (empty).
  foreach (<T> ta in S.minus(SS))    // For all untreated t's ta
    SS.insert(ta);                   // let ta be treated.
    foreach (<T> tb in S.minus(SS))  // For all untreated tb
    {                                // the pair ta and tb is checked:
      if ( ! equiv(ta,tb) )          // If ta and tb are not equivalent
        continue;                    // then try the next pair.
      <T> tc = P.unite(ta,tb);       // Otherwise unite [ta] and [tb] in P.
      if (tc == tb)                  // Then either tc = tb or tc = ta.
         break;                      // If tc = tb, then take the next ta.
      SS.insert(tb);                 // Otherwise tb is treated, and
    }                                // the next tb is taken.
  if (P.reps().size() == 1)          // If there is just one cell left
    return true;                     // then all t's are equivalent.
  else                               // Otherwise there
    return false;                    // are at least two cells in P.
}
```

works as follows. At the beginning the set of objects `S` to be considered is the set of minimum cell representatives `P.reps()` of the unit partition `P` of `S`. The set `SS` will hold all objects for which all relevant pairs have been checked, and is initially empty. For each object `ta`, untreated so far, first `ta` is set to treated by adding it to `SS` to make sure it is not considered twice. Then for all remaining untreated objects `tb` (with `ta` $\neq$ `tb`) the

pair `ta` and `tb` is checked for symmetry. If the pair is not symmetrical, the next object `tb` is considered. If they are symmetrical, the equivalence classes of `ta` and `tb` are united. The resulting representative `tc` is returned. It must be either `ta` or `tb`, since each of them represented its equivalence class. If `tb` is the new representative, `ta` is not a representative anymore and must not be checked against any other object. Therefore it is completely treated. Then the loop is broken and a new object `ta` is chosen. If `tb` is not the new representative then it must not be considered together with any other object `ta` and is therefore added to `SS`. Then a new object `tb` is checked against `ta`.

If $n = |S|$ is the number of objects considered, then Algorithm 18 calls '`Boolean equiv(<T> ta, <T> tb)`' at least n times (best case), and at most $n * (n-1)/2$ times (worst case). Then `P` is the unit partition (best case) or the discrete partition (worst case) of `S`.

Bibliography

[AH97] H. R. Andersen and H. Hulgaard. Boolean expression diagrams. In *LICS: IEEE Symposium on Logic in Computer Science*, 1997.

[AP01] J. Avenhaus and D. A. Plaisted. General algorithms for permutations in equational inference. *Journal of Automated Reasoning*, 26(3):223–268, 2001.

[ARMS02] F. Aloul, A. Ramani, I. Markov, and K. Sakallah. Solving difficult SAT instances in the presence of symmetry. In *Proc. of DAC'02*, pages 731–736, 2002.

[AT96] M. Agrawal and T. Thierauf. The boolean isomorphism problem. In *Proc. of IEEE Symposium on Foundations of Computer Science*, pages 422–430, 1996.

[Ave95] J. Avenhaus. *Reduktionssysteme*. Springer Verlag, 1995.

[Bac91] L. Bachmair. *Canonical Equational Proofs: Progress in Theoretical Computer Science*. Springer-Verlag, 1991.

[BBCZ98] S. Berezin, A. Biere, E. M. Clarke, and Y. Zhu. Combining symbolic model checking with uninterpreted functions for out-of-order processor verification. In *Proc. of FMCAD'98*, pages 369–386, 1998.

[BCC+99] A. Biere, A. Cimatti, E. M. Clarke, M. Fujita, and Y. Zhu. Symbolic model checking using SAT procedures instead of BDDs. In *Proc. of DAC'99*, pages 317–320, 1999.

[BCCZ99] A. Biere, A. Cimatti, E. M. Clarke, and Y. Zhu. Symbolic model checking without BDDs. In *Proc. of TACAS'99*, pages 193–207, 1999.

[BD02] R. Brinkmann and R. Drechsler. RTL-datapath verification using integer linear programming. In *Proc. of ASP-DAC/VLSI-Design'02*, pages 741–746, 2002.

[Ber00] J. Bergeron. *Writing Testbenches – Functional Verification of HDL Models*. Kluwer Academics Publishers, 2000.

[BFP86] C. A. Brown, L. Finkelstein, and P. Purdom. Intelligent backtracking using symmetry. In *Proc. of the Fall Joint Computer Computer Conference sponsored by IEEE Computer Society and the ACM*, pages 576–584, 1986.

[BGV99] R. E. Bryant, S. M. German, and M. N. Velev. Exploiting positive equality in a logic of equality with uninterpreted functions. In *Proc. of CAV'99*, pages 470–482, 1999.

[BM84] R. S. Boyer and J. S. Moore. *Automated Theorem Proving: After 25 Years*, chapter Proof-Checking, Theorem-Proving, and Program Verification, pages 119–132. Contemporary Mathematics. American Mathematical Society, 1984.

[BN98] F. Baader and T. Nipkov. *Term Rewriting and All That*. Cambridge University Press, 1998.

[Bri01] R. Brinkmann. Using symmetry for problem reduction in bounded-model-checking on the register-transfer-level. In *Proc. of SymCon'01, CP'01 Post-Conference Workshop*, 2001.

[BRS98] B. Borchert, D. Ranjan, and F. Stephan. On the computational complexity of some classical equivalence relations on boolean functions. *Theory of Computing Systems*, 31:679–693, 1998.

[Bry86] R. E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Transactions in Computers*, 8(35):677–691, 1986.

[BS97] R. J. Jr. Bayardo and R. C. Schrag. Using CSP look-back techniques to solve real-world SAT instances. In *Proc. of AAAI'97*, pages 203–208, 1997.

[CES83] E. M. Clarke, E. A. Emerson, and A. P. Sistla. Automatic verification of finite state concurrent systems using temporal logic specifications. In *Proc. of POPL'83*, pages 117–126, 1983.

[CFG+01] F. Copty, L. Fix, E. Giunchiglia, G. Kamhi, A. Tacchella, and M. Vardi. Benefits of bounded model checking at an industrial setting. In *Proc. of CAV'01*, pages 436–453, 2001.

[CFJ93] E. M. Clarke, T. Filkorn, and S. Jha. Exploiting symmetry in temporal logic model checking. In *Proc. of CAV'93*, pages 450–462, 1993.

[CGL92] E. M. Clarke, O. Grumberg, and D. E. Long. Model checking and abstraction. In *Proc. of POPL*, pages 343–354, 1992.

[CGLR96] J. Crawford, M. L. Ginsberg, E. M. Luks, and A. Roy. Symmetry-breaking predicates for search problems. In *Principles of Knowledge Representation and Reasoning*, pages 148–159. Morgan Kaufmann, 1996.

[CKZR02] M. Ciesielski, P. Kalla, Z. Zeng, and B. Rouzeyre. Taylor expansion diagrams: A compact, canonical representation with applications to symbolic verification. In *Proc. of DATE'02*, pages 285–291, 2002.

[CRM97] D. Cyrluk, H. Rueß, and O. Möller. An Efficient Decision Procedure for the Theory of Fixed-Sized Bit-Vectors. In *Proc. of CAV'97*, pages 60–71, 1997.

[DH78] L. L. Dornhoff and F. E. Hohn. *Applied Modern Algebra*. Macmillan Publishing, 1978.

[DH02] R. Drechsler and S. Höreth. Gatecomp: Equivalence checking of digital circuits in an industrial environment. In *Proc. of International Workshop on Boolean Problems*, pages 195–200, 2002.

[DJ94] N. Dershowitz and J.-P. Jouannaud. *Handbook of Theoretical Computer Science*, volume 2, chapter Rewrite Systems, pages 245–320. Elsevier, MIT Press, 1994.

[DLL62] M. Davis, G. Logemann, and D. Loveland. A machine program for theorem proving. *Communications of the ACM*, 5(7):394–397, July 1962.

[DP60] M. Davis and H. Putnam. A computing procedure for quantification theory. *Journal of the ACM*, 7(3):201–215, July 1960.

[Eme94] E. A. Emerson. *Handbook of Theoretical Computer Science*, volume 2, chapter Temporal and Modal Logic, pages 995–1072. Elsevier, MIT Press, 1994.

[ES93] E. A. Emerson and A. P. Sistla. Symmetry and model checking. In *Proc. of CAV'93*, pages 463–478, 1993.

[FDK98] F. Fallah, S. Devadas, and K. Keutzer. Functional Vector Generation for HDL Models Using Linear Programming and 3-Satisfiability. In *Proc. of DAC'98*, pages 528–533, 1998.

[Fit90] M. Fitting. *First-Order Logic and Automated Theorem Proving*. Springer-Verlag, 1990.

[FS83] H. Fujiwara and T. Shimono. On the acceleration of test generation algorithms. *IEEE Transactions on Computers*, C(31):1137–1144, 1983.

[Gel59] H. Gelernter. A note on syntactic symmetry and the manipulation of formal systems by machine. *Information and Control*, 2:80–89, 1959.

[HB95] R. Hojati and R. K. Brayton. Automatic datapath abstraction of hardware systems. In *Proc. of CAV'95*, pages 98–113, 1995.

[HD99] S. Höreth and R. Drechsler. Formal verification of word-level specifications. In *Proc. of DATE'99*, pages 52–58, 1999.

[HIKB96] R. Hojati, A. J. Isles, D. Kirkpatrick, and R. K. Brayton. Verification using uninterpreted functions and finite instantiations. In *Proc. of FMCAD'96*, pages 218–232, 1996.

[ID93] C. N. Ip and D. L. Dill. Better verification through symmetry. In *Proc. of CHDL'93*, volume 32 of *IFIP Transactions A: Computer Science and Technology*, pages 97–112. North-Holland, 1993.

[Joh01] P. Johannsen. BooStER: Speeding up RTL property checking of digital designs by word-level abstraction. In *Proc. of CAV'01*, pages 373–376, 2001.

[Joh03] P. Johannsen. *Speeding Up Hardware Verification by Automated Data Path Scaling.* PhD thesis, Christian-Albrechts-University of Kiel, 2003.

[KMSM02] W. Kunz, J. Marques-Silva, and S. Malik. *Logic Synthesis and Verification*, chapter SAT and ATPG: Algorithms for Boolean Decision Problems, pages 309–341. Kluwer Academic Publishers, 2002.

[LLR+00] M. Lajolo, L. Lavagno, M. Rebaudengo, M. Reorda, and M. Violante. Automatic test bench generation for simulation-based validation. In *Proc. ACM/IEEE 8th International Workshop on Hardware/Software Codesign, CODES*, 2000.

[LP85] O. Lichtenstein and A. Pnueli. Checking that finite state concurrent programs satisfy their linear specification. In *Proc. of POPL'85*, pages 97–107, 1985.

[MA02] K. L. McMillan and N. Amla. Automatic abstraction without counter examples. Technical report, Cadence Berkley Labs, Cadence Design Systems, 2002.

[McK81] B. D. McKay. Practical graph isomorphism. In *Congressus Numerantium*, volume 30, pages 45–87, 1981.

[McM93] K. L. McMillan. *Symbolic Model Checking: An Approach to the State Explosion Problem.* Kluwer Academic Publishers, 1993.

[MHB98] G. S. Manku, R. Hojati, and R. K. Brayton. Structural symmetry and model checking. In *Proc. of CAV'98*, pages 159–171, 1998.

[MMM95] J. Mohnke, P. Molitor, and S. Malik. Limits of using signature for permutation independent boolean comparison. In *Proc. of ASP-DAC'95*, pages 459–464, August 1995.

[MMZ+01] M. W. Moskewicz, C. F. Madigan, Y. Zhao, L. Zhang, and S. Malik. Chaff: Engineering an efficient SAT solver. In *Proc. of DAC'01*, pages 530–535, 2001.

[Moh99] J. Mohnke. *A Signature-based Approach for Formal Logic Verification.* PhD thesis, Martin-Luther-Universität Halle-Wittenberg, 1999.

[MR98] O. Möller and H. Rueß. Solving bit-vector equations. In *Proc. of FMCAD'98*, volume 1522, pages 36–48, 1998.

[MSS96] J. P. Marques-Silva and K. A. Sakallah. GRASP - A new search algorithm for satisfiability. In *Proc. of ICCAD'96*, pages 220–227, November 1996.

[MVF+02] W. McCune, R. Veroff, B. Fitelson, K. Harris, A. Feist, and L. Wos. Short single axioms for boolean algebra. *Journal of Automated Reasoning*, 29(1):1–16, 2002.

[NPW02] T. Nipkow, L. C. Paulson, and M. Wenzel. *Isabelle/HOL – A Proof Assistant for Higher-Order Logic*, volume 2283. Springer Verlag, 2002.

[PB97] M. Pandey and R. E. Bryant. Exploiting symmetry when verifying transitor-level circuits by symbolic trajectory evaluation. In *Proc. of CAV'97*, pages 244–255, 1997.

[PG86] D. Plaisted and S. Greenbaum. A structure-preserving clause form translation. *Journal of Symbolic Computation*, 2:293–304, 1986.

[PK00] V. Paruthi and A. Kuehlmann. Equivalence checking combining a structural SAT-solver, BDDs, and simulation. In *Proc. of ICCD'00*, pages 459–464, 2000.

[Plu98] D. Plump. *Handbook of Graph Grammars and Computing by Graph Transformation*, volume 2, chapter Term graph rewriting. World Scientific, 1998.

[Pre27] M. Presburger. Über die Vollständigkeit eines gewissen Systems der Arithmetik ganzer Zahlen, in welchem die Addition als einzige Operation hervortritt. In *Comptes Rendus du Premier Congrès des Mathématicienes des Pays Slaves*, pages 92–101, 395, Warsaw, 1927.

[PRSS99] A. Pnueli, Y. Rodeh, O. Shtrichman, and M. Siegel. Deciding equality formulas by small domains instantiations. In *Proc. of CAV'99*, pages 455–469, 1999.

[SBW98] C. Scholl, B. Becker, and T. M. Weis. Word-level decision diagrams, WLCDs and division. In *Proc of ICCAD'98*, pages 672–677, 1998.

[Sho84] R. E. Shostak. Deciding combinations of theories. *Journal of the ACM*, 31(1):1–12, 1984.

[SMMD99] C. Scholl, D. Möller, P. Molitor, and R. Drechsler. BDD minimization using symmetries. In *IEEE Transactions on CAD of ICs and Systems*, volume 18, pages 81–100, 1999.

[spe] http://www.verisity.com/products/specman.html.

[VB98] M. N. Velev and R. E. Bryant. Bit-level abstraction in the verification of pipelined microprocessors by correspondence checking. In *Proc. of FMCAD'98*, pages 18–35, 1998.

[VB01] M. N. Velev and R. E. Bryant. Effective use of boolean satisfiability procedures in the formal verification of superscalar and VLIW microprocessors. In *Proc. of DAC'01*, pages 226–231, 2001.

[ver95] *IEEE 1364 Verilog HDL standard*. IEEE, 1995.

[vhd93] *Standard VHDL Language Reference Manual, IEEE 1076 VHDL standard*. IEEE, 1993.

[Wol03] Wolfram Research, Inc. Eric Weisstein's world of mathemematics. http://mathworld.wolfram.com, 2003.

[ZKC01] Z. Zeng, P. Kalla, and M. Ciesielski. LPSAT: A unified approach to RTL satisfiability. In *Proc. of DATE'01*, pages 398–402, 2001.

[ZM03] L. Zhang and S. Malik. Validating SAT solvers using an independent resolution-based checker: Practical implementations and other applications. In *Proc. of DATE'03*, pages 880–885, 2003.

List of Figures

List of Tables

Lebenslauf

Persönliche Daten

Raik Brinkmann
geboren am 06.07.1972
in Brehna
ledig

Berufsweg

seit 2002 **Infineon Technologies AG, München, CVE-Entwicklungsgruppe**
Entwicklung von Basistechnologien für die Verifikation. Software-Architektur und -Release-Management.

1999 – 2002 **Siemens AG, Corporate Technology, München, Forschungsgruppe Formale Verifikation**
Doktorand, Entwicklung von Methoden zur Eigenschaftsbasierten Verifikation auf Wort-Ebene.
Eineinhalbjähriger Forschungsaufenthalt bei Infineon Technologies, San Jose, California, USA.

1996 – 1999 **Siemens AG, München, Bereich Information & Communication Networks**
Verifikations- und Entwicklungsingenieur für Embedded-Systems und System-on-Chip.

1991 – 1996 **Verschiedene Tätigkeiten während des Studiums**
Darunter vier Einsätze als Werkstudent bei der Siemens AG, Betreung von Praktikumsversuchen der Technischen Informatik an der TU-Clausthal, Freiberufliche Softwareentwicklung, Geschäftsführender Gesellschafter einer GbR.

Hochschulbildung

seit 1999 **Universität Kaiserslautern**
Doktorand des Fachbereiches Informatik.

1991 – 1996 **Technische Universität Clausthal**
Studium der Informatik mit Schwerpunkt Software-Engineering. Prädikat Sehr gut.
Thema der Diplomarbeit bei der Siemens AG: "Workflow-Management für das Hardware Design komplexer Logikbausteine".

Schulbildung

1989 – 1991 **Technische Universität Chemnitz**
Spezialklasse für Mathematik, Physik und Technik; Abitur.

1979 – 1989 **19. Polytechnische Oberschule, Halle/Saale**
Mittlere Reife.

Bibliographie

Raik Brinkmann und Rolf Drechsler. "RTL-Datapath Verification using Integer Linear Programming". Proceedings of ASP-DAC/VLSI Design 2002, January 07 - 11, 2002, Bangalore, India, pages 741-746, 2002.

Raik Brinkmann. "Using Symmetry for Problem Reduction in Bounded-Model-Checking on the Register-Transfer-Level". Proceedings of SymCon'01 – Symmetry in Constraint Satisfaction Problems, CP'01 Post-Conference Workshop, 2001.

Patente

"Elimination symmetrischer Variablen in HW-Verifikationsproblemen". Deutsche Patentanmeldung. Nr. 10146218.2, 2002.

"Schaltungsanordnung und Verfahren zur Hardware-Interruptbehandlung". Europäisches und US-Patent. Nr. 00102590.7-2201, 2000.

"Schaltungsanordnung und Verfahren zur Abarbeitung von Interruptanforderungen an einen Prozessor". Europäisches Patent. Nr. 00102719.2-2212, 2000.

Raik Brinkmann München, den 28. Juni 2003