Université de Montréal

# Model Checking for a First-order Temporal Logic Using Multiway Decision Graphs

par

Ying Xu

Département d'Informatique et de Recherche Opérationnelle

Faculté des Arts et des Sciences

Thèse présentée à la Faculté des études supérieures

en vue de l'obtention du grade de

Philosophiæ Doctor (Ph. D.)

en Informatique

avril, 1999



Université de Montréal

Faculté des études supérieures

Cette thèse intitulée:

# Model Checking for a First-order Temporal Logic Using Multiway Decision Graphs

présentée par:

Ying Xu

a été évaluée par un jury composé des personnes suivantes:

Guy Lapalme — président-rapporteur

Xiaoyu Song — directeur de recherche

Edward Cerny — co-directeur de recherche

Michel Boyer — membre du jury

Ganesh Gopalakrishnan — examinateur externe

Jean-Jules Brault — représentant du doyen

Thèse acceptée le: 99.06.22

# Résumé

L'exactitude lors de la conception est une préoccupation majeure pour tout système, et ceci est d'autant plus vrai pour les systèmes informatiques et les systèmes de circuit intégrés. Comme nous sommes toujours plus dépendants de ces systèmes, le coût de leur défaillance devient de moins en moins acceptable.

Traditionnellement, la simulation fut le principal mode de vérification de l'intégrité d'un système avant sa fabrication. Il existe différentes approches à la vérification d'un système à l'aide de simulations, chacune visant à explorer différents aspects de la conception du système. Cependant la méthodologie générale est la même: lorsque la phase de conception est terminée, l'équipe de vérification crée des bancs d'essais et utilise autant d'entrées que possible afin d'obtenir un degré de confiance suffisant dans la conception. Cependant, étant donné la complexité croissante des circuits intégrés, il devient rapidement impossible de simuler un large système de manière adéquate. C'est la raison pour laquelle il y a maintenant un renouveau d'intérêt de la recherche visant à utiliser la vérification formelle à titre de complément à la simulation pour vérifier l'intégrité de la conception d'un système.

La vérification formelle consiste à établir mathématiquement qu'une implantation d'un système satisfait sa spécification. Dans notre cas, l'implantation correspond à la description du système à vérifier. Quant à la spécification, elle décrit les propriétés que le système (c.à.d. l'implantation) doit satisfaire. L'implantation peut correspondre à différents niveaux d'abstraction du système, et la spécification (propriété) peut être exprimée de plusieurs manières: description du comportement, contraintes temporelles, formules de logique temporelle, etc.

La plupart des méthodes de vérification formelle peuvent être regroupées en 3 catégories: les démonstrateurs de théorèmes ("theorem proving"), la vérification d'équivalences, et la vérification du modèle ("model checking"). Les démonstrateurs de théorèmes constituent la technique de vérification formelle la plus générale: l'implantation et la spécification sont habituellement exprimées à l'aide de formules de la logique du premier ordre ou d'ordre supérieur. La relation entre l'implantation et la spécification est un théorème de la logique du démonstrateur qu'il faut prouver en utilisant les axiomes et les règles d'inférence logique. La puissance d'expression de la logique a un impact direct sur la puissance de cette méthodologie, ce qui permet de prendre en considération des systèmes avec chemins de données relativement complexes.

La vérification d'équivalences est une technique visant à vérifier l'équivalence de deux descriptions presque identiques d'un même système. La vérification d'équivalences est intéressante si on considère son utilisation suite à la synthèse du système, où souvent les changements manuels du système portent uniquement sur des aspects comme la rapidité, la puissance ou la testabilité.

La vérification des modèles est une technique visant à prouver des propriétés temporelles du système sous toutes les conditions possibles et permises. Des propriétés comme, "lorsqu'une requête pour le bus est émise, elle doit être satisfaite", peuvent être utilisées pour vérifier le comportement du système à vérifier. En gros, les vérificateurs de modèle ("model checkers") sont utilisés pour répondre à la question: "Est-ce que j'ai conçu ce qui était spécifié?"

Quoique les trois méthodes diffèrent sous certains aspects techniques, elles partagent les deux attributs suivants. Premièrement, aucun vecteur de test n'est requis. Ceci peut réduire la phase de vérification puisque le temps nécessaire pour la création des

vecteurs de test et pour l'évaluation des résultats est supprimé. Deuxièmement, pour les méthodes de vérification formelles, la preuve est mathématique plutôt qu'expérimentale. Comme l'exactitude des théorèmes est vraie indépendamment des valeurs pour lesquelles le théorème s'applique, l'exactitude de la conception d'un système qui a été formellement vérifié est vraie avec une certitude mathématique indépendamment des entrées soumisses au système. Cependant, l'expertise de haut niveau requise dans l'utilisation des démonstrateurs de théorèmes fait que les techniques basées sur ces démonstrateurs de théorèmes ne sont pas largement utilisées dans l'industrie. Pour cette raison, les vérificateurs de modèles (qu'on peut exécuter de manière complètement automatique) sont considérés présentement comme la technique la plus prometteuse qui puisse être utilisée pour vérifier les propriétés de systèmes complexes [5][18][20][66][55].

Durant les dernières décades, les chercheurs ont concentré leurs efforts à explorer les techniques de vérification de modèles. Ces techniques ont d'abord été introduites par Clarke et Emerson[22], et de manière indépendante par Quielle et Sifakis [61]. Les premières techniques de vérification de modèles étaient basées sur des graphes de décisions représentant de manière explicite l'espace d'états du système en utilisant une liste ou un tableau dont la taille est proportionnelle au nombre d'états dans le système [7]. Le nombre d'états dans le modèle peut croître de manière exponentielle en fonction du nombre de composantes dans la conception, ce qui fait qu'habituellement le tableau d'états est un facteur limitant l'application de ces graphes de décisions à des systèmes réels (à cause du problème de croissance exponentielle du nombre d'états).

Utilisant les graphes de décisions binaires ordonnés (Ordered Binary Decision Diagrams (OBDDs)) [11] pour représenter les ensembles d'états, les relations de transitions, et pour énumérer implicitement l'espace d'états, la vérification symbolique s'est avérée être une technique pratique pour la vérification automatique des circuits intégrés au niveau de la logique propositionnelle [14][16][24][31][49][65]. Cependant, ces méthodes

nécessitent que la description du système soit faite au niveau de la logique booléenne. Par conséquent, elles ne sont pas en général adéquates pour la vérification de circuits avec de larges chemins de données, encore une fois dû au problème d'explosion du nombre d'états dans le modèle. Le nombre d'états dans le modèle croît de manière exponentielle en fonction du nombre de variables d'états, même avec la technique des OBDDs, les structures de données deviennent trop grandes pour la taille des mémoires des ordinateurs actuels.

Étant motivé par le désir de combiner l'aspect automatique de la vérification des modèles avec la représentation abstraite des données des démonstrateurs de théorèmes, ce qui peut minimiser le problème d'explosion du nombre d'états de manière significative, nous avons développé un vérificateur de modèles pour la logique temporelle linéaire du premier ordre. Notre approche est basée sur le modèle de machine à états abstraits (ASMs) où une donnée peut être représentée par une seule variable de type abstrait, plutôt qu'un vecteur de variables booléennes. Une opération sur les données est alors représentée par un symbole fonctionnel non-interprété [26][27][68][2].

Plusieurs résultats rapportés dans la littérature sont reliés aux nôtres. Hungar, Grumberg et Damm [41] ont proposé une technique appelée "true symbolic model checking". Ils ont représenté les données et les opérations sur les données à l'aide de formules logiques du premier ordre et ils ont utilisé la logique FO-ACTL (First-Order ACTL) pour spécifier les propriétés. La logique FO-ACTL est une logique temporelle du premier ordre à branchements qui utilise uniquement le quantificateur universel. Ils ont nommé leur méthode "réellement symbolique" (truly symbolic) en contraste avec l'approche par encodage d'un ensemble d'états de la vérification symbolique de modèles (symbolic model checking) présentée dans [49]. Leur méthode est basée sur l'hypothèse que toutes les boucles de données se terminent, et sur la séparation entre la partie contrôle et la partie données dans les circuits intégrés typiques. Si les propriétés contiennent uniquement des signaux de contrôle, alors la méthode classique de vérification booléenne

de modèles est utilisée. Lorsque la propriété contient uniquement des données, ils éliminent d'abord tous les prédicats du premier-ordre de la formule en les remplaçant par la constante booléenne "True", ce qui résulte en une formule propositionnelle CTL. Si la formule propositionnelle CTL n'est pas vérifiée par un vérificateur booléen de modèles, alors ils concluent que la propriété originale a échouée. Sinon (la formule propositionnelle CTL est vérifiée), ils génèrent à partir du système et de la propriété en question, des conditions de vérification du premier ordre en utilisant la méthode des tableaux. Ensuite, ils les vérifient à l'aide d'un démonstrateur de théorèmes, ce qui complète la preuve.

Cyrluk et Narendran [33] ont défini une logique temporelle du premier-ordre Ground Temporal Logic (GTL), qu'on peut situer entre la logique temporelle propositionnelle et la logique du premier-ordre. Cyrluk et Narendran ont montré que la logique GTL est indécidable. Ensuite ils ont identifié un sous-ensemble décidable de GTL, ce fragment contient des formules de la forme $\Box p$ (toujours $p$), où $p$ est une formule GTL contenant un nombre arbitraire d'opérateurs (next) o, et aucun autre opérateur temporel. Cependant, ils n'ont pas montré comment construire la procédure de décision.

Hojati, Brayton et al. ont proposé un modèle concurrent appelé "integer combinational/sequence" (ICS), qui utilise des relations finies, des fonctions et des prédicats sur les nombres entiers qui sont interprétés et/ou non-interprétés, et des fonctions de mémoire interprétées, pour décrire les circuits intégrés avec abstraction des chemins de données [51][52]. La vérification des modèles ICS est accomplie en utilisant l'inclusion des langages formels ("language containment"). Ils ont montré que pour une classe de modèles à contrôle intensif (ICS), les variables entières du modèle peuvent être remplacées par une énumération finie, ce qui permet d'exécuter la vérification au niveau booléen sans en sacrifier l'exactitude. Ils ont donné un algorithme avec complexité linéaire pour reconnaître les formules appartenant à cette classe. Si les circuits contiennent des chemins de données complexes avec des symboles fonctionnels interprétés et non-interprétés, l'énumération

finie ne peut pas être utilisée. À la place, ils calculent les ensembles d'états atteignables en $n$ étapes en utilisant les BDDs, et vérifient qu'aucune erreur n'existe pour ces $n$ étapes.

Burch et Dill ont aussi utilisé la logique du premier-ordre sans quantificateur avec prédicat d'égalité pour vérifier la partie contrôle des microprocesseurs [18]. Ceci est dû au fait que la partie donnée peut être abstraite dans leur logique. Leur méthode inclut deux phases. La première phase compile la spécification et l'implantation en une formule logique; la formule est valide si et seulement si l'implantation satisfait la spécification. La seconde phase consiste à montrer la validité de la formule par le biais d'une procédure de décision qu'ils ont implantée. Ils ont appliqué leur méthode pour la vérification d'une implantation "pipeline" d'un sous-ensemble de l'architecture DLX. Cependant, leur méthode, contrairement à la nôtre, ne peut pas prendre en compte les propriétés impliquant des opérateurs temporels, en particulier, les propriétés de "liveness".

Dans cette thèse, nous présentons une logique temporelle branchements du premier ordre et les procédures de décision qui lui sont associées. En comparaison avec d'autres recherches, nous élevons le niveau d'abstraction de l'explicitation du problème, et nous explorons la vérification de modèles à un niveau d'abstraction plus élevé. Notre approche est basée sur les machines à états abstraits (ASMs). Une machine abstraite ASM est représentée par des graphes de décisions multi-choix (MDGs) [26][70]. Ces derniers sont une généralisation des graphes de décisions binaires ordonnés et réduits (ROBDDs) [11]. Les machines abstraites (ASMs) peuvent être utilisées pour décrire les systèmes au niveau transfert de registres (RTL). La vérification par ASMs est basée sur l'énumération des états dont la complexité est indépendante des chemins de données. Une implantation de ce concept existe actuellement et fournit des outils comprenant la vérification d'équivalences pour les circuits séquentiels, la vérification des processeurs, la vérification des invariants par exploration des états [69] et la vérification des propriétés temporelles.

Notre objectif principal dans cette thèse est de définir une logique temporelle du premier-ordre avec branchements appelée Abstract-CTL* et de développer des algorithmes de vérification des propriétés pour un sous-ensemble de Abstract-CTL*. Ce sous-ensemble est appelé $L_{MDG}$. $L_{MDG}$ inclut les propriétés de "safety" et "liveness" avec ou sans contraintes d'équité. Les propriétés de $L_{MDG}$ incluent **A***p*, **AG***p*, **AF***p*, **A***p***U***q*, **AG**(*c* =>(**F***p*)) et **AG**(*c*=>(*p***U***q*)), où *c*, *p* et *q* sont Next_let_formulas qui contiennent seulement l'opérateur temporel **X**("Next"), **G**, **F**, **U** signifiant respectivement "always", "eventually", "until", et **A** signifie "for all computation paths". En général, notre approche consiste à compiler la propriété à vérifier en un (des) ASM(s), puis à vérifier une propriété plus simple qui en découle, sur la machine produite par la combinaison de la machine représentant le modèle et la propriété.

En comparaison avec le travail de [33], nous allons montrer au cours de notre thèse que le fragment décidable de GTL est actuellement un sous-ensemble de la classe des propriétés que nous pouvons vérifier; en comparaison avec ICS [51][52], nos modèles ASM sont plus généraux que les modèles ICS dans le sens que le type de variables abstraites de notre système (qui correspondent aux variables entières dans les modèles ICS) peuvent recevoir n'importe quelles valeurs dans leur domaine, plutôt qu'une constante spécifique ou une fonction des constantes comme dans le modèle ICS. Pour la classe de modèles ICS où l'énumération finie ne peut pas être utilisée, notre système de vérification peut calculer tous les états atteignables et vérifier les propriétés de "safety" et aussi certaines propriétés de "liveness". En comparaison avec [41], notre logique temporelle du premier-ordre avec temps de branchements $L_{MDG}$ est moins expressive que FO-ACTL, puisque nous allouons seulement un niveau limité d'imbrication des opérateurs temporels. Cependant, dans notre approche, la propriété est vérifiée dans tout le modèle de manière automatique tandis que dans [41], un démonstrateur de théorèmes est éventuellement nécessaire pour valider les conditions de vérification générées.

La thèse est organisée de la manière suivante: dans le chapitre 2, nous explorons les techniques de base de la vérification formelle et les logiques utilisées dans ces techniques. Dans le chapitre 3, nous énonçons les fondements théoriques de cette thèse. Premièrement nous décrivons la logique formelle utilisée dans notre approche ASM. Deuxièmement, nous définissons notre modèle de calcul, c.à.d., la définition des machines à états abstraits et donnerons leur sémantique en termes d'arbre infini de calcul. Nous expliquons également comment l'énumération abstraite des états est accomplie. Dans le chapitre 4, nous définissons la syntaxe et la sémantique d'un modèle très général de logique temporelle appelé Abstract_CTL*, qui est une logique du premier-ordre avec branchements. Dans le chapitre 5, nous définissons $L_{MDG}$, qui est un sous-ensemble de Abstract_CTL* contenant la classe des propriétés pour lesquelles une procédure de décision existe. Dans le chapitre 6, nous présentons en détail les algorithmes de vérification de propriétés. Dans le chapitre 7, nous présentons un algorithme pour générer une description de circuit représentant une Next_let_formula. Dans le chapitre 8, nous montrons comment imposer des contraintes d'équité dans notre système de vérification et les algorithmes de vérification des propriétés de liveness avec des contraintes d'équité. Nous discutons aussi de certaines questions d'implantation. Dans le chapitre 9, nous démontrons l'exactitude de notre procédure de vérification. Dans le chapitre 10, nous appliquons le vérificateur de modèle basé sur MDG à deux exemples: un contrôleur de l'éclairage dans un tunnel, et un compteur abstrait. Dans le chapitre 11 nous concluons la thèse et nous indiquons des directions futures de recherche.

# Abstract

Using Ordered Binary Decision Diagrams (OBDD) to encode sets of states, the transition relations, and to perform an implicit enumeration of the state space, symbolic model checking has proven to be a very practical technique for the automatic verification of hardware designs at the propositional logic level. However, these methods still require the description of the design to be at the Boolean logic level, and thus in general they are not adequate for verifying circuits with large datapath again because of the state explosion problem. That is, the number of states in the model grows exponentially with the number of state variables and therefore, even with OBDD encoding the data structures become too large to fit typical current computer memories.

In this thesis, we study the automatic model checking with a first-order branching time temporal logic. Compared to other researches, we raise the level of abstraction at which the problem is stated and explore model checking at a higher abstraction level. Our approach is based on abstract descriptions of state machines (ASMs). An ASM is encoded using Multiway Decision Graphs (MDGs), of which Reduced Ordered Binary Decision Diagrams (ROBDDs) are a special case. ASMs can be used to describe designs at Register Transfer Level (RTL). The verification of ASMs is based on state enumeration whose complexity is independent of the width of the datapath.

The main task of the thesis is to define a first-order branching time temporal logic called Abstract-CTL* and develop property checking algorithms for a subset of Abstract-CTL* called $L_{\mathrm{MDG}}$, which includes safety properties and liveness properties with or without fairness constraints. The main property templates in $L_{\mathrm{MDG}}$ include $\boldsymbol{A}p$, $\boldsymbol{AG}p$, $\boldsymbol{AF}p$, $p\boldsymbol{U}q$, $\boldsymbol{AG}(c \Rightarrow (\boldsymbol{F}\ p))$ and $\boldsymbol{AG}(c \Rightarrow p\boldsymbol{U}q)$, where $c$, $p$ and $q$ are Next_let_formulas which contain only the temporal operator $\boldsymbol{X}$ ("Next"), $\boldsymbol{G}$, $\boldsymbol{F}$, $\boldsymbol{U}$ means "always", "eventually",

"until" respectively, and ***A*** means "for all computation paths". In general, our approach to model checking is to automatically build additional ASMs that represent the Next_let_formulas appearing in a property, connect these additional ASMs to the original one to be verified, and then check a simpler property on the composite machine.

**Key words:** formal verification, model checking, temporal logic, abstract descriptions of state machines, property, fairness constraint

# Acknowledgments

I could not have completed this thesis without the assistance of many people. First and foremost, I would like to thank my supervisors Prof. Xiaoyu Song and Prof. Eduard Cerny for their constructive technical advice, financial support and constant encouragement throughout my doctoral studies. I wish to especially thank Prof. Eduard Cerny for his many suggestions, comments and corrections in the writing of the thesis.

Many thanks also to Dr. Francisco Corella for his valuable discussions during my research work.

I have enjoyed studying and working with my friends and fellow graduate students in the Dept. IRO. Although the names are far too numerous to list here, I would like to thank everyone for making my years here enjoyable.

My special thanks to Dr. Zijian Zhou and his wife Haiyun Ma for their precious friendship and for always being there when I need their help.

Last but not least, I would like to thank my husband, my two sisters and the rest of my family for their constant moral support and encouragement, which were invaluable in completing this thesis.

# Table of Contents

# List of Tables

# List of Figures

# 1 Introduction

Correctness is a major consideration in the design of any system, and in particular, those of our concern - computers and other digital systems. As we become more and more dependent on such systems, the cost of a failure is becoming unacceptably high. Traditionally, simulation has been the only means of verifying the integrity of a design prior to manufacturing. There are different types of simulation to explore different aspects of the design, but the basic methodology is always the same: once a design had been developed, the verification team would create test benches and run as many vectors as needed to achieve sufficiently high confidence in the design. However, because of the increasing complexity of digital systems, it is rapidly becoming impossible to simulate large designs adequately. For this reason, there has been a surge of research interest in *formal verification* which could be deployed as a complement to simulation for determining the correctness of a design.

In general, the formal verification problem consists of mathematically establishing that an implementation satisfies a specification. The implementation refers to the system design that is to be verified. This entity can correspond to a design description at any level of the system abstraction hierarchy, not just the final physical layout. The specification refers to the property with respect to which correctness is to be determined. It can be

expressed in a variety of ways, such as behavioral description, an abstract structural description, a timing requirement, a temporal logic formula, etc.

Most formal verification methods can be classified into 3 classes: theorem proving, equivalence checking, and model checking. Theorem proving is the most general verification technique: an implementation and its specification are usually expressed as first-order or high-order logic formulas. Their relationship, equivalence or implication, is regarded as a theorem to be proven within the logic system using axioms and inference rules. Design can thus be represented at different logic levels rather than only at the Boolean level. Therefore, it allows a hierarchical verification methodology which can effectively deal with the overall functionality of designs having complex datapaths.

Equivalence checking is a technique to check the equivalence of two designs. In order for an equivalence checker to work, the designs must be "almost identical" and thus equivalence checking is most valuable in post synthesis design verification where often manual design changes focus on speed, power or testability considerations. These post synthesis iterations make no changes or only small changes to the behavior of the design. Equivalence checkers answer the question "Did my design iteration introduce any new errors into the design?", as shown in Figure 1.

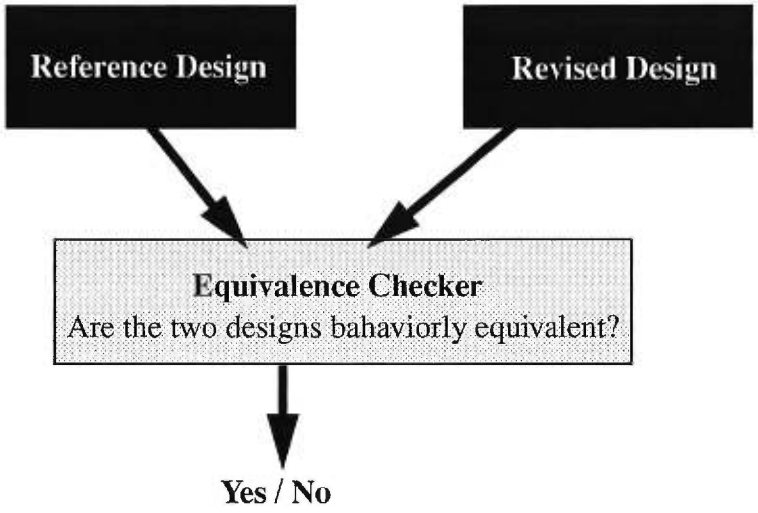



**Figure 1 -** Equivalence checking

Model Checking is a technique to prove temporal properties on a design model under all possible and allowable conditions. Properties like "after the bus is requested, then it must be granted" can be used to verify the behavior of the design in question. Roughly speaking, model checkers are used to answer the question "Did I design what was intended or specified?" as shown in Figure 2.

Broadly speaking, equivalence checking can be viewed as a special model checking technique in which the property to be proven is the equivalence of the two designs.

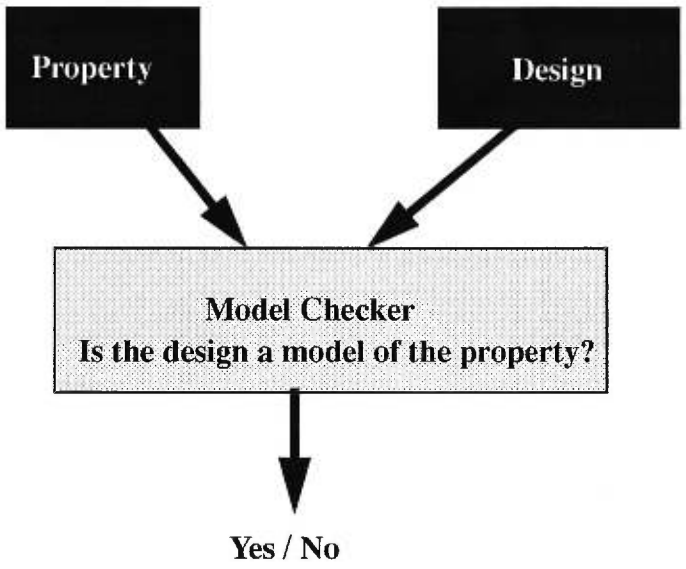



**Figure 2 -** Model checker

Although the three methods differ in technical aspects, they share the following two common attributes:

First, no test vectors are required. This can shorten the verification phase to some extent that the large amount of design time needed to create test vectors and to evaluate the results of the simulations will be saved.

Second, with formal verification methods, the proof is mathematical rather than experimental. Just as the correctness of a mathematically proven theorem holds regardless of the particular values that it is applied to, correctness of a formally verified system design holds with mathematical certainty regardless of its input values.

Due to the needed expertise in the use of theorem provers, theorem proving techniques are not widely accepted in industrial use. Instead, since model checking can be carried out fully automatically, it is seen today as the most promising technique that could be used to verify properties regarding complex designs [5] [18][21][77][51].

Over the last decade, researchers have put much effort to explore model checking techniques. Model checking was first introduced by Clarke and Emerson[23] and independently by Quielle and Sifakis [69]. The early model checking methods relied on decision algorithms that explicitly represent state space, using a list or table that grows in proportion to the number of states [7]. Because the number of states in the model may grow exponentially with the number of components in the design, the size of the state table is usually the limiting factor in applying these algorithms to realistic systems (the so -called *state explosion* problem).

Using Ordered Binary Decision Diagrams (OBDDs) [11] to encode sets of states, the transition relations, and to perform an implicit enumeration of the state space, symbolic model checking has proven to be a very practical technique for the automatic verification of hardware designs [14][16][25][32][61][76]. However, these methods still require the description of the design to be at the Boolean logic level, and thus in general they are not adequate for verifying circuits with large datapath again because of the state explosion problem. That is, the number of states in the model grows exponentially with the number of state variables and therefore, even with OBDD encoding the data structures become too large to fit typical current computer memories.

Being motivated by a desire to combine the automation feature of model checking

and the abstract representation of data in theorem proving which can significantly alleviate the state explosion problem, we developed model checking for a first-order branching time temporal logic. Our approach is based on a computation model called an *abstract description of state machines* (ASMs) where a data value can be represented by a single variable of abstract type, rather than by a vector of Boolean variables, and a data operation is represented by an uninterpreted function symbol[2][27][28][79].

## 1.1 Related work

To our knowledge, three previous developments reported in the literature are directly related to ours.

Hungar, Grumberg and Damm [47] proposed a "true symbolic model checking" technique. They represent data and data operations by first-order formulas and used FO-ACTL (First-Order ACTL), a first-order branching time temporal logic with only the universal path quantifier to specify properties. They called their method "truly symbolic" in contrast to the state set coding approach to symbolic model checking presented in [61]. Their method is based on the assumption that all data loops terminate, and on the separation of the control part and the data path in typical circuits. If the property only contains control signals, then Boolean model checking is applied. When the property contains data, they first eliminate all first-order predicates in the property formula to result in a propositional Computation Tree Logic (CTL) formula. This is achieved by replacing those predicates with the Boolean constant *true*. If the propositional CTL formula is not verified by a Boolean model checker, then they conclude that the original property fails. Otherwise (the propositional CTL formula is verified), the tableau method is used to generate a pure first-order verification condition from the system and the property to be proven. Then they complete the verification of the property by proving the verification condition using a theorem prover.

Cyrluk and Narendran [34] defined a first-order temporal logic - Ground Temporal Logic (GTL), which falls in between the first-order and the propositional temporal logics. Given a first-order language (FOL) consisting of function symbols, constants (0-ary function symbols), predicate symbols, equality, but no global variables, they define the alphabet of GTL as the alphabet of FOL along with the temporal operators ○(Next), o (Next, only applied to terms), ❑ (always), and a set of state variables *Vs* whose values can change over time. The terms of GTL are defined inductively: every state variable is a term; if $f$ is an n-ary function symbol and $t_1, ..., t_n$ are terms then $f(t_1, ..., t_n)$ is a term; if $t$ is a term then so is o$t$. An n-ary predicate $p(t_1, ..., t_n)$ is an atomic formula of GTL. Formulas of GTL are defined inductively: every atomic formula is a formula; if $A$ and $B$ are formulas, then so are $\neg A$, $A \wedge B$, $\bigcirc A$, and $\Box A$. A model $K = (S, W)$ for GTL consists of a model for the first-order language interpretation $S$ and an infinite sequence of states $W$, which is a computation path in CTL jargon. A formula $A$ of GTL is valid if and only if $A$ is true for every model $K$; and $A$ is satisfiable if and only if $A$ is true in some model $K$. In [34], Cyrluk and Narendran showed that the full GTL is undecidable. They then identified a decidable fragment of GTL, consisting of formulas in the form of $\Box p$ formulas where $p$ is a GTL formula containing arbitrary number of the "Next" temporal operators but no other temporal operators. For this decidable fragment, they said that it was possible to build an automatic validity checker. However, they did not show how to build the decision procedure.

Hojati, Brayton et al. proposed a concurrency model called integer combinational/ sequential (ICS), which uses finite relations, interpreted and uninterpreted integer functions and predicates, and interpreted memory functions to describe hardware systems with datapath abstraction [43][44][45][50]. Verification of ICS models is performed using language containment. They showed that for a subclass of "control-intensive" ICS models, integer variables in the model can be replaced by enumerated variables (i.e., finite instantiation) and then the property verification can be carried out at the Boolean logic level without sacrificing accuracy. They gave a linear time algorithm for recognizing those

subsets. For verifying properties of circuits with complex datapaths, i.e., the circuit contains interpreted and uninterpreted functions, finite instantiation cannot be used. Instead, they compute the set of states reachable in $n$ steps using BDDs, and check that no error exists in these $n$ steps.

Burch and Dill also used a subset of first-order logic, specifically, the quantifier-free logic of uninterpreted functions and predicates with equality and propositional connectives, for verifying microprocessor control circuitry [18]. Their logic is appropriate for verification of microprocessor control because it allows abstraction of datapath values and operations. Their method includes two phases. The first phase compiles a behavioral description of the specification and the implementation into a formula in the logic; the formula is valid if and only if the implementation is correct with respect to the specification. The second phase is a decision procedure that checks whether the formula is valid. They applied their method in the verification of a pipelined implementation of a subset of the DLX architecture. However, their method, unlike ours, cannot verify properties involving temporal operators, in particular, liveness properties.

## 1.2 Scope of the thesis

In this thesis, we study the automatic model checking with a first-order branching time temporal logic. Compared to other researches, we raise the level of abstraction at which the problem is stated and explore model checking at a higher abstraction level. Our approach is based on *abstract descriptions of state machines* (ASMs) where a data value is represented by a single variable of abstract type, rather than by Boolean variables, and a data operation is represented by an uninterpreted or partially interpreted function symbol. An ASM is encoded using Multiway Decision Graphs (MDGs) [27][81] of which Reduced Ordered Binary Decision Diagrams (ROBDDs) [11] are a special case. ASMs can be used to describe designs at Register Transfer Level (RTL). The verification of

ASMs is based on state enumeration whose complexity is independent of the width of the datapath. Thus, the state explosion problem caused by descriptions of large datapaths at the Boolean logic level is avoided.

The current ASM-based package provides tools for the verification of behavioral equivalence of sequential circuits, the verification of a microprocessor against its instruction set architecture, invariant checking using reachability analysis [80], and temporal property checking. The main objective of the thesis is to define a first-order branching time temporal logic called Abstract-CTL* and to develop property checking algorithms for a subset of Abstract-CTL* called $L_{\mathrm{MDG}}$. This includes safety properties and liveness properties with or without fairness constraints.

Compared to the work of [34], we shall see in the following chapters that the decidable fragment of GTL is actually a subset of the class of properties that we can verify; Compared to ICS [43][45], our ASM models are more general in the sense that the abstract sort variables in our system (corresponding to the integer variables in ICS models) can be assigned any value in their domains, rather a particular constant or function of constants as in the ICS model. For the class of ICS models where finite instantiations cannot be used, our verification system can still compute all the reachable states and check safety properties as well as certain liveness properties. For example, the abstract counter presented in Chapter 10 cannot be handled by the ICS model, but it can be described using the ASM model. Compared to [47], our first-order linear-time temporal logic $L_{\mathrm{MDG}}$ is less expressive than FO-ACTL, since we only allow limited nesting of temporal operators. However, in our approach the property is checked in the whole model automatically, while in [47] a theorem prover is eventually needed to validate the pure first-order verification condition.

## 1.3 Contributions

The results reported in this thesis were obtained through the collaboration with E. Cerny, X.Song and F. Corella. I made partial important contributions in item 1 and major original contributions in 2-5.

1. The definition of a first-order branching time temporal logic Abstract_CTL*, which can be used to specify temporal properties for a system described using ASM computation model;

2. The definition of $L_{\mathrm{MDG}}$ (a subset of Abstract_CTL* ) and the development and implementation of property checking algorithms for $L_{\mathrm{MDG}}$, including the algorithms for liveness property checking under fairness constraints;

3. The development and implementation of the ASM construction technique which efficiently builds an ASM for a property involving only the temporal operator "Next";

4. Experimentation on some benchmarks.

## 1.4 Outline

This thesis is organized as follows:

In Chapter 2, we review the basic formal verification techniques and the logics used in those techniques.

In Chapter 3, we give the theoretical foundations of this thesis. First, we describe the formal logic used in our ASM approach. Second, we define the computation model,

i.e., the definition of an abstract description of state machines and the definition of an abstract infinite computation tree. We also explain how abstract state enumeration proceeds.

In Chapter 4, we define the syntax and the semantics of a very general temporal logic called *Abstract_CTL**, which is a first-order branching time temporal logic.

In Chapter 5, we define $L_{MDG}$, a subset of Abstract_CTL*, as a property specification language for which we have been able to develop property checking procedures.

In Chapter 6, we present in detail the property checking procedures.

In Chapter 7, we give an algorithm for generating a circuit description representing a Next_let_formula, which is a formula including only the temporal operator "Next".

In Chapter 8, we show how to impose fairness constraints in our verification system and the algorithms for checking liveness properties under fairness constraints. We also discuss implementation issues.

In Chapter 9, we demonstrate the soundness of our verification procedure.

In Chapter 10, we verify some properties regarding the Island Tunnel Control bench mark using our model checker and also using VIS from University of California at Berkley. We also verify several properties regarding an abstract counter in which the value of the counter is described using a variable of abstract type.

In Chapter 11, we conclude the thesis and outline future directions of research.

# 2 Formal Verification Techniques

Since the idea of using formal methods for verifying systems was first introduced, numerous approaches to this problem have been explored by researchers. These approaches can be classified into 3 main categories:

- interactive, predicate logic based *theorem proving* techniques
- finite state automata based *equivalence checking* techniques
- propositional logic based *model checking* techniques

In this chapter, we shall review the basic ideas behind these techniques.

## 2.1 Theorem proving

One of the earliest approaches to formal hardware verification was to describe the implementation as well as the specification in a formal logic. The correctness result was then obtained within the logic, by proving that the specification and the implementation were suitably related (logical equivalence or logical implication). The underlying concept of this method is the notion of *formal theory.* A formal theory $S$ is defined by:

1. A finite alphabet. The symbols of this alphabet are the symbols of the theory. A finite sequence of these symbols is called an *expression* of *S*.

2. A subset of the expressions of *S* are the *well-formed formulas* of *S*.

3. A finite set of the well-formed formulas of *S* are *axioms* of *S*.

4. A finite set of rules of inference. A rule of inference allows the derivation of a new well-formed formula from a given finite set of well-formed formulas.

A *formal proof* in *S* is a finite sequence of well-formed formulas: $f_1, f_2, \ldots, f_n$, such that for every $i$, formula $f_i$ is either an axiom or can be derived by one of the rules of inference from the formulas $\{f_1, f_2, \ldots, f_{i-1}\}$. Traditionally, the last well-formed formula in a formal proof is called a *theorem* of *S*, and the formal proof is a proof of this theorem.

The formal logics that are usually employed in theorem proving can be classified as first-order predicate logic and high-order predicate logic.

### 2.1.1 First-order predicate logic

First-order predicate logic is one of the most extensively studied logics. Its language alphabet consists of a signature (countable sets of symbols for constants, functions, and predicates), symbols for variables, a set of standard Boolean connectives ($\neg$, $\vee$, $\wedge$, $\Rightarrow$, $\equiv$) and quantifiers ($\exists$, $\forall$). There are two main syntactic categories — *terms* and *formulas*. Terms consist of constants, variables, and function applications to argument terms. Formulas consist of atomic formulas (predicates), Boolean combinations of component formulas and quantified formulas (with quantification allowed on variables only). An interpretation for a first-order logic consists of a structure (a domain of discourse and appropriate mapping of the signature symbols) and an assignment for the variables

(mapped to domain elements). Semantically, terms denote elements in the domain, and formulas are interpreted as true/false. Different first-order languages are developed depending on the exact set of signature symbols used and their interpretations. Various proof systems have been studied for first-order logics. A representative one is the Boyer-Moore computational logic [8].

Boyer-Moore computational logic is a restricted form of first-order logic which was developed for the explicit purpose of reasoning about computations. Here we just give the brief introduction to this logic. A detailed description can be found in [8].

Boyer-Moore logic is a quantifier-free first-order logic with equality. Its syntax, which uses a prefix notation, resembles that of Lisp. The logic is mechanized by a collection of Lisp programs that permit the user to axiomatize inductively constructed data types, define recursive functions, and prove theorems about them. This collection of programs is frequently referred to as "the Boyer-Moore theorem prover". The main principles of the Boyer-Moore system are:

- The *shell principle*, which is used to define inductive abstract data types by means of a bottom object, a constructor and one or more accessors. A Boolean function, called a recognizer, checks whether an object belongs to the shell.

- The *definition principle*, which ensures that all new functions are defined either non-recursively in terms of pre-defined functions, or in the case of recursive definitions, a well-founded ordering exists on some measure of the arguments that decreases with each recursive call.

- The *induction principle*, on which the induction heuristics of the proof mechanism are based. An induction scheme is automatically generated according to the definition of the recursive functions involved in the theorem to be proved.

The Boyer-Moore system provides an automated facility for generating proofs in

the logic. However, the process of proof generation is not fully automatic, in that the theorem prover may need assistance from the user for setting up intermediate lemmas and helpful definitions. The strong mathematical foundation and heuristics that have been built into the system make it an effective tool used in a number of application areas.

An important approach to hardware verification within Boyer-Moore logic was made by W.A. Hunt, Jr. He demonstrated the use of the theorem prover for the verification of the FM8501, a microprogrammed 16-bit microprocessor similar in complexity to a PDP-11 [48]. The specification presents a programmer's view of FM8501 in the form of an interpretation function at the macro-instruction level. The implementation consists of its description as a hardware interpreter that operates at the micro-instruction level. Recursive function definitions within the logic are used to represent the various combinational and sequential hardware units. Verification is performed by proving a theorem that states the equivalence of the two descriptions, under an appropriate assumption on the initial conditions. Later, he extended his work to the verification of the FM8502, a 32-bit microprocessor with a richer instruction set [49]. Hunt demonstrated the effectiveness of using a mechanized (though not fully automated) theorem proving facility and also made a good use of the recursion and induction principles allowed by the Boyer-Moore logic to reason about hardware functions with arbitrarily-sized arguments.

### 2.1.2 Higher-order logic

Higher-order logic is a version of predicate calculus with three main extensions [41]:

- Variables can range over functions and predicates (hence "higher order").

- The logic is *typed* (each theory specifies a signature of type and individual constants).

•There is no separate syntactic category of formulas (formulas are identified with terms of type *bool*).

High-order logic allows quantification over arbitrary predicates and functions. This ability leads to a greater expressive power, but also the increased complexity of analysis compared to the first-order case. The incompleteness of a sound proof system for most high-order logics makes logical reasoning more difficult than in the first-order case, and one has to rely on ingenious inference rules and heuristics. In spite of these difficulties, the use of high-order logics in formal verification has become increasingly popular in the past few years. An important consideration in most cases is the use of some controlled form of logic and inferencing, in order to minimize the risk of inconsistencies, while maintaining the benefits of powerful representation and inference mechanisms.

Among the higher-order logic theorem proving systems HOL is the most typical one [41][33]. The HOL system which includes a HOL logic and a theorem proving system was developed by the Hardware Verification Group at the University of Cambridge, England. This system is based on a version of high-order logic developed by Gordon for the purpose of hardware specification and verification.

Syntactically, HOL uses the standard predicate logic notation with the same symbols for negation, conjunction, disjunction, implication, quantification, etc. There are four kinds of terms — constants, variables, function applications, and lambda-terms that denote functional abstractions. Semantically, types denote sets and terms denote members of these sets. Formulas, sequents, axioms, and theorems are represented by using terms of Boolean type.

The sets of types, type operators, constants, and axioms available in HOL are organized in the form of theories. There are two built-in primitive theories, *bool* and *ind*, for Booleans and individuals respectively. Other important theories, which are arranged in a hierarchy, have been added to axiomatize lists, products, sums, numbers, primitive

recursion, and arithmetic. On top of these, users are allowed to introduce application-dependent theories by adding relevant types, constants, axioms, and definitions. New types are introduced by specifying an existing representing type, a predicate that identifies the subset isomorphic to the new type and by proving appropriate theorems about them.

The HOL logic is embedded in an interactive functional programming language called ML. The overall HOL system supports a natural deduction style of proof, with derived rules formed from eight primitive inference rules, e.g., a collection of rewrite rules. All inference rules are implemented by using ML functions, and their application is the only way to obtain theorems in the system. Once proven, theorems can be saved in the appropriate theories to be used for future proofs. Most proofs done in the HOL system are goal-directed and are generated with the help of tactics and tacticals. A *tactic* is an ML function that is applied to a goal to reduce it to its subgoals, while a *tactical* is a functional that combines tactics to form new tactics. The tactics and tacticals in HOL are derived from the Cambridge LCF (logic for computable functions) system (which evolved from the Edinburgh LCF). The strict type discipline of ML ensures that no ill-formed proofs are accepted by the system.

Verification tasks in the HOL system can be set up in a number of different ways. The most common one is to prove that an implementation, described structurally, implies or is equivalent to, a behavioral specification. For example, a behavioral description for an Exor-gate can be represented as a predicate [42]:

$$\text{Exor\_Spec}(a, b, c) \equiv ( c = \neg(a=b) ),$$

and its structural implementation in terms of simpler Boolean gates (shown in Figure 3) can be represented as

$$\text{Exor\_Imp}(a,b,c) \equiv \exists p,q.\ \text{Nand}\ (a, b, p) \wedge \text{Or}\ (a, b, q) \wedge \text{And}\ (p, q, c).$$

The correctness theorem can be expressed as Exor_Imp(a,b,c) ≡ Exor_Spec(a, b,

c).

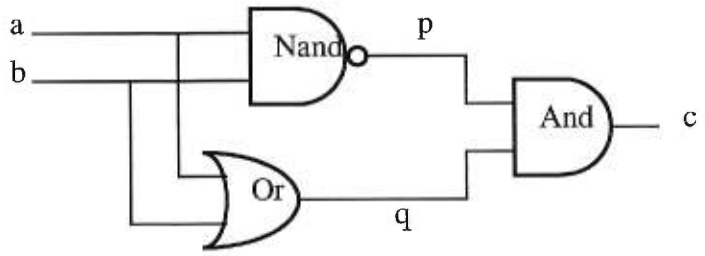



**Figure 3 -** Implementation of an Exor-Gate

Since its introduction, the HOL system has been used in the verification of numerous hardware designs. Camilleri, Gordon, and Melham demonstrated the correctness of a CMOS inverter, an n-bit CMOS full-adder, and a sequential device for computing factorial function [20]. Gordon and Herbert described the verification of memory devices with low-level timing specifications, modeling of combinational delays, and verification of a network interface chip implemented in ECL logic [40]. Other researchers outside the group at Cambridge have also used HOL for hardware verification, for example, the microprocessor called Viper was verified at the Royal Signals and Radar Establishment in England. Tahar and Kumar at the University of Karlsruhe in Germany used HOL for the verification of pipelined RISC processors [74].

Compared to the Boyer-Moore system, one important advantage of HOL is the greater ability to formulate abstractions by exploiting high-order logic. The ability to reason with high-order functions defined in terms of unspecified but well-typed functions allows one not only to concentrate on the important aspects of a problem but also to reason about a class of problems [1].

HOL has proved to be a powerful hardware verification system, deriving its strength on one hand from the expressiveness of higher-order logic, and on the other hand from the effectiveness of the automated theorem proving facilities it provides. Also, the ability to work with various abstraction mechanisms and hierarchical descriptions makes HOL very useful for handling large designs. An attractive feature is its ability to evolve continuously. New theories and associated theorems become part of the system, which can be drawn upon for future proofs. Complex derived rules, found useful in a particular context, can be saved and reused elsewhere.

Besides HOL, there are other higher-order logic theorem proving systems, such as PVS [63], Nuprl [26][52][53], etc.

### 2.1.3 Strength and weakness

First, theorem proving systems are usually very general in their applications. The ability to define appropriate theories and reason about them using a common set of inference rules provides a unifying framework within which all kinds of verification tasks can be performed.

Second, most theorem proving approaches find it easy to incorporate hierarchical verification of hardware systems. A circuit is described hierarchically, where a component is defined at one level in the hierarchy as an interconnection of components defined at lower levels. The system specification consists of a behavioral description of the components at all levels in the hierarchy. Verification involves proving that each component fulfils its part of the specification, assuming that its constituent components fulfil their specification. The specification once proven becomes a theorem and can be used in the proofs at the next hierarchical level.

Third, theorem proving approaches are better at the verification of the datapaths

than the control aspects of a circuit.

Finally, theorem proving is a deductive process. This raises both theoretical and practical complexity. Automation can be and has been provided to some degree. However, most of the theorem provers today still need much human interaction to guide the proof searching process, which makes the whole design verification process overwhelmingly tedious.

## 2.2 Equivalence checking

The purpose of equivalence checking is to verify that the functionality of a circuit is exactly the same as the one specified in its complete functional description; e.g., the circuit described as a gate netlist has the same behavior as its RTL description. There are two main methods to achieve this goal: FSM-based equivalence checking and structure-based equivalence checking.

### 2.2.1 FSM-based Equivalence checking

In this approach, both the implementation and the specification are represented as finite-state machines. One such model, called the *Moore machine*, is formally denoted by a 6-tuple $(S, I, O, NF, OF, s_0)$, where

- $S$ is a finite set of states,
- $I$ is an input alphabet,
- $O$ is an output alphabet,

- *NF*: $S \times I \rightarrow S$ is a next-state function,
- *OF*: $S \rightarrow O$ is an output function and
- $s_0$ ($s_0 \in S$) is an initial state.

In the Moore machine model, output is a function of the state alone. Another variation in which output is a function of both the state and the inputs is called the *Mealy machine* model. It has been proven that for any Moore machine $M_{moore}$, there is a Mealy machine $M_{mealy}$ equivalent to $M_{moore}$, and vice-versa [46].

To represent a sequential circuit as a Moore machine, its logic level description is organized in the form of three basic units as shown in Figure 4 — a set of latches (memorize $s \in S$), next-state logic *NL* (purely combinational, corresponds to *NF*), and output logic *OL* (purely combinational, corresponds to *OF*).

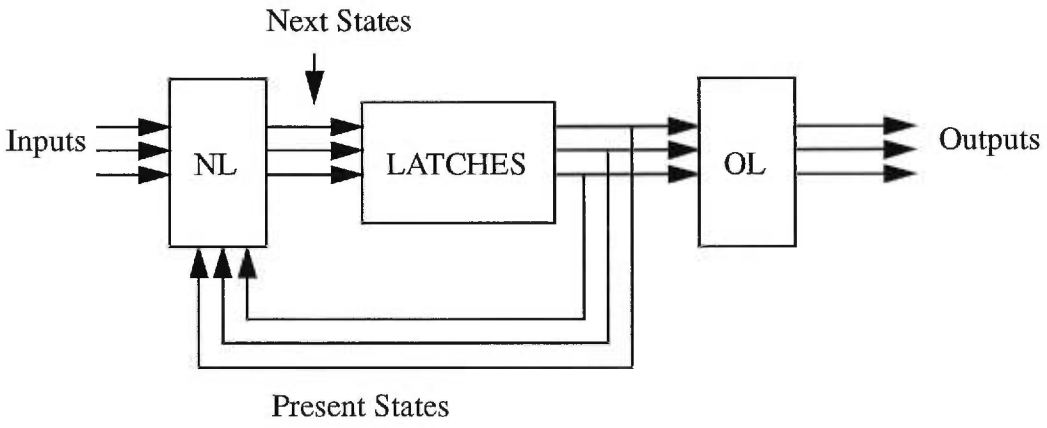



**Figure 4 -** Moore machine for sequential circuits

A finite-state machine can be viewed as a *transducer*, producing a sequence of

outputs for each possible sequence of inputs. Thus, two machines are equivalent if they produce the same output sequence for every possible input sequence.

Two machines with the same input alphabet, $M_1 = (S_1, I, O_1, NF_1, OF_1, s_{0.1})$ and $M_2 = (S_2, I, O_2, NF_2, OF_2, s_{0.2})$ can be *composed* to form a single machine $M = (S_1 \times S_2, I, O_1 \times O_2, NF, OF, (s_{0.1}, s_{0.2}))$, consisting of the two machines running in parallel [60]. The states of $M$ are pairs of states, one from $M_1$ and one from $M_2$. The state transition function $NF$ of $M$ is defined to map pairs of states to pairs of states by applying $NF_1$ to the first state in the pair and $NF_2$ to the second one. The output function is defined in the same way. $M$ is called the *product machine* of $M_1$ and $M_2$.

For checking the equivalence of the two machines, we can do a state exploration of the product machine with all possible input combinations starting from the initial state. Then, for each reachable state, we check the equivalence of the corresponding outputs of the two machines.

In this method, it is very important to represent the states and transition functions efficiently. The main limitation of this method springs from the fact that equivalence is sometimes a too strict relationship than desired for the satisfaction of a specification by an implementation.

The early exploration of this method was done by Coudert, Berthet and Madre [29][30]. They used the standard algorithm for the comparison of two Mealy machines, i.e., the output of the two machines should be the same for every transition reachable from the initial state. The significant contribution of their approach is the idea of using a *symbolic breadth-first search* of the state-transition graph of the composite machine, encoded using ROBDDs, instead of the usual depth-first techniques used in other methods.

The advantage of this method is that it is fully automated and it can handle designs with different state encoding. The drawback is that it is very costly in memory space and

time when the design has too many states.

### 2.2.2 Structure-based equivalence checking

The verification method first requires to map, one-to-one, the memory elements (flip-flops) of the two designs. Then, it checks if the corresponding combinational logic cones bounded by memory elements realize the same Boolean function. This method is normally used to compare an implementation netlist with an RTL description or with another netlist.

An important structure-based equivalence checking tool was developed by researchers at the Bull Research Centre in France. They developed a tautology-checker called PRIAM [3], which was used to verify the equivalence of a specification (expressed as a program in a hardware description language called LDS) and an implementation (also an LDS program, extracted from a structural description of the circuit, e.g., layouts, gate-level descriptions, etc.). Basically, each LDS program is reduced by symbolic execution to a canonical form of Boolean functions called a Typed Decision Graph (TDG) which is an improvement over ROBDDs [11], thereby reducing the task of checking equivalence to that of checking syntactic equality. The main drawback of this work was that both the specification and the implementation programs had to have the same states and the same state encoding, thus severely limiting its application.

This method is less prone to the state explosion problem compared to the FSM-based equivalence checking, and it can also be fully automated. However, it cannot handle sequential equivalence if the two designs have different state space and sometimes a helping hand to determine the state mapping is needed.

## 2.3 Model checking and temporal logics

One of the characteristics of the theorem proving approach is its structural rather than behavioral view of the verification process. Model checking takes the completely opposite approach. Here only the behavior of a system is checked and verified to satisfy some user-specified properties. In general, a model checker builds or accepts a finite-automaton model of the system and checks whether or not the specified property holds on the model. If it does not, the model checker returns a failure trace. Normally, the property is expressed in a temporal logic. Hence, studying temporal logics is very important for doing model checking.

### 2.3.1 Temporal logics

Propositional logic deals with absolute truths in a domain, i.e., given a domain, propositions are either true or false. Predicate logic extends the notion of truth by making it relative, in that truth of a predicate may depend on the actual arguments (variables) involved. Since these arguments can vary over elements in the domain, the truth of a predicate can also vary across the domain. Extending this notion further, modal logic provides for additional variability, where the meaning of a predicate (or a function) symbol may also change depending on what "world" it is in. Variability within a world is expressed by means of predicate arguments, whereas changes between worlds are expressed by using modal operators. The dynamic connectivity between worlds (represented as states) is specified by an accessibility relation.

Temporal Logic is a special type of Modal Logic. It provides a formal system for qualitatively describing and reasoning about how the truth values of assertions change over time. There are four basic operators in temporal logic:

- $\Box P$ is true in state $s$, if $P$ is true in all future states from $s$ (including $s$)

- $\Diamond P$ is true in state $s$, if $P$ is true in some future states from $s$

- $OP$ is true in state $s$, if $P$ is true in the next state from $s$

- $PUQ$ is true in state $s$, if either $Q$ is true in $s$ itself, or it is true in some future state of $s$, and until then $P$ is true at every intermediate state.

Historically, temporal logic was first applied by Pnueli to the task of specifying and verifying concurrent programs and reported in a landmark paper [68]. The following classes of properties were identified, all of which can be easily expressed in temporal logic:

- *Safety properties* — assert that nothing "bad" happens, typically represented as $\models \Box P$ , i.e., $P$ holds at all times in all models;

- *Liveness properties* — assert that eventually something "good" happens, typically represented as $\models P \Rightarrow \Diamond Q$, i.e., in all models, if $P$ is initially true then $Q$ will eventually be true;

- *Precedence properties* — assert the precedence order of events, typically represented as $\models P \text{ U } Q$, i.e., in all models, $P$ will hold until $Q$ becomes true;

Much work has been done in applying these ideas to hardware verification.

### 2.3.2 Classification of temporal logics

According to the details of the semantic model with respect to which temporal formulas are interpreted, temporal logics can be classified into different kinds. One important

distinction is whether the truth of a formula is determined with respect to a state, or with respect to the interval between states. The latter has given rise to what is commonly known as *Interval Temporal Logic*. Within the former one, there has been further categorization based on the difference in viewing the notion of time. In one case, time is characterized as a single linear sequence of events, leading to *Linear Time (Temporal) Logic*. In the other case, a branching view of time is taken, such that at any instant there is a branching set of possibilities into the future. This view leads to *Branching Time (Temporal) Logic*.

### 2.3.3 Propositional Linear Temporal logic (PLTL)

In a linear temporal logic the underlying structure of time is assumed to be isomorphic to the natural numbers with their usual ordering (**N**, <) [37]. Let *AP* be an underlying set of atomic proposition symbols. A *linear-time structure M=(S, x, L)* is defined such that

- $S$ is a set of states,
- $x$: $\mathbf{N} \rightarrow S$ is an infinite sequences of states, and
- $L$: $S \rightarrow 2^{(AP)}$ is a labelling of each state with the set of atomic propositions in *AP* that are true in the state.

Usually, the notation $x = (s_0, s_1, s_2, ...) = (x(0), x(1), x(2), ...)$ is employed to denote the *timeline x*, which is also referred to as a *fullpath*, or *computation sequence*, or *computation*.

The basic temporal operators of a Propositional Linear Temporal Logic (PLTL) are **F***p* ("sometime *p*", also read as "eventually *p*"), **G***p*("always *p*"; also read as "henceforth *p*"), **X***p* ("nexttime *p*"), and *p* **U** *q* ("*p* until *q*"). Their intuitive meaning is illustrated in Figure 5, where a circle represents a state, a solid circle represents a state in which *p* is

true, a shaded circle represents a state in which $q$ is true, and an arrow represents a state transition. The formulas of PLTL are built up from atomic propositions, the truth-functional connectives ($\wedge$, $\vee$, $\neg$) and the above-mentioned temporal operators.

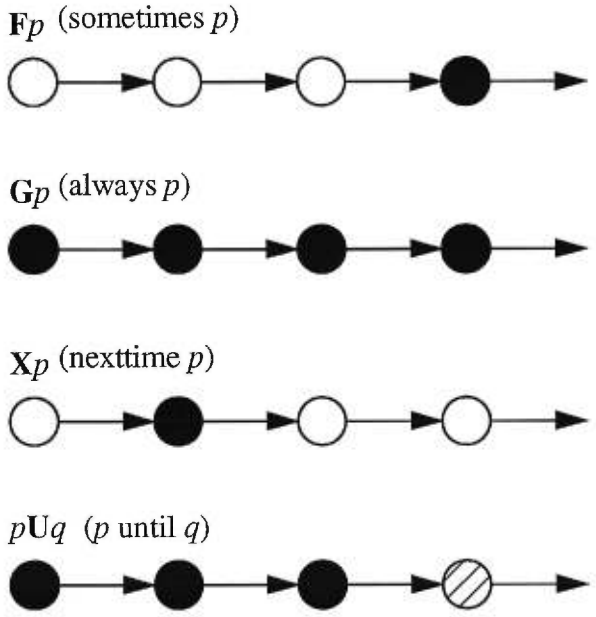



**Figure 5** - Intuition for linear-time operators.

**Syntax.** The set of formulas of PLTL is the least set of formulas generated by the following rules [37]:

(1) each atomic proposition $P$ is a formula;

(2) if $p$ and $q$ are formulas then $p \wedge q$ and $\neg p$ are formulas;

(3) if $p$ and $q$ are formulas then $p$ **U** $q$ and **X**$p$ are formulas.

The other formulas can then be introduced as abbreviations in the usual way: For the propositional connectives, $p \vee q$ abbreviates $\neg(\neg p \wedge \neg q)$, $p \Rightarrow q$ abbreviates $\neg p \vee q$, and $p \equiv q$

abbreviates $(p \Rightarrow q) \wedge (q \Rightarrow p)$. The Boolean constant *true* abbreviates $p \vee \neg p$, while *false* abbreviates $\neg true$. Then, the temporal connective $\mathbf{F}p$ abbreviates (*true* **U** $p$) and $\mathbf{G}p$ abbreviates $\neg\mathbf{F}\neg p$.

**Semantics.** the semantics of a formula $p$ of PLTL with respect to a linear-time structure $M=(S, x, L)$ is defined as follows [37]. We write $M, x \models p$ to mean that "in structure $M$ formula $p$ is true on timeline $x$." , $x^i$ denotes the suffix path $s_i , s_{i+1} , s_{i+2}$ .....: Although it is not explicitly stated, those PLTL properties are checked on all the paths

(1) $M, x \models P$ iff $P \in L(s_0)$, for atomic proposition $P$;

(2) $M, x \models p \wedge q$ iff $M, x \models p$ and $M, x \models q$,

(3) $M, x \models \neg p$ iff it is not the case that $M, x \models p$;

(4) $M, x \models p$ **U** $q$ iff $\exists j\ (x^j \models q$ and $\forall k < j\ (x^k \models p))$,

(5) $M, x \models \mathbf{X}p$ iff $x^1 \models p$.

(6) $M, x \models \mathbf{F}p$ iff $\exists j\ (x^j \models p)$;

(7) $M, x \models \mathbf{G}p$ iff $\forall j\ (x^j \models p)$;

The duality between the linear temporal operators is illustrated by the following assertions:

$\models \mathbf{G}\neg p \equiv \neg\mathbf{F}p$;

$\models \mathbf{F}\neg p \equiv \neg\mathbf{G}p$;

$\models \mathbf{X}\neg p \equiv \neg\mathbf{X}p$;

We say that a PLTL formula $p$ is *satisfiable* iff there exists a linear-time structure $M=(S, x, L)$ such that $M, x \models p$, and any such structure defines a *model* of $p$.

The following are two examples of PLTL formulas:

- $p \Rightarrow \mathbf{F}q$ intuitively means that "if $p$ is true now then at some future moment $q$ will be true."

- $\mathrm{G}(p \Rightarrow \mathbf{F}q)$ intuitively means that "whenever $p$ is true, $q$ will be true at some subsequent moment."

**Related work** One of the first examples of using PLTL for hardware verification was provided by Bochmann in manually verifying an asynchronous arbiter through reachability analysis [4].

To characterize the behavior of concurrent programs in terms of sequences of states, Pnueli proposed an abstract computational model called "fair transition systems" (FTS)[67]. An FTS consists of a set of states (not necessarily finite), some of which are specified to be initial, and a finite set of transitions. Nondeterminism is allowed by representing each transition as a function from a state to a set of states. In addition, *justice* and *fairness* requirements are included by specifying a justice set $J$ and a fairness set $F$, each of which is a set of subsets of transitions. An admissible computation of an FTS is a sequence of states and transitions, such that the starting state of the sequence is one of those designated as initial, each state follows from the previous one by an appropriate transition, and the computation terminates only if no transitions are enabled. It is also ensured that each admissible computation is just and fair, i.e., if an element of the justice (fairness) set, which is itself a set of transitions, is enabled continuously (infinitely often), then a transition belonging to that element will be taken at least once (infinitely often). PLTL formulas are interpreted over sequences of states that correspond to admissible computations of an FTS.

Lichtenstein and Pnueli presented a model checking algorithm for determining satisfiability of PLTL formulas with respect of finite state models similar to the fair transition systems FTS described above [58]. To check if a PLTL formula $\phi$ is satisfied by a program $P$, a product graph $G$ is constructed from the states of $P$ and $Cl(\phi)$ (the closure of subformulas of $\phi$). The construction of $G$ is such that $\phi$ is satisfied by $P$ if and only if there is an infinite path in $G$ from a starting state that contains $\phi$. This involves finding strongly connected components of $G$, and the overall complexity of the method is $O(|P|.2^{|\phi|})$.

Manna and Wolper used PLTL for the specification and synthesis of the synchronization part of communicating processes [59]. Sistla and Clarke proved that the problems of satisfiability and model checking in a particular finite structure are NP-complete for the PLTL logic with only the operator **F**, and are PSPACE-complete for the logics with various subsets of operators-{**F**, **X**}, {**U**}, {**X**, **U**} [73].

### 2.3.4 Computation Tree Logic (CTL)

Different kinds of Branching Time Temporal Logic (BTTL) have been proposed depending on the exact set of operators allowed. The common feature is that they are interpreted over branching tree-like time structures, where each moment may have many successor moments. The structure of time corresponds to an infinite tree. Along each path in the tree, the corresponding timeline is isomorphic to the natural numbers **N**. In BTTL, the usual temporal operators (**F**, **G**, **X**, **U**) are regarded as state quantifiers. Path quantifiers are provided to represent all path (**A**) and some path (**E**) from a given state. Here we only concentrate on the so called Computation Tree Logic (CTL), proposed first by Clarke and Emerson who also presented efficient algorithms for CTL model checking [22][23].

**Syntax**. CTL severely restricts the type of formulas that can appear after a path quantifier— only single linear time operator **F**, **G**, **X**, or **U** can follow a path quantifier and

time operators cannot be combined directly with the propositional connectives. The syntax of CTL [37] is:

1. Every atomic proposition is a CTL formula.

2. If $f$ and $g$ are CTL formulas, then so are

$\neg f$, $(f \wedge g)$, $\mathbf{AX}f$, $\mathbf{EX}f$, $\mathbf{A}\,(f\mathbf{U}g)$, $\mathbf{E}\,(f\mathbf{U}g)$

The remaining operators are derived from these according to the following rules:

$f \vee g = \neg(\neg f \wedge \neg g)$

$\mathbf{AF}g = \mathbf{A}(\text{true}\ \mathbf{U}\ g)$

$\mathbf{EF}g = \mathbf{E}(\text{true}\ \mathbf{U}\ g)$

$\mathbf{AG}f = \neg\mathbf{E}(\text{true}\ \mathbf{U}\ \neg f)$

$\mathbf{EG}f = \neg\mathbf{A}(\text{true}\ \mathbf{U}\ \neg f)$

Since all the operators are prefixed by **A** or **E**, the truth or falsehood of a formula depends only on the given state $s$, and not on the particular branch.

It was demonstrated by Clarke, Emerson and Sistla that CTL is an efficient means for verifying finite-state systems. In their approach, a finite-state system is modeled as a labelled transition graph which can be viewed as a finite *Kripke structure* represented as a triple $M=(S, R, P)$ [24], where

- $S$ is a finite set of states,

- $R$ is a total binary relation on states and represents possible transitions, and

• $P$ is a mapping that assigns to each state the set of atomic propositions that are true in the state.

A *path* within this structure is naturally defined as an infinite sequence of states, with each adjacent pair related by $R$.

**Semantics.** As its name suggests, CTL interprets temporal formulas over structures that resemble infinite computation trees. In the context defined above, given $M$ and an initial state $s_0$, it considers the infinite computation tree rooted at $s_0$, generated by considering all possible nondeterministic transitions at every state. The truth of a CTL formula is defined inductively as follows [37]:

- $(M, s_0) \models p$ iff $p \in P(s_0)$, where $p$ is an atomic proposition
- $(M, s_0) \models \neg f$ iff $(M, s_0) \not\models f$
- $(M, s_0) \models f \wedge g$ iff $(M, s_0) \models f$ and $(M, s_0) \models g$
- $(M, s_0) \models \mathbf{AX} f$ iff for all states $t$ such that $(s_0, t) \in R$, $(M, t) \models f$
- $(M, s_0) \models \mathbf{EX} f$ iff for some states $t$ such that $(s_0, t) \in R$, $(M, t) \models f$
- $(M, s_0) \models \mathbf{A}(f \mathbf{U} g)$ iff for all paths $(s_0, s_1, s_2 \ldots)$, $\exists k \geq 0$ such that $(M, s_k) \models g$, and $\forall i, 0 \leq i < k$, $(M, s_i) \models f$
- $(M, s_0) \models \mathbf{E}(f \mathbf{U} g)$ iff for some paths $(s_0, s_1, s_2 \ldots)$, $\exists k \geq 0$ such that $(M, s_k) \models g$, and $\forall i, 0 \leq i < k$, $(M, s_i) \models f$

Figure 6 shows the intuitive meaning of **AX***f*, **EX***f*, **AF***f*, **EF***f*, **AG***f* and **EG***f*. A solid circle represents a state in which $f$ is true.

Some examples of interesting properties expressible in CTL are:

- **AG**(**AF***f*): $f$ holds infinitely often along all paths.

- **AG**(*Request* $\Rightarrow$ **AXAXAX** *f*): *f* will hold in all the states which are reached by 3 transitions since a state where Request is made.
- **AG**(*Request* $\Rightarrow$ **AF** *Grant*): it is always true that if a request is made, there will eventually be a grant signal.
- **EF***f*: it is possible to reach a state where *f* holds.

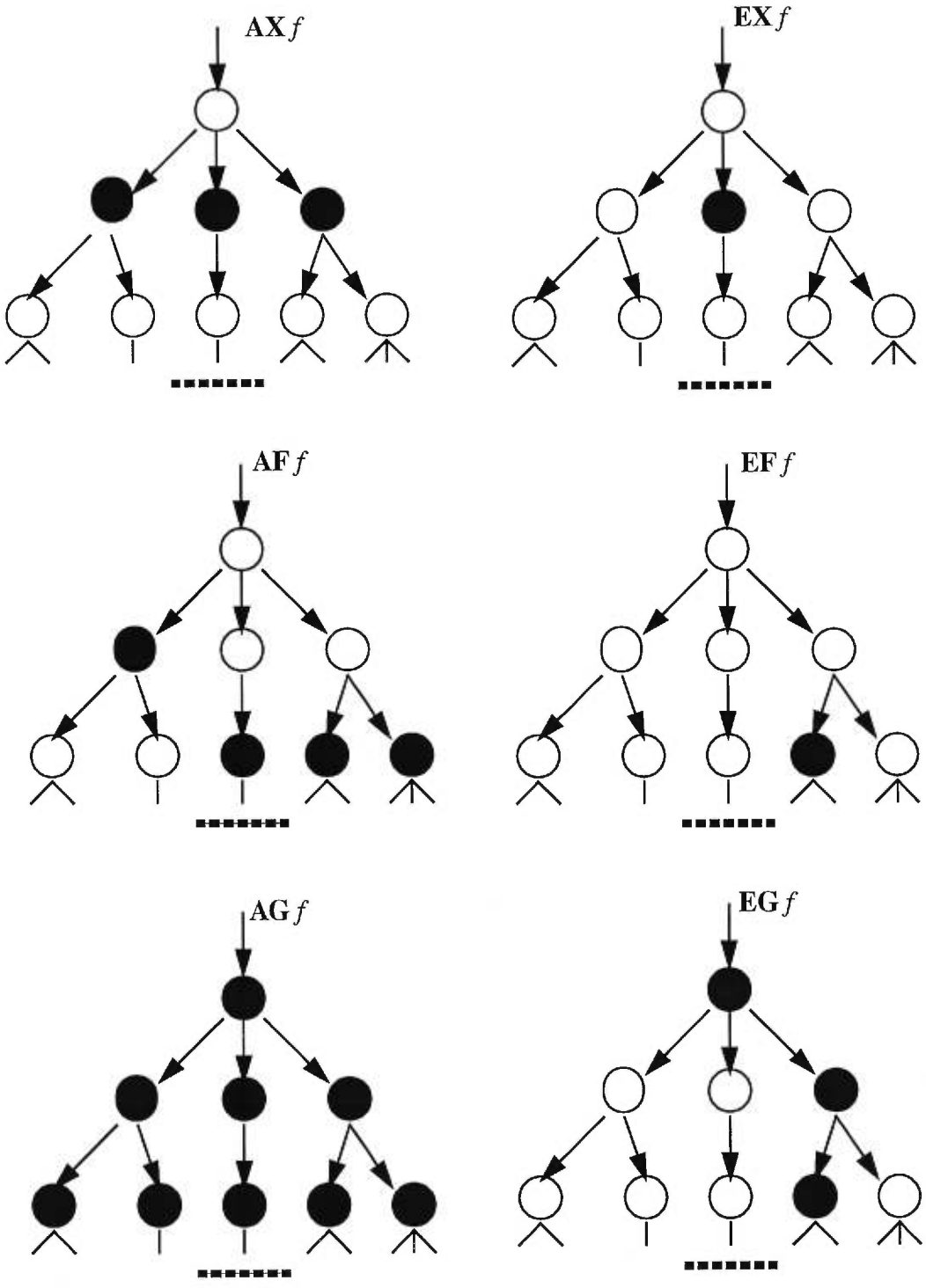



**Figure 6** - The intuitive meanings of **AX***f*, **EX***f*, **AF***f* , **EF***f*, **AG***f* and **EG***f*

**Related work**. Clarke, Emerson and Sistla showed that there is an algorithm for determining whether a CTL formula $f$ is true in state $s$ of the Kripke structure $M = (S, R, P)$ which runs in time O(length($f$)×(|$S$|+|$R$|)) [24].

An important consideration in the modeling of concurrency is the notion of *fairness*. Among possible fairness constraints, the following are very common ones [37]:

- *Unconditional fairness* (also known as *impartiality*): an infinite sequence is impartial iff every process is executed infinitely often during the computation.
- *Weak fairness* (also known as *justice*): an infinite computation sequence is weakly fair iff every process enabled almost everywhere is executed infinitely often.
- *Strong fairness* (also known simply as *fairness*): an infinite computation sequence is strongly fair iff every process enabled infinitely often is executed infinitely often.

Since fairness cannot be expressed in CTL, Clarke et al. modified the semantics of CTL to introduce the notion of fairness [24]. The new logic, called $CTL^F$, has the same syntax as CTL, but the structure is now a 4-tuple ($S$, $R$, $P$, $F$). $S$, $R$, $P$ have the same meaning as in CTL and $F$ is a collection of predicates on $S$. Fair paths in this context are defined as those along which states occurring infinitely often satisfy each predicate that belongs to $F$. $CTL^F$ has exactly the same semantics as CTL, except that the path quantifiers range over fair paths only. Model checking for $CTL^F$ is done by first identifying fair paths using strongly connected components in the graph of $M$, followed by application of the model checking algorithm only to those paths. The time complexity is $O(n \times m \times p)$ where $n = \max(|S|, |R|)$, $m$=length($f$) and $p$=$|F|$.

Since $CTL^F$ still cannot express strong fairness, Emerson and Lei defined FCTL by extending the notion of fairness in CTL to consider fairness constraints that are Boolean combinations of **F***p* (infinitely often *p*, same as **GF***p*) and **G***p* (almost always *p*, same as **FG***p*) operators [38]. Combinations of these operators can express strong fairness as well as impartiality and weak fairness. Model checking for FCTL is proved to be NP-complete in general, but it is shown to be of linear complexity when the fairness constraint is in a special canonical form.

CTL* extends CTL by allowing basic temporal operators where the path quantifier (**A** or **E**) is followed by an arbitrary linear-time formula, allowing Boolean combinations and nestings, over **F**, **G**, **X** and **U**. CTL* is sometimes informally referred to as full branching time logic. For example, **EF***p* is a CTL basic modality; **EFAF***q* is a formula of CTL (but not a basic modality) obtained by nesting **AF***q* within **EF***p* (by substituting **AF***q* for *p* in **EF***p*). **E**(**F***p*∧**F***q*) is a basic modality of CTL*. Emerson and Lei presented a model checking algorithm for CTL* which is shown to be PSPACE-complete [38].

Queille and Sifakis independently proposed a model checking algorithm for a logic with CTL modalities (without the "Until" operator) [69]. Formulas are interpreted with respect to transition systems that are derived from an interpreted Petri-net description of an implementation, with a verification system called CESAR. In their algorithm, interpretation of temporal operators is iteratively computed by evaluating fixed points of predicate transformers. However, they did not provide any means for handling fairness in their approach.

### 2.3.5 LTTL versus BTTL

In linear time logics, temporal operators are provided for describing events along a single future time line, although when a linear formula is used for program specification there is usually an *implicit universal quantification* over all possible futures. In contrast, in

branching time logics the operators usually reflect the branching nature of time by allowing *explicit quantification* over possible futures. One argument presented by the supporters of branching time logic is that it offers the ability to reason about *existential* properties in addition to *universal* properties [38].

Lichtenstein and Pnueli argued that since most formulas to be checked are small in practice, using LTTL model checking was a viable alternative to BTTL [58].

Emerson and Lei argued that branching time logic is *always better* than linear time logic for model checking [38]. They proved that given a model checking algorithm for an LTTL logic, there exists a model checking algorithm of the *same* complexity for the corresponding BTTL logic (e.g., CTL*), since BTTL is essentially path-quantification of LTTL formulas. They demonstrated that handling explicit path quantifiers and even nested path quantifiers costs (essentially) nothing.

Thus, the real issue is not which of the two (LTTL or BTTL) is better; rather, it is what basic modalities are needed in a branching time logic, i.e., what linear time formulas can follow the path quantifiers.

### 2.3.6 Symbolic Model Checking

Symbolic model checking has lately received a great deal of attention from various researchers. It was initially explored by Coudert, Madre, and Berthet [31], and independently by McMillan [61] and by Bose and Fisher [6]. The underlying idea common to these approaches is the use of symbolic Boolean representations for sets of states and transition functions (or relations) of a sequential system, in order to avoid building its global state-transition graph explicitly. Efficient symbolic Boolean manipulation techniques are then used to evaluate the truth of temporal logic formulas with respect to these models. Symbolic representation allows the regularity in state-space

of some circuits (e.g., datapaths) to be captured succinctly, thus facilitating verification of much larger circuits compared to the explicit state enumeration techniques, as shown by Burch et al. [17].

McMillan presented a method for model checking that avoids the state explosion problem by representing the Kripke model implicitly with a Boolean formula described using Bryant's ROBDDs [11]. It allows a CTL model checking algorithm to be implemented using well developed automatic techniques for manipulating Boolean formulas. Since the Kripke model is symbolically represented, there is no need to actually construct it as an explicit data structure. Hence, the state explosion problem can be reduced, although there exist pathological examples of explosive growth complexity.

In Carnegie Mellon University, Clarke, McMillan et. al. developed the Symbolic Model Verifier (SMV) [17][62][61]. SMV is a tool for checking finite state systems against specifications in CTL. McMillan and Schwalbe successfully applied SMV to the verification of the Encore Gigamax cache consistency protocol and found some critical design errors [62], thus demonstrating the effectiveness of symbolic model checking techniques for industrial applications.

The method used by Burch et al. [17] is very general and can handle nondeterministic systems, thus allowing its application to both synchronous and asynchronous circuits. However, this generality is gained at the cost of increased complexity of representing the complete transition relation symbolically using Bryant's ROBDDs [11]. Bose and Fisher, on the other hand, model systems as deterministic Moore machines, and use symbolic representations of the next-state functions (not relations) using ROBDDs [6]. The latter are derived directly from the symbolic simulation of the circuit using the switch-level simulator COSMOS [13]. Coudert at al. also used a deterministic Moore machine model with symbolic representations of the next-state functions [31]. However, they used more sophisticated Boolean manipulation operations (e.g. "constraint" and "restrict" operators) to keep down the size of their internal data

representations called TDGs (Typed Decision Graphs). TDGs are similar to ROBDDs, but typically they occupy about 1/2 of the space required by ROBDDs.

Bryant and Seger have presented another extreme in this spectrum of symbolic methods [12]. They avoided explicit representation even of the next-state function. Instead, they used the simulation capability of COSMOS to symbolically compute the next-state of each circuit node of interest. This restricts them to using a limited form of temporal logic that can express properties over finite sequences only (unlike the other approaches that can handle full CTL). They reason within a symbolic Ternary algebra (with logic values 0,1 and X) to compute the truth values of formulas.

### 2.3.7 Available Model Checkers

A number of BDD-based model checking tools have developed over the last 10 years. The well known ones are as follows:

- SMV (Symbolic Model Verifier) [61][84]: a CTL model checker developed by McMillan at Carnegie-Mellon University.
- VIS (Verification Interacting with Synthesis) [9] [10][83]: an integrated tool for verification, simulation and synthesis of finite state systems, developed at University of California at Berkeley. It contains a Fair CTL Model Checker and a behavioral equivalence checker for sequential circuits, language emptiness check for Büchi automata and combinational verification.
- FormalCheck [57][85][86]: an ω-automata based model checker based on Cospan developed at Bell Labs Design Automation, Lucent Technologies. The reduction algorithms and refinement methodologies embedded in FormalCheck makes the tool applicable to industrial-size designs.

- CheckOFF-M [87]: a model checker for a branching time interval logic developed at Siemens and commercialized by Abstract Hardware Limited Corporation.

### 2.3.8 Strength and weakness

The most significative advantage of model checking techniques is that they can be made completely automatic. The drawback of these approaches is that they are not general in the way that theorem provers are. A model checking verification system will work only for the kind of logic and models that it is designed for, and the state explosion problem is a major obstacle for model checking to be widely used in an industrial design flow.

Compared to theorem proving, model checking approaches are weak for dealing with hierarchical verification and abstraction; however, they are better at reasoning about the control aspects of circuits and are more automatic.

## Summary

In this chapter, we reviewed the existing formal verification techniques: theorem proving, equivalence checking and model checking, and especially the different logics deployed in the various verification techniques. These constitute the theoretical background on which the thesis is based.

# 3 Abstract Description of State Machines

Abstract description of State Machines (ASMs) is a model used for describing hardware designs at the Register Transfer Level (RTL). It was first introduced by Corella, Langevin, Cerny, Zhou, and Song [27] [28]. Using ASMs, a data value can be represented by a single variable of abstract type, rather than by a vector of Boolean variables, and a data operation is represented by an uninterpreted function symbol. The model checking method based on a first-order temporal logic as developed in this thesis allows to verify properties on designs represented by ASMs. Thus, it is necessary to review first the terminology related to ASMs. We also give the definition of an *abstract computation forest* on which we then define the semantics of our first-order temporal logic.

## 3.1 A many-sorted first-order logic

As in an ordinary many-sorted first-order logic, the vocabulary consists of *sorts*, *constants*, *variables*, and *function symbols* (or *operators*). Constants and variables have sorts. An n-ary function symbol ($n > 0$) has a type $\alpha_1 \times \alpha_2 \times ... \times \alpha_n \rightarrow \alpha_{n+1}$, where $\alpha_1$ ...

$\alpha_{n+1}$ are sorts. We deviate from standard many-sorted first-order logic by introducing a distinction between *concrete* (or *enumerated*) *sorts*, and *abstract sorts*; the difference is that concrete sorts have *enumerations*, while abstract sorts do not. The enumeration of a concrete sort $\alpha$ is a set of distinct constants of sort $\alpha$. We refer to constants occurring in enumerations as *individual constants*, and to other constants as *generic constants.* An individual constant can appear in the enumeration of more than one sort $\alpha$, and is said to be of sort $\alpha$ for each of them. Variables and generic constants, on the other hand, have unique sorts.

The distinction between abstract and concrete sorts leads to a distinction between three kinds of function symbols. Let $f$ be a function symbol of type $\alpha_1 \times \alpha_2 \times ... \times \alpha_n \rightarrow \alpha_{n+1}$. If $\alpha_{n+1}$ is an abstract sort then $f$ is an *abstract function symbol.* If all the $\alpha_1 ... \alpha_{n+1}$ are concrete, $f$ is a *concrete function symbol.* If $\alpha_{n+1}$ is concrete while at least one of $\alpha_1 ... \alpha_n$ is abstract, then $f$ is referred to as a *cross-operator.* Both abstract function symbols and cross-operators may be *uninterpreted*, or *partially interpreted* by conditional rewrite rules. However, a concrete function symbol must have an explicit definition, and the symbol as such never appears in a logic expression.

The *terms* and their *types* (*sorts*) are defined inductively as follows: a constant or a variable of sort $\alpha$ is a term of type $\alpha$; and if $f$ is a function symbol of type $\alpha_1 \times \alpha_2 \times ... \times \alpha_n \rightarrow \alpha_{n+1}$, $n \geq 1$, and $A_1, .. ., A_n$ are terms of types $\alpha_1 ... \alpha_n$, then $f(A_1,...,A_n)$ is a term of type $\alpha_{n+1}$. A term consisting of a single occurrence of an individual constant has multiple types (the sorts of the constant) but every other term has a unique type. We say that a term, variable or constant is concrete (resp. abstract) to indicate that it is of concrete (resp. abstract) sort. A term is *concretely reduced* iff it contains: (i) the individual constants; (ii) the abstract generic constants; (iii) the abstract variables; and (iv) the terms of the form $f(A_1,...,A_n)$ where $f$ is an abstract function symbol and $A_1, .. ., A_n$ are concretely-reduced terms. Thus, the concretely-reduced terms are those that have no concrete subterms other than individual constants. A term of the form $f(A_1, ..., A_n)$ where $f$ is a cross-operator and

$A_1 \ldots A_n$ are concretely-reduced terms is called a *cross-term*. An *equation* is an expression $A_1 = A_2$ where $A_1$ and $A_2$ are terms of same type $\alpha$. *Atomic formulas* are the equations, plus **T** (truth), and **F** (falsity). Formulas are built from the atomic formulas in the usual way using logical connectives and quantifiers.

An *interpretation* is a mapping $\psi$ that assigns a denotation to each sort, constant and function symbol such that:

1. The denotation $\psi(\alpha)$ of an abstract sort $\alpha$ is a non-empty set.

2. If $\alpha$ is a concrete sort with enumeration $\{a_1, a_2, \ldots, a_n\}$ then $\psi(\alpha) = \{\psi(a_1), \psi(a_2), \ldots, \psi(a_n)\}$ and $\psi(a_i) \neq \psi(a_j)$ for $1 \leq i < j \leq n$.

3. If $c$ is a generic constant of sort $\alpha$, then $\psi(c) \in \psi(\alpha)$. If $f$ is a function symbol of type $\alpha_1 \times \ldots \times \alpha_n \rightarrow \alpha_{n+1}$, then $\psi(f)$ is a function from the cartesian product $\psi(\alpha_1) \times \ldots \times \psi(\alpha_n)$ into the set $\psi(\alpha_{n+1})$.

Let $X$ be a set of variables, a *variable assignment* with domain $X$ compatible with an interpretation $\psi$ is a function $\varphi$ that maps every variable $x \in X$ of sort $\alpha$ to an element $\varphi(x)$ of $\psi(\alpha)$. We write $\Phi_X^{\psi}$ for the set of $\psi$-compatible assignments to the variables in $X$, $\psi,\varphi \models P$ if a formula $P$ denotes truth under an interpretation $\psi$ and a $\psi$-compatible variable assignment $\varphi$ to the variables that occur free in $P$, $\models P$ if a formula $P$ denotes truth under every interpretation $\psi$ and every $\psi$-compatible variable assignment to the variables that occur free in $P$.

## 3.2 Directed Formulas (DFs)

Given two disjoint sets of variables $U$ and $V$, a *directed formula* of type $U \rightarrow V$ is a formula

in disjunctive normal form (DNF) such that:

1. Each disjunct is a conjunction of equations of the form

   $A = a$, where $A$ is a term of concrete sort $\alpha$ of the form "$f(B_1, \ldots, B_n)$" ($f$ is thus a cross-operator) that contains no variables other than elements of $U$, and $a$ is an individual constant in the enumeration of $\alpha$, or

   $w = a$, where $w \in (U \cup V)$ is a variable of concrete sort $\alpha$ and $a$ is an individual constant in the enumeration of $\alpha$, or

   $v = A$, where $v \in V$ is a variable of abstract sort $\alpha$ and $A$ is a term of type $\alpha$ containing no variables other than elements of $U$.

2. In each disjunct, the left hand sides (LHSs) of the equations are pairwise distinct.

3. Every abstract variable $v \in V$ appears as the LHS of an equation $v = A$ in each of the disjuncts. (Note that there need not be an equation $v = a$ for every concrete variable $v \in V$.)

Intuitively, in a DF of type $U \rightarrow V$, the $U$ variables play the role of independent variables, the $V$ variables play the role of dependent variables, and the disjuncts enumerate possible cases. In each disjunct, the equations of the form $u = a$ and $A = a$ specify a case in terms of the $U$ variables, while the other equations specify the values of (some of the) $V$ variables in that case. The cases need not be mutually exclusive, nor exhaustive.

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

accomplished by case splitting.

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

Let $P$ be a DF of type $U \rightarrow V$. For a given interpretation $\psi$, $P$ can be used to represent the set of vectors $Set_V^{\psi}(P) = \{ \phi \in \Phi_V^{\psi} \mid \psi, \phi \models (\exists U)P \}$.

In the following sections, DFs are used for two distinct purposes: to represent sets (viz. sets of states as well as sets of input vectors and output vectors) and to represent relations (viz. the transition and output relations).

## 3.3 Basic algorithms of DFs

We recall the basic algorithms used in [27] [80], but here we give their definitions in terms of DFs, since those algorithms will be needed later in the model checking procedures.

***Disjunction*:** The disjunction algorithm is $n$-ary. It takes as inputs a set of DFs $P_i$, $1 \le i \le n$, of types $U_i \rightarrow V$, and produces a DF $R = \mathbf{Disj}(\{P_i\}_{1 \le i \le n})$ of type $(\bigcup_{1 \le i \le n} U_i) \rightarrow V$ such that

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

Note that this algorithm requires that all the $P_i$, $1 \leq i \leq n$, have the same set of abstract primary variables. If two DFs $P_1$, $P_2$ do not have the same set of abstract primary variables, then there is no DF $R$ such that $\models R \Leftrightarrow (P_1 \vee P_2)$.

***Conjunction***: The conjunction algorithm takes as inputs a set of DFs $P_i$, $1 \leq i \leq n$, of types $U_i \rightarrow V_i$ and produces a DF $R = \text{Conj}(\{P_i\}_{1 \leq i \leq n})$ of type

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

such that $\models R \Leftrightarrow (\bigwedge_{1 \leq i \leq n} P_i)$. Note that for $1 \leq i < j \leq n$, $V_i$ and $V_j$ must not have any abstract variables in common, otherwise the conjunction cannot be computed.

***Relational product:*** The algorithm takes as inputs a set of DFs $P_i$, $1 \leq i \leq n$, of types $U_i \rightarrow V_i$, a set of variables $E$ to be existentially quantified, and a renaming substitution $\eta$, and produces a DF $R = \textbf{RelP}(\{P_i\}_{1 \leq i \leq n}, E, \eta)$ such that

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

The algorithm computes the conjunction of the $P_i$, existentially quantifies the variables in $E$, and applies the renaming substitution $\eta$. For $1 \leq i < j \leq n$, $V_i$ and $V_j$ must not have any abstract variables in common.

The result of only computing the conjunction is a DF of type

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

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

$$((\bigcup_{1 \le i \le n} U_i) \backslash (\bigcup_{1 \le i \le n} V_i)) \rightarrow ((\bigcup_{1 \le i \le n} V_i) \backslash E) \quad .$$

The domain of $\eta$ must be a subset of $((\bigcup_{1 \le i \le n} V_i) \backslash E)$. The type of the result $R$ is then

$$((\bigcup_{1 \le i \le n} U_i) \backslash (\bigcup_{1 \le i \le n} V_i)) \rightarrow (((\bigcup_{1 \le i \le n} V_i) \backslash E) \cdot \eta) \quad .$$

***Pruning by subsumption:*** The algorithm takes as inputs two DFs $P$ and $Q$ of types $U \rightarrow V_1$ and $U \rightarrow V_2$ respectively, and produces a DF $R$ = PbyS($P$, $Q$) of type $U \rightarrow V_1$ derivable from $P$ by *pruning* (i.e., by removing some of the disjuncts) such that

$$\models R \vee (\exists U)\, Q \Leftrightarrow P \vee (\exists U)\, Q \qquad (3.1).$$

The disjuncts that are removed from $P$ are *subsumed* by $Q$, hence the name of the algorithm.

Since $R$ is derivable from $P$ by pruning, after the formulas represented by $R$ and $P$ have been converted to DNF, the disjuncts in the DNF of $R$ are a subset of those in the DNF of $P$. Hence

$\models R \Rightarrow P$. And, from (3.1), it follows tautologically that

$\models P \wedge \neg(\exists U)Q \Rightarrow R.$ (3.2).

Thus we have

$\models (P \wedge \neg(\exists U)Q \Rightarrow R) \wedge (R \Rightarrow P).$ (3.3).

We can then view $R$ as approximating the logical difference of $P$ and $(\exists U)Q$. In general, there is no DF logically equivalent to $P \wedge \neg(\exists U)Q$. If $R$ is **F**, then it follows tautologically from (3.1) that $\models P \Rightarrow (\exists U)Q$.

## 3.4 Abstract Description of State Machines (ASMs)

An *abstract description of state machine M* is a tuple $D = (X, Y, Z, F_I, F_T, F_O)$, where

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

   Given an interpretation $\psi$, an input vector of the state machine $M$ represented by $D$ is a $\psi$-compatible assignment to the set of input variables $X$; thus the set of input vectors, or input alphabet, is $\Phi_X^{\psi}$. Similarly, $\Phi_Z^{\psi}$ is the set of output vectors. A *state* is a $\psi$-compatible assignment to the set of state variables $Y$; hence, the state space is $\Phi_Y^{\psi}$. A state $\phi$ can also be described by an assignment $\phi' = \phi \circ \eta^{-1} \in \Phi_Y^{\psi}$ to the next state variables.

A variable in $X \cup Y \cup Z$ is called an *ASM_variable.*

2. $F_I$ is a DF representing the set of initial states, of type $U \rightarrow Y$, where $U$ is a set of abstract variables disjoint from $X \cup Y \cup Y' \cup Z$. Typically, $F_I$ is a one-disjunct DF representing the set of initial states.

   Given an interpretation $\psi$, a state $\phi \in \Phi_Y^{\psi}$ is an initial state iff $\psi, \phi \models (\exists U) F_I$.

   Thus the set of *initial states* is $S_I = Set^{\psi}(F_I) = \{\phi \in \Phi_Y^{\psi} \mid \psi, \phi \models (\exists U) F_I\}$.

3. $F_T$ is a DF of type $(X \cup Y) \rightarrow Y'$ representing the transition relation.

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

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

4. $F_O$ is a DF of type $(X \cup Y) \rightarrow Z$ representing the output relation.

   Given an interpretation $\psi$, the output relation of the state machine $M$ represented by $D$ is $R_O = \{ (\phi, \phi', \phi'') \in \Phi_X^{\psi} \times \Phi_Y^{\psi} \times \Phi_Z^{\psi} \mid \psi, \phi \cup \phi' \cup \phi'' \models F_O \}$.

For every interpretation $\psi$ of the sorts, constants and function symbols of the logic, the abstract description $D = (X, Y, Z, F_I, F_T, F_O)$ represents the state machine $M = ( \Phi_X^{\psi},$

$\Phi_Y^\Psi$, $\Phi_Z^\Psi$, $S_I, R_T, R_O)$ with the set of input vectors $\Phi_X^\Psi$, the state space $\Phi_Y^\Psi$, the set of output vectors $\Phi_Z^\Psi$, the set of initial states $S_I$, the transition relation $R_T$, and the output relation $R_O$.

Let $P_1, P_2$ be two DFs of type $U \rightarrow Y$. Then for a given interpretation $\psi$, the two sets of states represented by $P_1, P_2$ are respectively $S_1 = Set^\Psi(P_1) = \{\phi \in \Phi_Y^\Psi \mid \psi, \phi \models (\exists U)P_1\}$ and $S_2 = Set^\Psi(P_2) = \{\phi \in \Phi_Y^\Psi \mid \psi, \phi \models (\exists U)P_2\}$. We say that $P_1$ and $P_2$ are *equivalent* DFs (and furthermore $S_1$ and $S_2$ are equivalent sets) if $\mathbf{PbyS}(P_1, P_2) = \mathbf{F}$ and $\mathbf{PbyS}(P_1, P_2) = \mathbf{F}$.

## 3.5 State Enumeration and Invariant Checking

Given an abstract description of an ASM $D = (X, Y, Z, F_I, F_T, F_O)$, we can compute the set of the reachable states of the machine $M = (\Phi_X^\Psi, \Phi_Y^\Psi, \Phi_Z^\Psi, S_I, R_T, R_O)$ represented by $D$, for any interpretation $\psi$, using the DF algorithms of Section 2.3. At the same time we can check that a given condition on the outputs of the machine, the *invariant*, holds on all these states. (When doing state exploration only, steps 6, 7 and 8 in the following algorithm are skipped.)

An invariant is represented by a DF $C$ of type $V \rightarrow Z$, where $W$ is a set of abstract variables disjoint from $X, Y, Y', Z$ and $U$. For a given interpretation $\psi$, an output vector $\varphi \in \Phi_Z^\Psi$ (an assignment to the output variables) is deemed to satisfy the invariant iff $\psi,\varphi \models (\exists V)C$; therefore, the set of output vectors that satisfy the invariant is $Set_Z^\Psi(C) = \{\varphi \in$

$\Phi_Z^{\Psi} \mid \psi, \varphi \models (\exists V)C \}$.

The invariant checking algorithm based on reachability analysis is as follows:

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

Variables $I$, $N$, $P$, $Q$ and $R$ represent sets of states, and $O$ represents a set of output vectors. Before each iteration, $R$ contains the states reached so far, while $Q$ is the frontier set, i.e., a subset of $Set_Y^{\Psi}(R)$ containing at least all those states that entered $Set_Y^{\Psi}(R)$ for the first time in the previous iteration. In line 5, **Fresh**($X$, $K$) constructs a one-disjunct DF representing a conjunction of equations $x = u$, one for each abstract input variable $x \in X$, where $u$ is a fresh variable from the set of auxiliary abstract variables $U$. The value of the loop counter $K$ is used to generate the fresh variables. This one-disjunct DF is assigned to $I$, which represents the set of input vectors. In line 6, the relational product operation is used to compute the DF representing the set of output vectors produced by the states in the frontier set. The resulting DF is assigned to $O$. Then, in line 7, the pruning-by-subsumption operation is used to remove from $O$ those disjuncts that represent output vectors which satisfy the invariant $C$. The resulting DF is assigned to $P$. In line 8, if $P$ is

not **F**, then the procedure stops and reports failure. If $P$ is **F**, then$Set_Z^{\Psi}(O) \subseteq Set_Z^{\Psi}(C)$, i.e., every output vector produced by a state in the frontier set satisfies the invariant, and the verification procedure continues. In line 9, the relational product operation is used again, this time we compute the DF representing the set of states that can be reached in one step from the frontier set of states. Note that the DF $Q$ representing the frontier set is of type $U \rightarrow Y$, the DF $I$ representing the set of input vectors is of type $U \rightarrow X$, and the DF $F_T$ representing the transition relation is of type $(X \cup Y) \rightarrow Y'$. The result of taking the conjunction of these three DFs would be of type $U \rightarrow (X \cup Y \cup Y')$, the result of subsequently removing the variables in $X \cup Y$ by existential quantification would be of type $U \rightarrow Y'$, and the result of subsequently applying the renaming substitution $\eta$ is thus of type $U \rightarrow Y$. The **RelP** operation performs these three operations in one pass, and assigns the resulting DF of type $U \rightarrow Y$ to $N$. Lines 10 and 11 check whether $Set_Y^{\Psi}(N) \subseteq Set_Y^{\Psi}(R)$by the same method used in lines 7 and 8 to check whether $Set_Z^{\Psi}(O) \subseteq Set_Z^{\Psi}(C)$. If this is indeed the case then every state reachable from the frontier set was already in $Set_Y^{\Psi}(R)$. The fixpoint has been reached and $R$ represents all the reachable states. Therefore, the procedure terminates and reports success. Otherwise the DF assigned to $Q$ in line 10 represents the new frontier set. Line 12 simplifies $R$ by removing from it any disjuncts that are subsumed by $Q$, using **PbyS**. There may be such disjuncts because $Q$ was not computed earlier as an exact difference. Line 13 then computes the new value of $R$ by taking the disjunction of $R$ and $Q$ which represents the set of states $Set_Y^{\Psi}(R) \cup Set_Y^{\Psi}(Q)$ and assigns it to $R$.

## 3.6 Abstract Computation Forest

Given an ASM $D = (X, Y, Z, F_I, F_T, F_O)$, for a given interpretation $\psi$, we describe the next-

state computation of the machine $M = ( \Phi_X^{\Psi}, \Phi_Y^{\Psi}, \Phi_Z^{\Psi}, S_I, R_T, R_O)$ represented by $D$ as a *computation forest* $F = (V, E)$ which may contain an infinite number of infinite trees:

- $V$ is a set nodes. Each node in $V$ represents a *total_state:* which is an $\Psi$-compatible assignment to the set of state, input and output variables. A total_state can be described as $s = (\phi, \phi', \phi'') \in \Phi_X^{\Psi} \times \Phi_Y^{\Psi} \times \Phi_Z^{\Psi} \mid \Psi, \phi \cup \phi' \cup \phi'' \models F_O$.
- Each edge in $E$ is a pair of total_states $<s_i, s_j>$, indicating that total_state $s_j$ is derived from total_state $s_i$ by one transition step. This can be formally described as follows:

$$s_i = (\phi_i, \phi_i', \phi_i'') \in \Phi_X^{\Psi} \times \Phi_Y^{\Psi} \times \Phi_Z^{\Psi} \mid \Psi, \phi_i \cup \phi_i' \cup \phi_i'' \models F_O;$$

$$s_j = (\phi_j, \phi_j', \phi_j'') \in \Phi_X^{\Psi} \times \Phi_Y^{\Psi} \times \Phi_Z^{\Psi} \mid \Psi, \phi_j \cup \phi_j' \cup \phi_j'' \models F_O;$$

and $\Psi, \phi_i \cup \phi_i' \cup (\phi_j' \circ \eta^{-1}) \models F_T$.

A *computation path* in a computation forest is an infinite sequence of total_states $s_0, s_1, s_2, ....., s_n, .....$ such that $\forall i \geq 0, <s_i, s_{i+1}> \in E$. We use $\pi_0 = ( s_0, s_1, s_2, .... )$ to denote a full path and $\pi_i = ( s_i, s_{i+1}, s_{i+2}, ...)$ a suffix path starting from $s_i$.

## 3.7 The MinMax example

To show how a computation forest is derived, we use a simplified version of the **MinMax** state machine which first appeared in [30]. The machine has 2 input variables $X = \{r, x\}$ and 3 state variables $Y = \{c, m, M\}$, where $r$ and $c$ are of the Boolean sort $B$, a concrete sort with enumeration $\{0, 1\}$, and $x$, $m$, and $M$ are of an abstract sort *wordn*. The intended interpretation of *wordn* is a finite set equipped with a total order $\leq$, e.g., the set of 64-bit signed integers. There are no output variables from this circuit, i.e., $Z = \Phi$.

A graphical representation of the **MinMax** state machine is shown in Figure 7: the circles correspond to the values of the control state variable $c$ and the arrows correspond to the control transitions of the machine. The transition labels specify the conditions under which each transition is taken and an assignment of values to the abstract next-state variables $m'$ and $M'$.

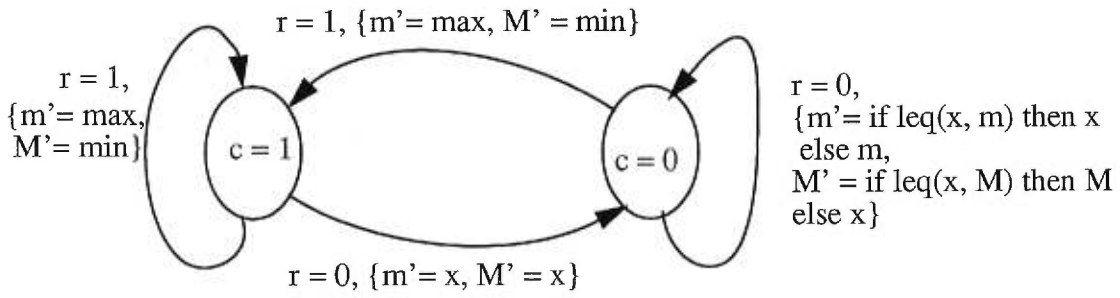



**Figure 7** - A graphical representation of the MinMax state machine

The machine stores in $m$ and $M$, respectively, the smallest and the greatest values presented at the input $x$ since the last reset ($r = 1$). When the machine is reset, $m$ is loaded by the maximal possible value **max** and $M$ by the minimal possible value **min**. The

smallest and greatest values are computed using an operator **leq** such that for any two values $a$ and $b$ of sort *wordn*, **leq**($a$, $b$) = 1 if and only if $a$ is less than or equal to $b$. Formally, the intended denotations of **min** and **max** are the smallest and largest elements of the total order (*wordn*, $\leq$), and the intended denotation of **leq** is the characteristic function of the order relation $\leq$. In the abstract description of the **MinMax** state machine, the abstract sort *wordn* is uninterpreted, the **min** and **max** symbols are uninterpreted generic constants of sort *wordn*, and **leq** is an uninterpreted cross-operator of type $wordn \times wordn \rightarrow B$.

Assuming that the DF representing the set of initial states is $F_I$: c=1 $\wedge$ m= max $\wedge$ M= min, where max and min are generic constants, Figure 8 shows a part of one tree in the infinite computation forest of the **MinMax** state machine. The dotted lines represent the continuation of the tree. Each square box in Figure 8 represents a state (G, $\varphi$) which is an assignment $\varphi$ to the state variables satisfying a guard $G$. When $G$ is True, it is satisfied by any assignment. One square box plus the assignment to the input variables indicated on the arrow from that box can be seen as a total_state. For a variable $x$ of sort $\alpha$, the assignment $\varphi(x)$ represents any value of $\psi(\alpha)$; therefore, there may exist an infinite number of instances of a tree in the computation forest.

# Summary

In this chapter, we reviewed the definitions and the formal logic used in our ASM approach. We also gave the definition of an abstract infinite computation tree. It is based on this computation model that the first-order temporal logic model checking method was developed in this thesis.

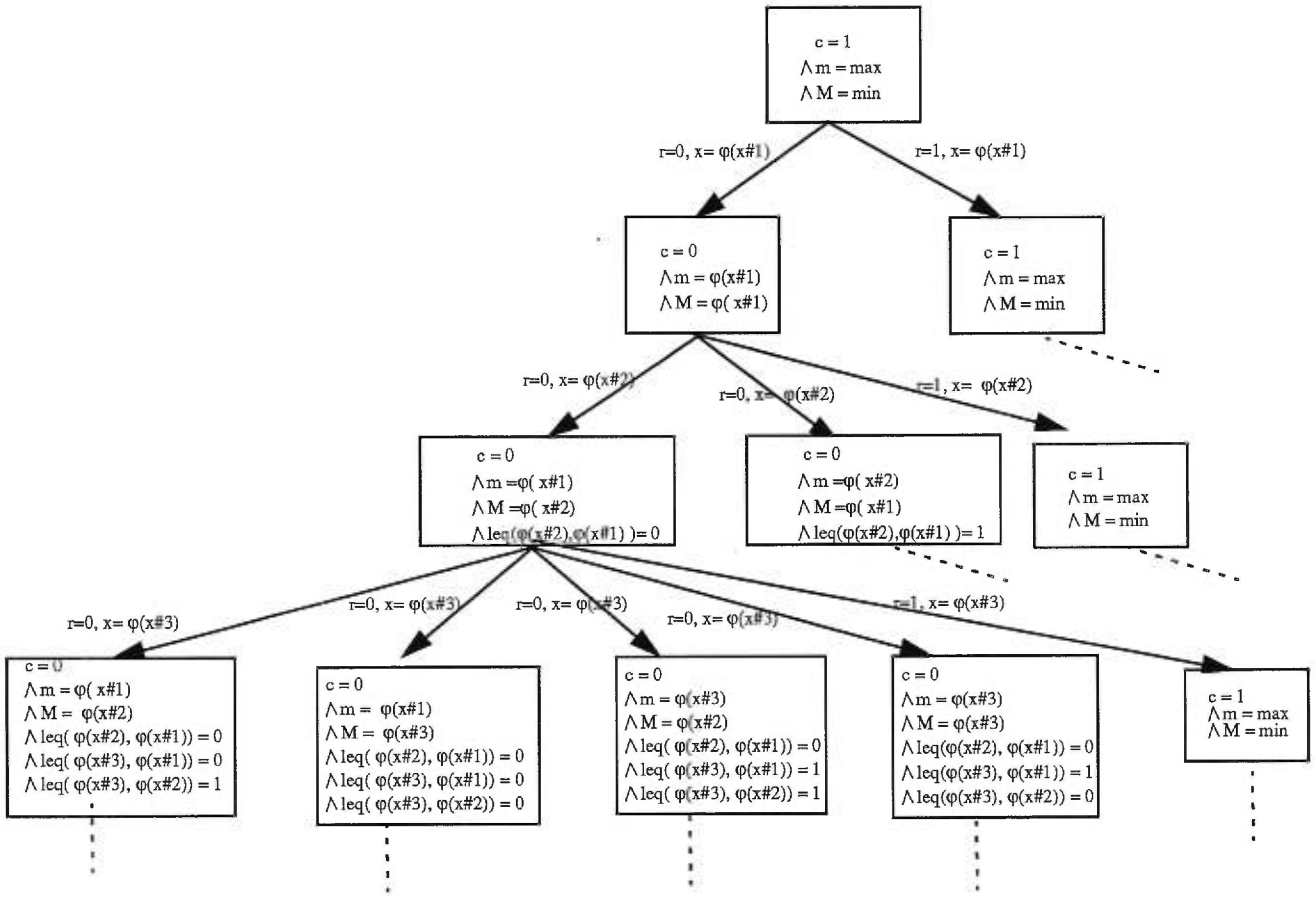



**Figure 8 -** Abstract computation tree of the MinMax example

# 4 A First-Order Branching Time Temporal Logic: Abstract_CTL*

As a propositional branching time temporal logic, CTL (Computational Tree Logic) developed by Clarke and Emerson [22] is widely used as a property specification language for model checking. CTL severely restricts the type of formulas that can appear after a path quantifier **A** or **E**, namely, only single linear temporal operator **F, G, X,** or **U** can follow a path quantifier, and temporal operators cannot be combined directly using propositional connectives.

CTL* extends CTL by allowing temporal operators in which a path quantifier (**A** or **E**) is followed by an arbitrary linear-time formula, allowing Boolean combinations and nesting over **F, G, X** and **U**. CTL* is sometimes informally referred to as a full branching time logic. For example, **EF***p* is a basic CTL modality; **EFAF***q* is a formula of CTL (but not a basic modality) obtained by nesting **AF***q* within **EF***p* (by substituting **AF***q* for *p* in **EF***p*). **E(F***p*∧**F***q)* is a basic modality of CTL*. Below we extend CTL* from the propositional logic level to the first-order logic level and define a first-order branching time temporal logic: *Abstract_CTL**.

## 4.1 Syntax of Abstract_CTL*

Given an abstract description of an ASM and a set of *ordinary variables* which are available for use in the specification of the property to be verified, the *state formulas* are defined as follows:

(S1) if $t_1$ is an ASM_variable, $t_2$ is an ASM_variable, or a constant, or an ordinary variable, then the equation $t_1 = t_2$ is a state formula.

(S2) if $p$, $q$ are state formulas, then so are $!p$, $p\&q$, $p|q$ *and* $p->q$;

(S3) if $t$ is an ASM_variable, $v$ an ordinary variable, and $p$ a state formula, then

**LET** $(v = t)$ **IN** $p$ is a state formula;

Note: the **LET** construct allows us to use an ordinary variable $v$ to remember the value of an ASM_variable $t$ at the current state.

(S4) if $p$ is a path formula, then **A***p* and **E***p* are state formulas.

The *path formulas* are defined as follows:

(P1) each state formula is also a path formula;

(P2) if $p$, $q$ are path formulas then so are $!p$, $p\&q$, $p|q$, $p->q$, **X***p*, **G***p*, **F***p* and *p***U***q*;

(P3) If $t$ is an ASM_variable, $v$ an ordinary variable, and $p$ a path formula, then

**LET** $(v = t)$ **IN** $p$ is a path formula.

We allow the formula **LET** $(v_1 = t_1)$ & ... & $(v_n = t_n)$ **IN** $p$ as a shorthand for

**LET** $(v_1 = t_1)$ **IN** ( ( **LET** $(v_1 = t_1)$ **IN** (..... **LET** $(v_n = t_n)$ **IN** $p$ ) ) ) . And we call $(v_1 = t_1)$ & ...& $(v_n = t_n)$ a *Let_equation.*

## 4.2 Semantics of Abstract_CTL*

A formula of Abstract_CTL* is interpreted in terms of a computation forest $F$ derived from an ASM under a given interpretation $\psi$.

A state formula (resp. path formula) has a meaning relative to a total_state (or a path) and the assignment to the ordinary variables. We use $Val_{\phi \cup \sigma}(t)$ to denote the value of variable $t$ under a $\psi$-compatible assignment $\phi$ to the state, input and output variables, and a $\psi$-compatible assignment $\sigma$ to the ordinary variables.

We write $s, \sigma \models p$ (resp. $\pi_i, \sigma \models p$) to mean that a state formula $p$ (resp. path formula $p$) is true at a total_state $s$ (resp. along a $\pi_i$ ) of the computation forest under an assignment $\sigma$ to the ordinary variables. We then define $\models$ inductively as follows:

- $s, \sigma \models t_1 = t_2$ iff $Val_{s \cup \sigma}(t_1) = Val_{s \cup \sigma}(t_2)$.
- $s, \sigma \models !p$ iff it is not the case that $s, \sigma \models p$.
- $s, \sigma \models p \ \& \ q$ iff $s, \sigma \models p$ and $s, \sigma \models q$.
- $s, \sigma \models p \mid q$ iff $s, \sigma \models p$ or $s, \sigma \models q$.
- $s, \sigma \models p->q$ iff $s, \sigma \models !p$ or $s, \sigma \models q$.

- $s$, $\sigma \models$ **LET** $(v = t)$ **IN** $p$ iff $s$, $\sigma' \models p$ where $\sigma' = (\sigma \setminus \{(v, \sigma(v))\}) \cup \{(v, Val_{s \cup \sigma}(t))\}$.

- $s_i$ , $\sigma \models \mathbf{A}p$ iff $\pi_i$, $\sigma \models p$ for every path $\pi_i = (s_i, s_{i+1}, \ldots, )$ in the computation forest.

- $s_i$ , $\sigma \models \mathbf{E}p$ iff $\pi_i$, $\sigma \models p$ for some path $\pi_i = (s_i, s_{i+1}, \ldots, )$ in the computation forest.

- $\pi_i$, $\sigma \models p$ where $p$ is a state formula, iff $s_i$ , $\sigma \models p$.

- $\pi_i$, $\sigma \models !p$ iff it is not the case that $\pi_i$, $\sigma \models p$.

- $\pi_i$, $\sigma \models p\&q$ iff $\pi_i$, $\sigma \models p$ and $\pi_i$, $\sigma \models q$.

- $\pi_i$, $\sigma \models p|q$ iff $\pi_i$, $\sigma \models p$ or $\pi_i$, $\sigma \models q$.

- $\pi_i$, $\sigma \models p->q$ iff $\pi_i$, $\sigma \models !p$ or $\pi_i$, $\sigma \models q$.

- $\pi_i$, $\sigma \models \mathbf{X}p$ iff $\pi_{i+1}$ , $\sigma \models p$.

- $\pi_i$, $\sigma \models \mathbf{G}p$ iff $\pi_j$ , $\sigma \models p$ for all $j \geq i$.

- $\pi_i$, $\sigma \models \mathbf{F}p$ iff $\pi_j$ , $\sigma \models p$ for some $j \geq i$.

- $\pi_i$, $\sigma \models p\mathbf{U}q$ iff for some $k \geq i$, $\pi_j$ , $\sigma \models q$, and $\pi_k$, $\sigma \models p$ for all $j$ $(i \leq j < k)$.

- $\pi_i$, $\sigma \models$ **LET** $(v = t)$ **IN** $p$ iff $\pi_i$, $\sigma' \models p$, where $\sigma' = (\sigma \setminus \{(v, \sigma(v))\}) \cup \{(v, Val_{si \cup \sigma}(t))\}$.

## Summary

In this chapter, we defined the syntax and the semantics of Abstract_CTL*, which is a very general first-order branching time temporal logic. This logic can be used to specify properties for a system described using ASM computation model. In the next chapter, we will define a subset of Abstract_CTL*, for which we have been able to develop property checking procedures.

# 5 Specification Language $L_{\mathrm{MDG}}$: a subset of Abstract_CTL*

Similar to CTL* which is based on propositional logic, Abstract_CTL* subsumes both linear time temporal and branching time temporal logics at the first-order logic level. As the model checking problem for CTL* was shown to be PSPACE-complete [38], we expect that the complexity of model checking for Abstract_CTL* would not be less than that.

Below we define a property specification language $L_{\mathrm{MDG}}$ [78] which is a subset of Abstract_CTL*. Our verification system can verify properties expressed in $L_{\mathrm{MDG}}$.

The *basic formulas of* $L_{MDG}$ are equations $t_1 = t_2$, where $t_1$ is an ASM_variable, $t_2$ is an ASM_variable, or an ordinary variable, or a constant. The *Next_let_formulas* are defined as follows:

(1) each basic formula of $L_{\mathrm{MDG}}$ is a Next_let_formula;

(2) if $p$, $q$ are Next_let_formulas, then so are: $!p$ (not $p$), $p\&q$ ($p$ and $q$), $p|q$ ($p$ or $q$), $p \text{->} q$ ($p$ implies $q$), **X***p*, **LET** *Let_equation* **IN** $p$.

In the Next_let_foumulas, we allow finite depth of nesting of the "next-time" temporal operator. In this sense, it is similar to the Symbolic Trajectory formulas [72].

## 5.1 Syntax of $L_{\mathrm{MDG}}$

We give the syntax of $L_{\mathrm{MDG}}$ in BNF. A *terminal symbol* is written in bold style, a *nonterminal symbol* is written in regular style starting with an upper case letter. Square brackets denote options. The start symbol is Property_file.

Property_file ::=

Property;

Property ::=

**A** ( Next_let_formula )

| **AG** ( Next_let_formula )

| **AF** ( Next_let_formula )

| **A** ( Next_let_formula ) **U** ( Next_let_formula )

| **AG** ( ( Next_Let_formula ) => (**F** ( Next_let_formula ) ) )

| **AG** ( ( Next_let_formula ) => (( Next_let_formula ) **U** ( Next_let_formula )) )

Next_let_formula ::=

**X** Next_let_formula

| **LET** ( Let_equation ) **IN** ( Next_let_formula )

| Next_let_formula **->** Next_let_formula

(Note: the first Next_let_formula can only contain concrete variables)

| Next_let_formula **&** Next_let_formula

| Next_let_formula | Next_let_formula

| **!** Next_let_formula

(Note: the Next_let_formula can only contain concrete variables)

| ( Next_let_formula )

| Basic_formula

Basic_formula ::=

Lterm = Rterm | *True* | *False*

(Note: *True, False* are Boolean constants)

Lterm ::= ASM_variable_Name

Rterm ::=

ASM_variable_Name

| OrdVar_Name

| IntegerConstant

| SymbolicConstant

| Function (only applies in a Next_let_formula prefixed by Let_equation)

Let_equation ::=

Let_equation & Let_equation

| ( Let_equation )

| OrdVar_Name = ASM_variable_Name

Function ::= Function_Name ( Parameter_List)

Parameter_List ::= Parameter | Parameter_List , Parameter

Parameter ::= OrdVar_Name | Function

ASM_variable_Name ::= [a-bd-eg-uw-z][A-Za-z0-9_]*

OrdVar_Name ::= [v][A-Za-z0-9_]*

Function_Name ::= [f][A-Za-z0-9_]*

IntegerConstant ::= [0-9]*

SymbolicConstant ::= [c][A-Za-z0-9_]*

Though only limited nesting of temporal operators is allowed in the $L_{MDG}$ syntax, additional formula templates are actually covered based on the following equivalences [37]. Since both path quantifiers can be prefixed to the following formulas, these equivalences are valid on both linear and branching time models.

$\models \mathbf{FF}p \equiv \mathbf{F}p$

$\models \mathbf{GG}p \equiv \mathbf{G}p$

$\models (\mathbf{F}p \vee \mathbf{F}q) \equiv \mathbf{F}(p \vee q)$

$\models (\mathbf{G}p \wedge \mathbf{G}q) \equiv \mathbf{G}(p \wedge q)$

$\models \mathbf{XF}p \equiv \mathbf{FX}p$

$\models \mathbf{XG}p \equiv \mathbf{GX}p$

$\models \mathbf{X}(p\mathbf{U}q) \equiv (\mathbf{X}p)\mathbf{U}(\mathbf{X}q)$

$\models \mathbf{X}(p \vee q) \equiv (\mathbf{X}p \vee \mathbf{X}q)$

$\models \mathbf{X}(p \wedge q) \equiv (\mathbf{X}p \wedge \mathbf{X}q)$

In $L_{\mathrm{MDG}}$, the existential path quantifier **E** is not allowed in the language. Given a property in $L_{\mathrm{MDG}}$ regarding an ASM under a given interpretation $\psi$, the property holds on the ASM iff the property is true for all paths starting from each of the initial total_states; i.e., the property is true for all paths in the abstract computation tree.

## 5.2 Semantics of $L_{\mathrm{MDG}}$

Since $L_{\mathrm{MDG}}$ is a subset of Abstract_CTL*, the semantics of Abstract_CTL* applies.

## 5.3 Examples of properties in $L_{MDG}$

An important verification task in designing correct sequential circuits is the checking of *safety* and *liveness* properties. Intuitively, a safety property asserts that "nothing bad happens". More precisely, a safety property defines a prefix closed language [16] [75]: any finite prefix $\beta$ of a sequence of states $\sigma$ that satisfies the safety property $P$ also satisfies the property:

$$\sigma \models P \Leftrightarrow (\forall\ \beta \mid \beta \leq \sigma \text{ and } |\beta| < \infty\ ,\ \beta \models P)$$

Below we give some examples of properties specified in $L_{MDG}$. Safety properties can be expressed using a Next_let_formula prefixed by "AG", as shown in Examples 1 to 4.

Example 1: A traffic light will never show red *(red* = 1*)* and green *(green* = 1*)* at the same time:

**AG**( ! ((*red* = 1) & (*green* = 1)));

Example 2: If there is a request *(req* =1*)* at any time, then an acknowledgment *(ack*=1*)* should be generated 3 transitions later:

**AG**((*req* = 1) -> **X**(**X**(**X**(*ack* = 1))) );

Example 3: Whenever a pedestrian presses the push button of the traffic light *(request=1)*, he/she will receive green light *(pgreen = 1)* within 3 clock cycles:

**AG**( (*request* = 1) -> (**X**(*pgreen* = 1) | **X**(**X**(*pgreen* = 1)) | **X**(**X**(**X**(*pgreen* = 1)))) );

Example 4: If there is a request *(req* =1*)* at time t, then the data at in port *Din* at time t will show up at out port *Dout* at t+1:

**AG**( (*req* =1) -> **LET** (*v* = *Din*) **IN** (**X** (*Dout* = *v*)) );

Sometimes, we only want to know that something happens in a fixed amount of time after the initial state of the transition system. Example 5 illustrates such a situation.

Example 5 : A green light *(green* =1*)* always shows up in 3 cycles after the initial state:

**A**(**X**(**X**(**X**(*green* =1))));

The liveness properties are referred to as "eventuality" properties or "progress' properties. Roughly speaking, a liveness property asserts that "something good will happen", related to an unbounded but finite temporal interval. Liveness properties are necessary for expressing that a system, after having received a particular input sequence, will produce some outputs in a finite amount of time, but the exact amount is not known.

Example 6 : If there is a request *(req* =1*)* at time t, then an acknowledgment *(ack*=1*)* will be eventually generated.

**AG**((*req*=1) => (**F**(*ack*=1)) );

# Summary

In this chapter, we defined $L_{MDG}$, a subset of Abstract_CTL*, as the property specification language for MDG-based model checker. Both safety and liveness properties can be expressed in $L_{MDG}$. We also gave some examples of properties specified in $L_{MDG}$. In the next chapter, we will present in detail the property checking algorithms.

# 6 Model Checking for Properties in $L_{\mathrm{MDG}}$

## 6.1 Introduction

In general, our approach to model checking is to automatically build additional ASMs that represent the Next_let_formulas appearing in the property to be verified, connect these additional ASMs to the original, and then check a simpler property on the composite machine [78].

Given a Next_let_formula $P$ regarding an ASM $D = (X, Y, Z, F_I, F_T, F_O)$, an ASM $D_p = (Xp, Yp, Zp, F_{Ip}, F_{Tp}, F_{Op})$ can be constructed to represent the Next_let_formula. The input variables of $D_p$ are the ASM_variables of $D$ which appear in the property, i.e., $Xp \subseteq X \cup Y \cup Z$. They represent the values at the "current" cycle. Let $n$ be the maximum nesting number of **X** operators in the property, the set of state variables $Yp$ and the transition relation $F_{Tp}$ are constructed so as to "remember" the values of input variables of $D_p$ or the results of comparison of the variables in the past $n$ (or less than $n$) cycles. The set of the

state variables of $D_p$ contains a special state variable of Boolean type, *Flag*, which indicates the truth of the Next_let_formula one cycle earlier. The initial set of states $F_{Ip}$ are assigned differently depending on which property template the Next_let_formula $P$ corresponds to. The general idea is that the initial states of $D_p$ should not affect the result of verifying $P$ on the original ASM $D$. There is no output from $D_p$, i.e., $Zp$ is empty. Hence, there is no output relation either. The details of an algorithm for constructing an ASM representing a Next_let_formula are given in Chapter 7. Figure 9 shows how the composite machine is connected.

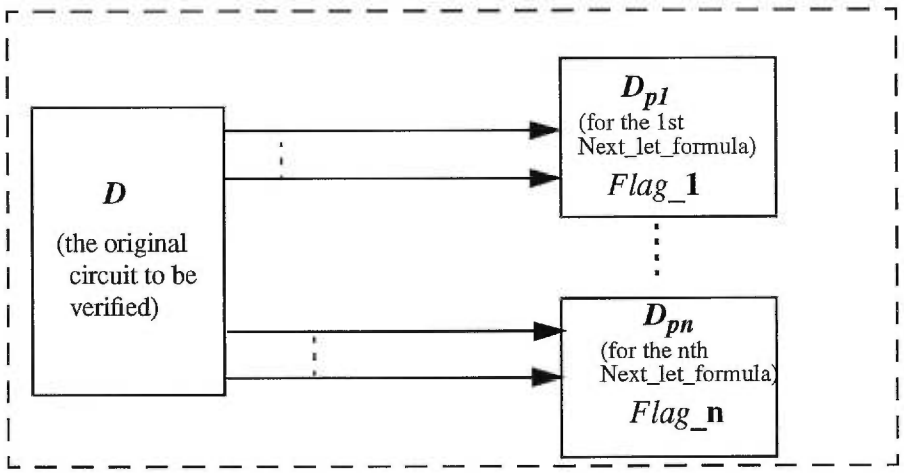



**Figure 9** - Connection of the ASMs $D, D_{p1}, \ldots, D_{pn}$ for property checking

## 6.2 Model checking algorithms

In the following subsections, we describe algorithms for verifying the various forms of the formulas in $L_{\text{MDG}}$. When our property checking algorithms report success to a query, then

the property holds for an ASM under any interpretation. It is possible that a property holds for the ASM under the intended interpretation of the abstract function symbols and constants, but not under every interpretation. In that case, we can obtain a false negative answer with respect to the original, non-abstracted problem. However, if all the data operations are viewed as black boxes, a property is expected to hold for every interpretation; it is in this sense that we say that our algorithms are applicable to designs where data operations are viewed as black boxes.

### 6.2.1 AG(Next_let_formula)

To check a property of the form **AG**(Next_let_formula) on an ASM $D = (X, Y, Z, F_I, F_T, F_O)$, we first construct an ASM $D_p = (Xp, Yp, Zp, F_{Ip}, F_{Tp}, F_{Op})$ to represent the Next_let_formula. Second, we construct a composite machine from $D$ and $D_p$, that is $M = (X_m, Y_m, Z_m, G_I, G_T, G_O)$, where

- $X_m = X$ is the set of the input variables of $D$;
- $Y_m = Y \cup Yp$ is a set of the state variables, containing both the variables in $Y$ and $Yp$; However, since $M$ is a composite machine (the states of $D_p$ are derived from $D$) rather than the product machine of $D$ and $D_p$, under each interpretation $\psi$, the state space of $M$ is actually a subset of $\Phi_Y^{\psi} \times \Phi_{Y_P}^{\psi}$;
- $Z_m = Z$ is the set of the output variables of $D$;
- $G_I = F_I \wedge F_{Ip}$ is a DF of type $U \rightarrow Y_m$ representing the set of initial states of $M$;
- $G_T = F_T \wedge F_{Tp}$ is the abstract description of the transition relation of $M$;

- $G_O = F_O$ is the abstract description of the output relation of $M$.

In addition, η' is the function that maps each state variable of $M$ to the corresponding next-state variables.

Finally, we do reachability analysis on the composite machine $M$ and check "*Flag* = 1" at each reachable state.

For example, to check the property **AG***(req=1* -> **LET** *(v =Din)* **IN** *(***X** *(Dout = v)))* on an ASM $D$, we build a composite ASM as shown in Figure 13 in Chapter 7, and then check **AG***(Flag* =1*)* on that machine.

The algorithm to check a property in the form of **AG**(*Flag*=1) is as follows:

```
(1)   Check_AG(M, C)
      /* M is the composite machine, G_I is the set of initial states of M, */
      /* G_T is the transition relation of M */
      /* C is the DF containing Flag =1. */
(2)    begin
(3)   R := G_I; Q := G_I; K := 0;
(4)    loop
(5)   P := PbyS(Q, C);
(6)   if P ≠ F then return failure;/* if the property is not satisfied, report failure */
(7)   K := K + 1;
(8)   I := Fresh(X_m, K);                          /*generate input values */
(9)   N := RelP({I, Q, G_T}, X_m ∪ Y_m, η' );      /* compute next states */
(10)  Q := PbyS(N, R);                             /* compute frontier set of states */
(11)  if Q = F then return success;                /* if fixpoint reached, report success */
(12)  R := PbyS(R, Q);          /* simplify R by removing states subsumed by Q */
(13)  R := Disj(R, Q);                             /* compute all states reached so far */
(14)   end loop;
(15)   end;
```

## 6.2.2 A(Next_let_formula)

To check a property in the form of **A**(Next_let_formula), we need to construct a composite ASM in the same way as we treated the property **AG**(Next_let_formula) and then transfer the problem to checking **A**(*Flag* = 1*)* on the composite machine. However, here we only need to check *Flag* =1 on the states reached in $n$+1 transitions from the initial state, where $n$ is the maximum nesting depth of the **X** operators in the property. The additional 1 cycle delay is caused by the fact that *Flag* is a state variable. For example, to check a property **A**(**X**($p$=1) ∣ **XX**($q$=1)), the maximum nesting depth of the **X** operators is 2. The circuit representing the composite ASM is shown in Figure 10. We then check (*Flag* =1) in the states reached by 3 transitions on the composite machine.

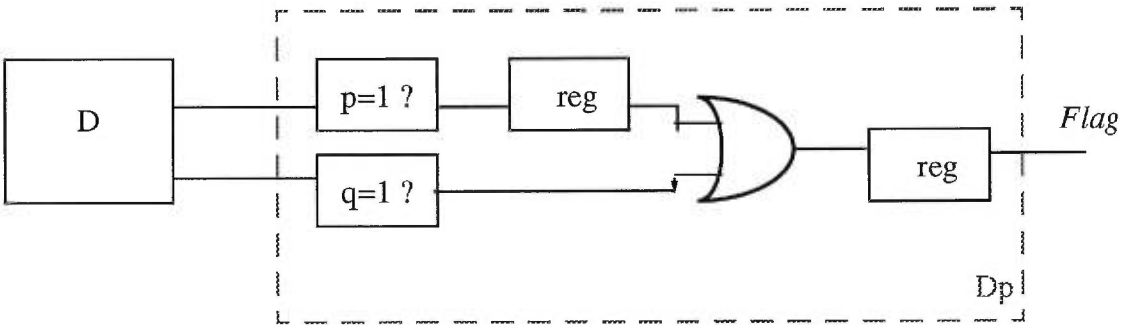



**Figure 10** - The composite machine for **A**(**X** $p$=1 ∣ **XX** $q$=1)

The algorithm to check a property in the form of **A**(*Flag* =1) on a composite machine with the maximum nesting depth of **X** in the original property is as follows. Intuitively, it runs upto depth $n$ and then check if *Flag* =1 is true on all the states.

(1) **Check_A**($M$, $n$, $C$)
/* $M$ is the composite machine, $G_I$ is the set of initial states of $M$, */
/* $G_T$ is the transition relation of $M$, */
/* $n$ is the maximum nesting depth of **X** in the property. */
/* $C$ is the DF containing *Flag*=1. */
(2) begin

(3) $Q := G_I$;
(4) for $K = 1$ to $n$+1 loop
(5) $I :=$ **Fresh**($X_m$, $K$); /* generate input values */
(6) $N :=$ **RelP**({$I$, $Q$, $G_T$}, $X_m \cup Y_m$, η' );/* compute next states */
(7) $Q := N$;
(8) end loop;
(9) $P :=$ **PbyS**($Q$, $C$);
(10) if $P =$ **F** then return success;
/* check if *Flag=1* is satisfied, if not report failure */
(11) else return failure;
(12) end;

## 6.2.3 AF(Next_let_formula)

We will also use an additional ASM to represent the Next_let_formula, build the composite machine $M$, and then verify that the property **AF**(*Flag* = 1) holds at all the initial states of the composite machine $M$. The algorithm to check **AF**(*Flag* = 1) is as follows:

(1) **Check_AF**($M$, $C$)
/* $M$ is the composite machine, */
/* $G_I$ is set of initial states of $M$, */
/* $G_T$ is the transition relation of $M$, */
/* $C$ is the DF containing *Flag*=1. */
(2) begin
(3) $\Sigma := \varnothing$;
/* Σ is a set containing DFs representing each a set of states not satisfying *Flag*=1 after a transition step */
(4) $P := G_I$;
(5) K := 0;
(6) loop
(7) $Q :=$ **PbyS**($P$, $C$); /* remove states satisfying *Flag*=1 */
(8) if $Q =$ **F** then return success;

(9) if $\exists\, T \in \Sigma$, **PbyS**($T$, $Q$) = **F** return failure;
/*This step checks if DF $Q$ covers any one of the DFs in $\Sigma$, i.e., for each DF $T$ in $\Sigma$, **PbyS**($T$, $Q$) = **F** is checked to detect a cycle over the states that do not satisfy *Flag*=1. If there is a cycle, then failure is reported*/
(10) $\Sigma := \Sigma \cup \{Q\}$; /* add DF $Q$ as an element into $\Sigma$ */
(11) K := K+1;
(12) $I$ := **Fresh**($X_m$, $K$); /* generate input values */
(13) $P$:= **RelP**($\{I, Q, G_T\}$, $X_m \cup Y_m$, $\eta'$ ); /* compute next states */
(14) end loop
(15) end

The algorithm removes the states satisfying *Flag* = 1 from the reached set of states at each transition step and keeps the set of states that do not satisfy *Flag* = 1 as the frontier set for the next-state computation. At each transition step, the frontier set is recorded as an element of $\Sigma$. If an empty frontier set is reached, then the algorithm succeeds. If the whole frontier set covers any set in $\Sigma$, it means that there is at least one cycle and the states in the cycle do not satisfy *Flag*=1 (see the proof in Section 10), i.e., there is at least one infinite path on which **F**(*Flag*=1) does not hold, then the algorithm stops and reports failure.

Figure 11 shows an example of checking **AF***(Flag=1)*. The labels 1, 2, ..., 6 represent the reachable states, and only state 6 satisfies *Flag=1*. The computation stops by reporting failure when it detects that Set($Q$) covers a set in $\Sigma$ . We can also see that one cycle exists (3 -> 5 -> 3), and *Flag=1* is not satisfied in the states along the cycle.

Furthermore, to check **AG**($c$ => (**F** $p$)) on ASM $D$, where $c$ and $p$ are Next_let_formulas, we can build a composite machine $M$ from $D$, an ASM of $c$, and an ASM of $p$, and then verify **AG**((*FlagC*=1) => (F (*FlagP*=1))) on $M$. First, we do reachability analysis to get all the reachable states of $M$ (represented by $W$), then we collect from $W$ the states satisfying "*FlagC* = 1" ($V$ := **Conj**($W$, $C_c$) where $C_c$ is a DF containing *FlagC* = 1), and finally we apply the algorithm **Check_AF** with the set $V$ as the set of initial states.

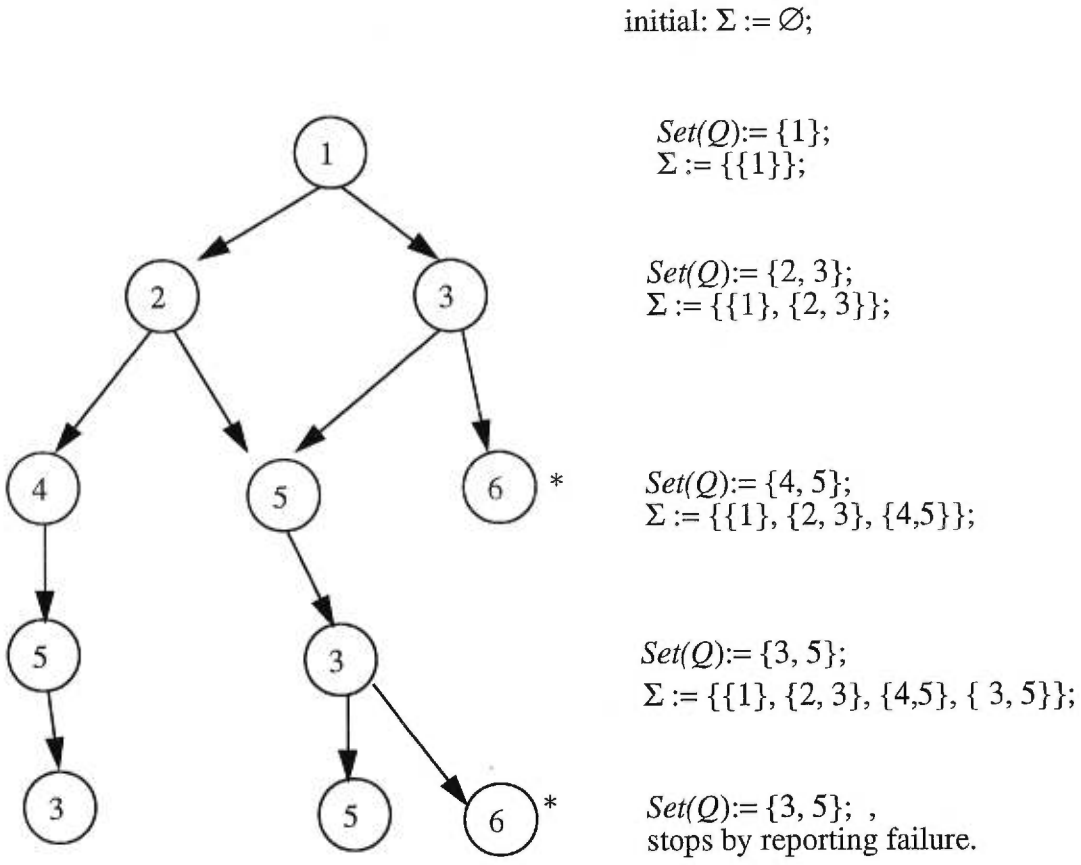



**Figure 11 -** An example of checking **AF***(Flag=1)*.

A CTL formula **AGAF***p* is a special case of **AG** ( *c* => (**F** *p*) ) in which *c* is the Boolean constant *True*.

## 6.2.4 A(Next_let_formula)U(Next_let_formula)

We use additional ASMs to represent the Next_let_formulas and then transfer the problem to checking **A**(*FlagP*=1)**U**(*FlagQ*=1) on the composite machine.

(1) **Check_AU**($M$, $C_p$, $C_q$)
/*M is the composite machine */
/*$G_I$ is the set of initial states of $M$ */
/* $G_T$ is the transition relation of $M$, */
/* $C_p$ is the DF containing $FlagP = 1$. $C_q$ is the DF containing $FlagQ = 1$*/
(2) begin
(3) $\Sigma := \varnothing$;
/* $\Sigma$ is a set containing DFs representing each the set of states satisfying $FlagP = 1$ but not $FlagQ = 1$*/
(4) $P := G_I$;
(5) K := 0;
(6) loop
(7) $Q :=$ **PbyS** ($P$, $C_q$ ); /*remove from S states with $FlagQ = 1$*/
(8) if $Q =$ **F** return success;
(9) if $\exists\, T \in \Sigma$, **PbyS**($T$, $Q$) = **F** return failure;
/*This step checks if DF $Q$ covers any one of the DFs in $\Sigma$, i.e., for each DF $T$ in $\Sigma$, **PbyS**($T$, $Q$) = **F** is checked to detect a cycle in which $FlagP = 1$ is true but $FlagQ = 1$ never becomes true. If there is a cycle, then failure is reported*/
(10) $R =$ **PbyS** ($Q$, $C_p$); /*remove from $Q$ states with $FlagP = 1$*/
(11) if $R \neq$ **F** return failure;
(12) $\Sigma := \Sigma \cup \{Q\}$; /* add DF $Q$ as an element into $\Sigma$ */
(13) K := K+1;
(14) $I :=$ **Fresh**($X_m$, $K$); /* generate input values */
(15) $P$:= **RelP**($\{I, Q, G_T\}$, $X_m \cup Y_m$, $\eta'$ ); /* compute next states */
(16) end loop;
(17) end;

The above algorithm removes from the reached set of states those states satisfying $FlagQ$=1. If the leftover Set($Q$) is empty, then the algorithm stops by reporting success. Otherwise, if there is at least one cycle where states keep satisfying $FlagP$=1, i.e., $FlagQ$=1 never becomes true, then there is at least one path starting from the initial state where $p\mathbf{U}q$ does not hold, it stops and reports failure. Otherwise, it checks whether all the states in Set($Q$) satisfy $FlagP$=1. If there are some states where $FlagP$=1 does not hold, which means that there are some path(s) on which $FlagP$=1 does not hold in every state before a state satisfying $FlagQ$=1 is reached, then it also stops and reports failure.

Otherwise, it computes the next states reachable from Set($Q$) and repeats the process.

To check property **AG**($c$ => $p$**U**$q$ ) on machine $D$, we need to build a composite machine $M$ from $D$, an ASM representing c, an ASM representing p, and an ASM representing q, and transfer to checking the property **AG**(($FlagC$=1) => (($FlagP$=1)**U**($FlagQ$=1))) on M. We then do reachability analysis to obtain all the reachable states of $M$ (represented by $W$), collect from $W$ the states satisfying "$FlagC$ = 1" ($V$ := **Conj**($W$, $C_c$) where $C_c$ is a DF containing $FlagC$ = 1), and finally apply the algorithm **Check_AU** with the set $V$ as the set of initial states.

A CTL formula **AGA**$p$**U**$q$ is a special case of **AG**($c$ => $p$**U**$q$) in which $c$ is the Boolean constant *True*.

# Summary

In this chapter, we described property checking algorithms for formulas in the form of **AG**$p$, **A**$p$, **AF**$p$, and **A**$p$**U**$q$, where $p$, $q$ are Next_let_formulas. All the property checking algorithms are based on the "forward" image computation. That is, at each iteration, we check whether the property holds on all the current states. If yes, then the reachable states by one transition step from the current ones are computed and the property checking continues, otherwise the property checking stops by reporting failure. These property checking algorithms are the basic algorithms in our verification system. They can be used to check more complicated properties, such as **AG** ($c$ => (**F** $p$)) and **AG**($c$ => $p$**U**$q$).

Another point worth mentioning is that even though we do not allow the existential path quantifier **E** in $L_{MDG}$, the above algorithms can be used to verify properties involving

**E** in situations where there is no abstract variable used in the description of the design being verified. For example, to verify **EF***p*, we can transform the problem to checking **AG**(!*p*), and then reverse the verification result.

# 7 Construction of an ASM from a Next_let_formula

In this chapter, we give a procedure for generating a circuit description representing a Next_let_formula *p*. The descriptions are produced in MDG-HDL [80], which is a hardware description language at the register transfer (RT) level. It allows the use of abstract variables for representing data. The ASM model of the Next_let_formula is automatically produced from the circuit description by the parser in the MDG verification package.

The procedure to generate a circuit description for a Next_let_formula is as follows.

**Step1: Build a parsing tree for *p***

1) If *p* is in one of the following templates: **AF***p*, A*q***U***p*, **AG**(*c* => (**F***p*)), or **AG**(*c* => (*q***U***p*)), then rewrite *p* to *p'* = *True* & *p*; (*True* and *False* are two Boolean constants in this procedure.)

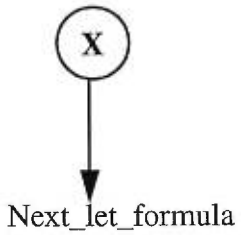
   If *p* is in one of the following templates: **AG***p*, A*p***U***q*, rewrite *p* to *p'* = *False* | *p*;

If $p$ is in $\mathbf{A}p$, then let $p' = p$.

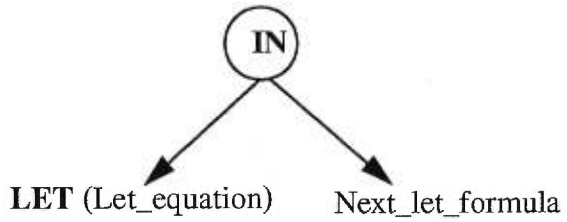
This is a preliminary step. When we construct the property circuit in Step 3, we put certain registers connected to a constant signal *True* or *False*, with the initial values *False* or *True* respectively, to make sure that the property is checked after $n$ cycles from the initial state where $n$ is the maximum nesting depth of the **X** operators in the property.

2) A *parsing tree* is constructed to represent $p'$; The *leaves* of the parsing tree are the constants *True*, *False*, the atomic formulas, and the Let_equations; the *root* and the internal nodes of the parsing tree can be **X**, **IN**, **&**, **|**, **!**, **->**. Recursively, a Next_let_formula is transformed into a parsing tree according to the following rules:

- " **X** ( Next_let_formula) " is transformed to



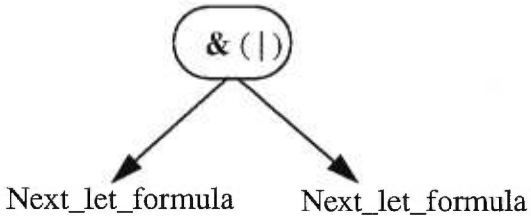

- " **LET** ( Let_equation ) **IN** ( Next_let_formula ) " is transformed to



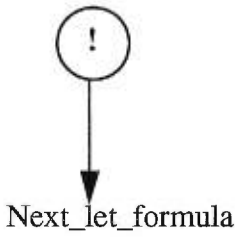

- " ( Next_let_formula) ->( Next_let_formula) " is transformed to " !(Next_let_formula) | (Next_let_formula) " first and then treated as " | " ( or ) operation.

The constant *False* replaces **!**(*True*);

- " ( Next_let_formula ) **&** ( Next_let_formula) ", and " ( Next_let_formula) | (Next_let_formula ) " are transformed to




- " **!** ( Next_let_formula ) " is transformed to




The *brother* of a node *v* (represented as *Brother(v)* ) in a parsing tree is defined as the node that shares the same parent node with *v*. A node may have a brother which is an empty node.

**Step2: Assign attributes to each node of the parsing tree**

The attributes include the *ID number*, the *depth of the X operators*, the leaf node flag, and the root node flag.

When the parsing tree is traversed in depth-first postorder, each node is assigned an

order number when visited. This order number is referred to as the *ID number* of the node.

The *depth of the X operators* of a node $v$ (represented as *DepthX(v)*) is defined recursively as follows:

- 0 if the node is a leaf node or the node is empty; and
- *max (DepthX(v1), DepthX(v2))* if the node is not an X node and *v1*, *v2* are the left and right children of the node; and
- 1+ *max (DepthX(v1), DepthX(v2))* if the node is an X node and *v1*, v2 are the left and right children of the node;

If a node is a leaf node (root node), then that node is marked with a leaf node flag (a root node flag).

In addition, a list containing all the leaf nodes is generated.

**Step 3: Construct the circuit**

Traverse the attributed parsing tree using depth-first postorder:

- If node $v$ contains an atomic formula which is an equation $t_1 = t_2$, then a comparator is inserted. If $t_1$, $t_2$ are of concrete sort, then a table is used to describe the function of the comparator by enumerating all the values that $t_1$ and $t_2$ can take. For example, if $t_1$ and $t_2$ are of Boolean sort, then the truth table describing the function of the comparator in MDG_HDL [80] is as follows:

*component(comp_Comparator, table([[$t_1$,$t_2$, result],*

*[0,0,1],*

*[1,1,1],*

*[0,1,0],*

*[1,0,0]])).*

If $t_1$, $t_2$ are of abstract sort, then a partially interpreted cross-operator *AbsComparator* (abstract comparator) is used to denote the truth of $t_1 = t_2$ in the current state of the circuit. The following rewriting rule provides a minimum interpretation of the comparator: AbsComparator(X, X) = 1, which can be interpreted as " the values of the 2 abstract terms are the same if the 2 terms are syntactically the same". This rewriting rule is always used before any other rewriting rules are applied.

In the process of building the circuit, the inputs of the table or the cross_operator are the variables appearing in the equation. The output of the table or the cross_operator is given a new variable name. (The variables generated by the property circuit are named as *addedSignal_1*, *addedSignal_2*, ..., *addedSignal_N*). Then, *DepthX(Brother(v))* is looked up. Let $m$ = *DepthX(Brother(v) ) - DepthX(v)*, if $m > 0$ then $m$ registers are added. The input of the first register is the output of the concrete or the abstract comparator discussed above. The output of the last register is referred to as the output of node $v$. The $m$ registers are initialized to the Boolean constant *True*.

- If $v$ contains an atomic formula and there is an ordinary variable in the formula, then we search for an "IN" node from $v$ up. The number of X nodes (*LetXNum*) along the path is counted. When an 'IN" node is reached, we look up its left child node (which contains a Let_ equation) to find one equation that contains the same ordinary variable. If it fails to find it, we keep searching up the tree until the ordinary variable is found in a Let_equation. *LetXNum* registers (state variables that take the sort of the ASM_variable to which the ordinary variable refers) are now added, such that the input of the 1st register is the variable that the ordinary variable refers to, the output of the last register use the ordinary

variable name. The initial values of those added state variables have no affect on the property verification result (see proof in Chapter 9). If the atomic formula v contains has an abstract variable, then each of the added state variables is initialized to a fresh variable which could take any value in its domain; If the atomic formula has a concrete variable, then each of those added registers is initialized to the first value in its sort (this is what has been implemented, actually, those added registers can be initialized to any value in their sorts). A concrete or an abstract comparator is then constructed, with the inputs being the variables in the atomic formula and the output being a new variable. Again, Let $m$= *DepthX(Brother(v) )* - *DepthX(v)*, if $m > 0$ then $m$ registers are added at the output of the comparator. The input of the 1st register is the output of the table or the crossterm AbsComparator. The output of the last register is referred to as the output of node v. The $m$ registers are initialized to *True*.

- If $v$ contains "*False*", and $m > 0$ where $m$ = *DepthX(Brother(v) )* - *DepthX(v)* , then $m$ registers are added, the input of the 1st register is a constant *False*, the output of the last register has the name of the output of node $v$. The $m$ registers are initialized to *True*. If a node $v$ contains "*False*", and $m \leq 0$, then a signal connecting to the Boolean constant *False* is the output of node $v$.

- If $v$ is "|" or "&", then an "OR" or "AND" gate is added, respectively. The inputs of the gate are the outputs of the two child nodes and the output is given a new name. Again, let $m$= *DepthX(Brother(v) )* - *DepthX(v)*, if $m > 0$ then $m$ registers are added at the output of the gate. The input of the 1st register is the output of the gate. The $m$ registers are initialized to *True*. The output of the last register has the name of the output of node $v$. If $m \leq 0$, the output of the gate is the output of node $v$.

- If $v$ contains "!", then a "NOT" gate is added. The input of the gate is the output of its child node. The output is given a new variable name. $m$ registers are added at the output of the gate if $m$ = *DepthX(Brother(v) )* - *DepthX(v)* and $m > 0$. The

input of the 1st register is the output of the gate. The output of the last register has the name of the output of node *v*. The *m* registers are initialized to *True*.

- If *v* contains a Let_equation, then nothing is done on this node. This node is used when a node is visited which contains an atomic formula referring to the ordinary variables in the Let_equation.

- If *v* contains "X" and $m > 0$ where $m$= *DepthX(Brother(v) ) - DepthX(v)* , then *m* registers are added. The input of the 1st register is the output of the child of the "X" node. The output of the last register has the name of the output of the "X" node. The *m* registers are initialized to *True*. If a node contains "X" and $m \leq 0$, then the output of its child node is directly referred to as the output of node "X".

- If *v* contains "IN" and $m > 0$ where $m$= *DepthX(Brother(v) ) - DepthX(v)* , then *m* registers are added. The input of the 1st register is the output of the right child of the "IN" node. The output of the last register has the name of the output of the "IN" node. The *m* registers are initialized to *True*. If a node contains "IN" and $m \leq 0$, then the output of right child of node "IN" is referred to as the output of node "IN".

- If *v* is the root node, then a register is added taking the output of *v* as the input, and the output is given the name *Flag*. If the Next_let_formula *p* is in one of the templates **AG***p*, **A***p*, **A***p***U**q, then the register is initialized to *True*. Otherwise, the register is initialized to *False*. The initial value of *Flag* is such that it does not affect the verification result.

The following is an example illustrating the construction of a circuit representing a Next_let_formula.

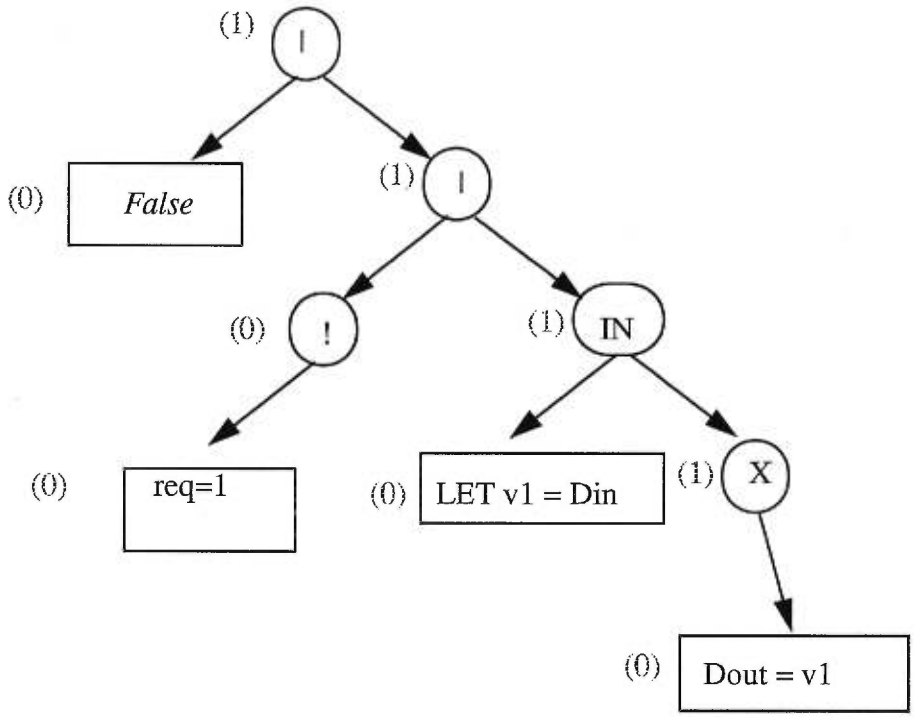
Given a property **AG***(req=1* -> **LET** *(v1=Din)* **IN** *(***X** *(Dout =v1)))*, where *req* is a Boolean variable, and *Din* and *Dout* are of abstract sort, the following steps are used to construct a circuit of the Next_let_formula *p = (req=1* -> **LET** *(v1=Din)* **IN** *(***X** *(Dout*

*=v1))):*

Step 1: The Next_let_formula *p* is transformed to

*False* | (!( *req* =1)) | **LET** (*v1* = *Din*) **IN** (**X** (*Dout* = *v1*))

The following parsing tree is then constructed (Figure 12):.




A square represents a leaf node.

**Figure 12 -** The parsing tree of **AG***(req=1* -> **LET** *(v1=Din)* **IN** *(***X** *(Dout =v1)))*

Step 2: *DepthX(v)* is associated with each node of the parsing tree. (It is indicated at the left-hand side of a node.)

Step 3: Construct the circuit:

The parsing tree with DepthX marked on each node is traversed in depth-first postorder:

1) The node "*False*" is visited first. Since $m = 1 - 0 = 1$, one register is added. The input of the register is the constant *False*, the output of the register corresponds to the output of this node. The initial value of the register is *True*.

   The MDG-HDL description [80] of a signal connecting to a Boolean constant *False* and the added register is as follows:

   *component(prop_comp_1, constant_signal(value(0), signal(addedSignal01))).*

   *component(prop_comp_2, reg(input(addedSignal01),output(addedSignal02))).*

   The output of node "*False*" is the signal *addedSignal02*.

2) The node "req=1" is visited. Since it is a leaf node and contains an atomic formula without any ordinary variable, the following truth table description of a comparator is constructed (described in MDG-HDL [80]):

   *component(prop_comp3, table([ [req,addedSignal03],*

   *[1,1]*

   */ 0])).*

   Since this node has no brother, no register is added at the output of the comparator.

   The output of node "req=1" is the signal *addedSignal03*.

3) The node "!" is visited. A "NOT" gate is added whose input is the output of node req=1. The MDG-HDL declaration is as follows:

*component(prop_comp4, not(input(addedSignal03), output(addedSignal04))).*

Since $m$ = *DepthX(Brother(v) )* - *DepthX(v)* = 1-0 = 1, one register is added at the output of the gate:

*component (prop_comp5, reg(input(addedSignal04), output(addedSignal05))).*

The output of node "!" is the signal *addedSignal05*;

4) The node "LET v1=Din" is visited. Since it contains a Let_equation, nothing is done at this moment.

5) The node "Dout = v1" is visited. Since it contains an ordinary variable v1, we need to find an "IN" node from the node up and count the number of "X" nodes along the path. From the left child of the "IN" node, we find the equation "LET v1 = Din". Since *LetXNum* = 1, one register is added. The input of the register is *Din*, and the output is *v1*. Then an abstract comparator AbsComp is inserted. The inputs of AbsComp are *Dout* and *v1*, the output is *addedSignal06*. The MDG-HDL description of the abstract comparator is as follows:

*component (prop_comp6, reg(input(Din), output(v1))).*

*component(prop_comp7, transform(inputs([Dout, v1]),*
*function(AbsComparator), output(addedSignal06))).*

Since the node "Dout=v1" has no brother, no register is added at the output of AbsComparator.

6) The node "X" is visited. Since $m < 0$ , no component is added, and the output of its child, i.e., *addedSignal06* is used as the output of the "X" node.

7) The node "IN" is visited. Again, no component is added since $m < 0$, but the

output of its right child, i.e., *addedSignal06* is used as output of the "IN" node.

8) The node "|" is visited. An "OR" gate is added. The inputs of the gate are the outputs of the nodes "!" and "IN", and the output is *addedSignal07*. The "OR" gate is described in MDG-HDL as follows:

*component(prop_comp8, or(input(addedSignal05, addedSignal06), output(addedSignal07))).*

9) Finally the root node "|" is reached. An "OR" gate is added. The inputs of the gate are the outputs of the nodes "*False*" and "|", and the output is *addedSignal08*. Since it is a root node, one register is added. The input of the register is *addedSignal08*, and the output is *Flag*. Since the property is in the template of **AG***p*, the initial value of *Flag* is assigned to *True*.

*component(prop_comp9, or(input(addedSignal02,addedSignal07), output(addedSignal08))).*

*component (prop_comp10, reg(input(addedSignal08), output(Flag))).*

The resulting circuit is shown in Figure 13.

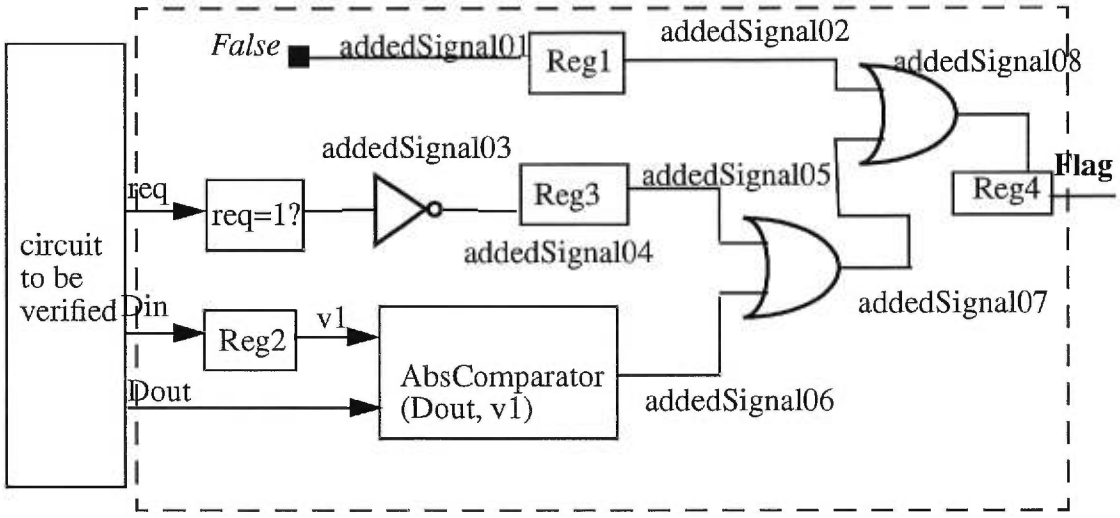



**Figure 13 -** The additional circuit for **AG***(req=1* -> **LET** *(v1=Din)* **IN** *(***X** *(Dout =v1)))*.

# Summary

In this chapter, we described the procedure of constructing an ASM from a Next_let_formula. Through this method, we were able to transform the verification of a more complex property on an ASM to the verification of a simplified property on a composite ASM built form the original ASM and the ASMs of the Next_let_formulas in the original property. The correctness of this problem transformation shall be verified in Chapter 9.

In the next chapter, we will proceed with the definition of fairness constraints in our verification system and the verification of liveness properties under fairness constraints.

# 8 Verification of Liveness Properties with Fairness Constraints

When we verify liveness properties, we are usually only interested in *fair* computation paths. A fair path is a computation path along which certain states or certain combination of states are not sustained forever. For example, in a system where a shared memory resource is accessed by several devices, we are only interested in the computation paths along which no device occupies the shared memory resource forever.

## 8.1 Fairness constraints

In the literature, different methods for specifying fairness constraints have been developed for CTL model checking [38] and for language containment using L-automata [55].

For CTL model checking [15][61], Reference [38] contains model checking algorithms for the so-called Fair Computation Tree Logic (FCTL). An FCTL specification $(p_0, \Phi_0)$ consists of a functional assertion $p_0$ and an underlying fairness assumption $\Phi_0$. The functional assertion $p_0$ is expressed in essentially CTL syntax with basic modalities of

the form either **A** (for all fair paths), or **E** (for some fair path) followed by one of the linear time operators **G***p*, **F***p*, **X***p*, or *p***U***q*. All path quantifies are thus related to the underlying fairness assumption $\Phi_0$ specified in the canonical form $\vee_{i=1}^{n} \wedge_{j=1}^{m} (\mathbf{F}^{\infty} p_{ij} \vee \mathbf{G}^{\infty} q_{ij})$ where $p_{ij}$ and $q_{ij}$ are atomic propositions, and $\mathbf{F}^{\infty}$, $\mathbf{G}^{\infty}$ are two basic infinitary operators: $\mathbf{F}^{\infty} f$ (infinitely often *f*) holds for a computation path $\pi$ in the Kripke structure iff the CTL formula *f* holds for an infinite number of times along $\pi$; $\mathbf{G}^{\infty} g$ (almost everywhere *g*) holds for a path iff the CTL formula $\neg g$ holds for only a finite number of times along $\pi$, i.e., after a finite amount of time *g* holds forever along $\pi$.

Using language containment [55][56], to verify that a property is satisfied on a system model, one verifies that the language of the system model is contained in the language of the property. The system is modeled as L-processes, and the property is modeled as a L-automata. For both L-automata and L-processes (each of which defines an $\omega$-regular language), the acceptance condition is defined by a set of *recur* edges, and a set of *cycle sets* of states. A sequence of states is accepted by an L-automaton if and only if it has a run of the automaton which either traverses a given recur edge infinitely often or eventually enters in some cycle set and remains in the cycle set forever. For an L-process, the acceptance condition is the negation of that of an L-automaton: a sequence is accepted if and only if it is a run which traverses no recur edge infinitely often, and does not remain forever in some given cycle set. Thus, in the case of L-process, the acceptance condition can be understood as an "exception" condition. L-processes provide a natural mechanism to model a "system" process, as the exception condition can be interpreted as a "fairness" property, excepting "unfair" sequences which for example, never leave a set of states (a cycle set). L-automata provide natural properties, which are to be verified, i.e., a "liveness" property may be defined in terms of sequences which traverse a given set of recur edges infinitely often. For example, to check that a process is eventually granted access to a resource, provided that the process does not remain in a set of states where it never requests the resource, we can impose this fairness condition on an L-process *S* which models the whole system. We then express the liveness property in terms of an L-

automaton $T$ and verify that the language containment Language($S$) $\subset$ Language($T$) holds.

## 8.2 Our approach to imposing fairness constraints

In our verification system, we impose fairness constraints using a subset of the criteria employed in the method based on language containment, namely, by specifying cycle sets. Let $H_i$ , $i = 1, ..., n$, be $n$ "exception" conditions, and $S_\omega$ the set of infinitely repeating states along a computation path. If at least one $H_i$ holds on all states in $S_\omega$, then the computation path is not fair and need not satisfy the property under investigation. That is, only those computation paths along which the states satisfy every $!(H_i)$ infinitely often are considered. Therefore, $!(H_i)$ $(1 \leq i \leq n)$ can be viewed as the fairness constraints for the property checking. We call the formula representing the exception condition $H_i$ as an H_formula.

Next we give the syntax of language for specifying fairness constraints. The terminal symbols are written in bold style, and the nonterminal symbols are written in regular style starting with an upper case letter. Square brackets denote options. Note that only concrete ASM_variables may appear in the H_formulas. All the fairness constraints imposed are stored in a file, which is interpreted in before the model checking procedure is invoked.

Fairness_file ::=
    Fairness_constraint_list

Fairness_constraint_list::=
    Fairness_constraint;
    [Fairness_constraint_list]

Fairness_constraint ::=

**!** (H_Formula)

H_Formula ::=

**X** H_Formula

| H_Formula **->** H_Formula

| H_Formula **&** H_Formula

| H_Formula **|** H_Formula

| **!** H_Formula

| ( H_Formula )

| Atomic_H_formula

Atomic_H_Formula ::=

Lterm = Rterm | *True* | *False*

Lterm ::= ASM_variable_Name (the ASM_variable must be of concrete sort.)

Rterm ::=

ASM_variable_Name (the ASM_variable must be of concrete sort.)

| IntegerConstant

| SymbolicConstant

ASM_variable_Name ::= [a-bd-uw-z][A-Za-z0-9_]*

IntegerConstant ::= [0-9]*

SymbolicConstant ::= [c][A-Za-z0-9_]*

In the following sections, we will present the algorithms of checking liveness properties under fairness constraints.

## 8.3 Verification of AF*q* with fairness constraints

To verify that **AF***q* (where *q* is a Next_let _formula) holds in the initial states of an ASM *D* under the fairness constraints $!H_1$, $!H_2$, ..., $!H_n$, we build the additional ASMs to represent *q* and $H_i$ $(1 \le i \le n)$ using the algorithm described in Section 7, merge all the additional ASMs with the ASM *D* to build a composite machine *M*, and then verify that property **AF***(FlagQ=1)* holds for the initial states of *M* under the fairness constraints $!(FlagH_1 = 1)$, ..., $!(FlagH_n = 1)$. The algorithm to carry out this verification is as follows:

(1) **Check_AF_fair**($M$, $Cq$, $H_1$, ..., $H_n$)
/* $M$ is the composite machine, */
/* $G_I$ is the set of initial states of $M$, */
/* $G_T$ is the transition relation of $M$, */
/* $Cq$ is the DF representing the formula $FlagQ = 1$. */
/* $H_i$ $(1 \le i \le n)$ is the DF representing the formula $FlagH_i = 1$. */
(2) begin
(3) $\Sigma := \Phi$;
/* $\Sigma$ is a set containing DFs representing each the set of states not satisfying $FlagQ = 1$ at each transition step */
(4) $S := G_I$;
(5) $K := 0$;
(6) loop1
(7) $S_{notq}$ := **PbyS** ($S$, $C_q$ ); /*remove from S states with $FlagQ = 1$*/
(8) if $S_{notq}$ = **F** then return success;
(9) if $\exists T \in \Sigma$, **PbyS**($T$, $S_{notq}$) = **F** then return failure;
/*This step is to check if DF $S_{notq}$ covers any one of the DFs in $\Sigma$, i.e., for each DF $T$ in $\Sigma$, **PbyS**($T$, $S_{notq}$) = **F** is checked to detect a cycle. If there is a cycle, then failure is reported*/

(10) $\Sigma := \Sigma \cup \{S_{notq}\}$; /* add DF $S_{notq}$ as an element into $\Sigma$ */
(11) $S_1 := S_{notq}$;
(12) for $i$=1 to $n$ do
(13) $S_{notH} :=$ **PbyS**($S_1$, $H_i$); /*remove from $S_1$ states with $FlagH_i = 1$ */
(14) $S_2 :=$ **Conj**($S_1$, $H_i$); /* $S_2$ represents the states in $S_1$ with $FlagH_i = 1$ */
(15) if $S_2 ==$ **F** then $S_{4notq} =$ **F**;
(16) if $S_2 \neq$ **F** then begin
(17) $S_3 := S_2$ ; $S_f := S_2$; $L := 0$;
(18) loop2 /* to compute all the states reachable from $S_2$ with $FlagH_i = 1$ */
(19) $L := L+ 1$;
(20) $I_2 :=$ **Fresh**($X_m$, $L$); /* generate new input values */
(21) $N_1 :=$ **RelP**($\{I_2, S_f, G_T\}$, $X_m \cup Y_m$, $\eta'$ ); /* compute next states */
(22) $N_2 :=$ **PbyS**($N_1$, $C_q$); /* remove from $N_1$ the states with $FlagQ = 1$ */
(23) $N_3 :=$ **Conj**($N_2$, $H_i$); /* pick from $N_2$ the states with $FlagH_i = 1$ */
(24) $S_f :=$ **PbyS**($N_3$, $S_3$ ); /* compute the frontier states */
(25) if $S_f =$ **F** then exit loop2;
/* if all the states reachable from $S_2$ have been visited, exit loop 2 */
(26) $S_3 :=$ **PbyS**($S_3$ , $S_f$);
(27) $S_3 :=$ **Disj**($S_3$ , $S_f$); /* add the states of $S_f$ to $S_3$ */
(28) end loop2;
(29) $S_{41} :=$ **RelP**($\{I_2, S_3$ , $G_T\}$, $X_m \cup Y_m$, $\eta'$ ); /* compute the next states of $S_3$ */
(30) $S_4 :=$ **PbyS**($S_{41}$ , $H_i$); /* remove from $S_{41}$ the states with $FlagH_i = 1$ */
(31) $S_{4notq} :=$ **PbyS**($S_4$ , $Cq$);
(32) end_if
(33) $S_1 :=$ **Disj**($S_{4notq}$, $S_{notH}$ );
(34) end_for
(35) if $S_1 \neq$ **F** then begin
(36) K := K+1;
(37) $I_1 :=$ **Fresh**($X_m$, $K$); /* generate input values */
(38) $S :=$ **RelP**($\{I_1$ , $S_1$ , $G_T\}$, $X_m \cup Y_m$, $\eta'$ ); /* compute next states */
(39) end_if
(40) end loop1
(41) end

In this algorithm, $\Sigma$ is a set containing the DFs representing each a set of states not satisfying $FlagQ = 1$ from the states reached after every transition, $S$ represents the frontier set of states to be further explored, and $n$ is the number of fairness constraints.

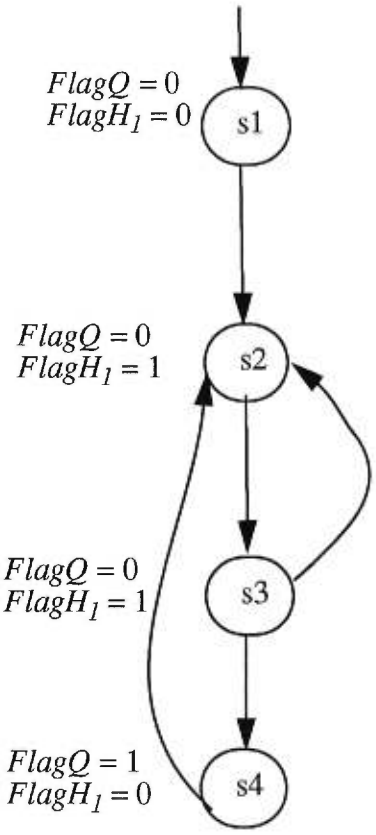
In loop1, Lines (7) - (10) verify whether the states in the frontier set $S$ satisfy $FlagQ = 1$. If yes, then the computation stops by reporting success. Otherwise, if the set of states not satisfying ($FlagQ$=1) covers any set in $\Sigma$, this means that there is at least one cycle in which the states do not satisfy ($FlagQ$=1), i.e., there is at least one path along which $\mathbf{F}(FlagQ = 1)$ does not hold. In this case, the algorithm stops and reports failure. Otherwise, the set of states not satisfying $FlagQ = 1$, which is represented by $S_{notq}$ is added to $\Sigma$.

The Lines (13) to (34) form another loop which is executed $n$ times. This loop deals with each exception condition. At every $i$th ($1 \leq i \leq n$) iteration, $S_2$ represents the set of states in $S_1$ that satisfy the excepting condition $FlagH_i = 1$. If $S_2$ is not empty, all states that are reachable from the states in $S_2$ by any number of transitions and satisfy $FlagH_i = 1$ but do not satisfy $FlagQ = 1$ are computed and stored in $S_3$. In other words, $S_3$ could contain cycles which are formed by the states satisfying $FlagH_i = 1$ but not $FlagQ = 1$. (The way to compute $S_3$ is the same as the reachability analysis, hence it may not terminate.) Then, one more transition is done to compute the set of states reachable by one transition step from the states of $S_3$, but not satisfying $FlagH_i = 1$. These states are stored in $S_4$. $S_{4notq}$ represents the set of states in $S_4$ that do not satisfy $FlagQ = 1$. $S_1$ is the union of the sets of states represented by $S_{44notq}$ and $S_{notH}$ at each iteration of the loop.

If $S_1$ is not empty, then $S$ is computed to represent the states reachable in one transition step from the states in $S_1$. The computation continues in loop1 with $S$ being the new frontier set of states to be checked.

Next we give an example showing how this algorithm works. Suppose we wish to verify $\mathbf{AF}(FlagQ = 1)$ under the fairness constraint $!(FlagH_1 = 1)$ on the ASM state transition graph given in Figure 14. In the figure, a node represents a state and an edge represents a possible transition from one state to another. We also give the values of $FlagQ$ and $FlagH_1$ in each state in the transition graph. We shall see that the algorithm stops and

reports success at the 3rd iteration in loop1. However, checking **AF**(*FlagQ* = 1) without the fairness constraint would fail on the path s1 -> s2 -> s3 -> s2 -> s3 -> s2 -> s3 ... if no fairness constraint is imposed.




initial: $\Sigma = \varnothing$;
Set($S$) = {s1};

the 1st time in loop1:
Set( $S_{notq}$) = {s1};
$\Sigma$ = {{s1}};
Set ($S_1$) = {s1};
Set ($S_2$) = $\varnothing$;
Set ($S$) = { s2 };

the 2nd time in loop1:
Set($Q$) = {s2};
$\Sigma$ = {{s1}, {s2}};
Set ($S_1$) = $\varnothing$;
Set ($S_2$) = {s2};
Set ($S_3$) = { s2, s3};
Set ($S_4$) = { s4 };
Set ($S$) = { s4};

the 3rd time in loop1:
Set($Q$) = $\varnothing$, return sucess.

**Figure 14 -** Example of checking **AF**(*FlagQ* =1) under fairness constraint !($FlagH_1$=1).

To check **AG** ( $c$ => (**F** $p$) ) where $c$ and $p$ are Next_let_formulas under the fairness constraints !$H_1$, !$H_2$, ..., !$H_n$ on an ASM $D$, we can build a composite machine $M$ from $D$ and ASMs representing $c$, $p$, $H_i$ ($1 \le i \le n$), and then transfer the problem to checking **AG**( (*FlagC*=1) => (**F** (*FlagP*=1))) on $M$ under the fairness constraints !($FlagH_i$ = 1) ($1 \le i \le n$). We first carry out reachability analysis to get all the reachable states of $M$ (represented by

$W$), collect from $W$ the states satisfying "$FlagC = 1$" ($V :=$ **Conj**($W$, $C_c$), where $C_c$ is a DF containing $FlagC = 1$), and finally apply the algorithm **Check_AF_fair** with the set $V$ as the set of initial states.

## 8.4 Verification of A$p$U$q$ with fairness constraints

To verify that **A**$p$**U**$q$ (where $p$ and $q$ are Next_let_formulas) holds in the initial state of an ASM $D$ under the fairness constraints $!H_1$, $!H_2$, ..., $!H_n$ , we build additional ASMs to represent $p$, $q$, and $H_i$ ($1 \leq i \leq n$), and then transfer the problem to checking **A**($FlagP$=1)**U**($FlagQ$=1) on the initial state of the composite machine derived from $D$ and the additional ASMs. The algorithm to verify **A**($FlagP$=1)**U**($FlagQ$=1) under the fairness constraints $!(FlagH_i = 1)$ ($1 \leq i \leq n$) is as follows:

(1) **Check_AU_fair**($M$, $C_p$, $C_q$, $H_1$, ..., $H_n$)
/* $M$ is the composite machine, */
/* $G_I$ is the set of initial states of $M$, */
/* $G_T$ is the transition relation of $M$, */
/* $C_p$ is the DF containing $FlagP = 1$. */
/* $C_q$ is the DF containing $FlagQ = 1$*/
/* $H_i$ ($1 \leq i \leq n$) is the DF representing formula $FlagH_i = 1$. */
(2) begin
(3) $\Sigma := \Phi$;
/* $\Sigma$ is a set containing DFs representing each the set of states satisfying $FlagP = 1$ but not $FlagQ = 1$ at each transition step */
(4) $S := G_I$;
(5) $K := 0$;
(6) loop1
(7) $S_{notq} :=$ **PbyS** ($S$, $C_q$ ); /*remove from $S$ states with $FlagQ = 1$*/
(8) if $S_{notq} =$ **F** then return success;
(9) if $\exists\, T \in \Sigma$, **PbyS**($T$, $S_{notq}$) = **F** then return failure;
/*This step checks if DF $S_{notq}$ covers any one of the DFs in $\Sigma$, i.e., for each DF $T$ in $\Sigma$, **PbyS**($T$, $S_{notq}$ ) = **F** is checked to detect a cycle. If there is a cycle, then failure is reported*/

(10) $R$ = **PbyS** ($S_{notq}$, $C_p$); /*remove from $S_{notq}$ states with $FlagP = 1$ */
(11) if $R \neq \mathbf{F}$ then return failure;
(12) $\Sigma := \Sigma \cup \{S_{notq}\}$; /* add DF $S_{notq}$ as an element into $\Sigma$ */
(13) $S_1 := S_{notq}$;
(14) for $i$=1 to $n$ do
(15) $S_{notH}$ := **PbyS**($S_1$, $H_i$); /*remove from $S_1$ states with $FlagH_i = 1$ */
(16) $S_2$ := **Conj**($S_1$, $H_i$); /* $S_2$ represents the states in $S_1$ with $FlagH_i = 1$ */
(17) if $S_2 == \mathbf{F}$ then $S_{4notq} = \mathbf{F}$;
(18) if $S_2 \neq \mathbf{F}$ then begin
(19) $S_3 := S_2$ ; $S_f := S_2$; $L := 0$;
(20) loop2 /* to compute al the states reachable from $S_2$ with $FlagH_i = 1$ */
(21) $L := L+ 1$;
(22) $I_2$ := **Fresh**($X_m$, $L$); /* generate new input values */
(23) $N_1$ := **RelP**($\{I_2, S_f, G_T\}$, $X_m \cup Y_m$, $\eta'$ ); /* compute next states */
(24) $N_2$ := **PbyS**($N_1$, $C_q$); /* remove from $N_1$ the states with $FlagQ = 1$ */
(25) $N_3$ := **Conj**($N_2$, $H_i$); /* pick from $N_2$ the states with $FlagH_i = 1$ */
(26) if **PbyS**($N_3$, $C_p$) $\neq \mathbf{F}$ then return failure;
/* if the states in $N_3$ do not satisfy $FlagP = 1$, report failure */
(27) $S_f$ := **PbyS**($N_3$, $S_3$ ); /* compute the frontier set of states */
(28) if $S_f = \mathbf{F}$ then exit loop2;
/* if all the states reachable from $S_2$ have been visited, exit loop 2 */
(29) $S_3$ := **PbyS**($S_3$ , $S_f$ );
(30) $S_3$ := **Disj**($S_3$ , $S_f$ ); /* add the states of $S_f$ to $S_3$ */
(31) end loop2;
(32) $S_{41}$ := **RelP**($\{I_2, S_3, G_T\}$, $X_m \cup Y_m$, $\eta'$ ); /* compute the next states of $S_3$ */
(33) $S_4$ := **PbyS**($S_{41}$ , $H_i$); /* remove from $S_{41}$ the states with $FlagH_i = 1$ */
(34) $S_{4notq}$ := **PbyS**($S_4$ , $Cq$);
(35) *if* **PbyS**($S_{4notq}$ , $Cp$) $\neq \mathbf{F}$ then return failure;
(36) end_if
(37) $S_1$ := **Disj**($S_{4notq}$, $S_{notH}$ );
(38) end_for
(39) if $S_1 \neq \mathbf{F}$ then begin
(40) $K := K+1$;
(41) $I_1$ := **Fresh**($X_m$, $K$); /* generate input values */
(42) $S$ := **RelP**($\{I_1 , S_1 , G_T\}$, $X_m \cup Y_m$, $\eta'$ ); /* compute next states */
(43) end_if
(44) end loop1
(45 end

This algorithm is similar to the algorithm **Check_AF_fair,** except that Lines (10), (11), (26) and (35) are added. Those four lines verify whether the states along a path satisfy $FlagP = 1$ before a state satisfying $FlagQ = 1$ is reached.

It is worth mentioning that the **Check_AU_fair** algorithm is conservative. It requires that $FlagP$=1 be satisfied in all the states along all paths before a state satisfying $FlagQ$ =1 is reached. Along some paths, if the states repeating forever are covered by a cycle set and there is no other state reached by those states as shown in Figure 15, **Check_AU_fair** will report failure. However, it is not necessary that $FlagP$=1 holds in these states, since this path should not even be considered. **Check_AU_fair** may thus return a false negative answer. In models of real systems, this situation happens rarely.

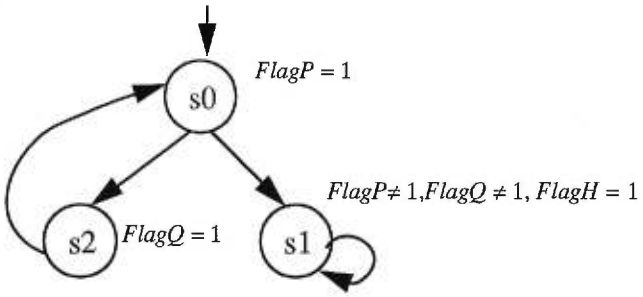



**Figure 15 -** Example of a false negative answer when verifying ($FlagP$=1)**U**($FlagQ$=1) under the fairness constraint !($FlagH$ = 1).

To check **AG**( $c$ => $p$**U**$q$ ) where $c$, $p$, $q$ are Next_let_formulas under the fairness constraints $!H_1$, $!H_2$, ..., $!H_n$ on an ASM $D$, we build a composite machine $M$ from $D$ and ASMs representing $c$, $p$, $q$, $H_i$ ($1 \leq i \leq n$), and then transfer the problem to checking **AG**( ($FlagC$=1) => (($FlagP$=1) **U** ($FlagQ$ =1))) on $M$ under the fairness constraints !($FlagH_i$ = 1) ($1 \leq i \leq n$). We then do reachability analysis to get all the reachable states of $M$ (represented by $W$), collect from $W$ the states satisfying "$FlagC$ = 1" ($V$ := **Conj**($W$, $C_c$)

where $C_c$ is a DF containing $FlagC = 1$), and finally apply the algorithm **Check_AU_fair** with the set $V$ as the set of initial states.

## 8.5 Implementation issues

To check properties expressed in $L_{MDG}$ automatically, we developed programs that

- check if the signals in a property (except the ordinary variables) are declared in the original circuit description; report any errors;
- check the syntax of the property; report any errors;
- build additional circuits to represent the Next-let-formulas in the property and the exception conditions if fairness constraints are imposed;
- merge the description of the additional circuits with the description of the original circuit, which means adding declarations of components and signals of the additional circuits to the original circuit description file and the variable order file;

The above programs were implemented in C with Yacc & Lex. The model checking algorithms were developed using the current MDGs package implemented in Quintus Prolog V3.2 [80].

# Summary

In this chapter, we gave the definition of fairness constraints in our verification system, and presented the algorithms of verification of liveness properties under fairness constraints.

In the next chapter, we show that the verification procedures presented in this thesis are sound.

# 9 Soundness of the Verification Procedures

In this chapter, we show that the verification procedures presented in Chapters 6 and 7 are sound. This is achieved by stating and proving a number of theorems.

## 9.1 Correctness of Algorithm Check_AG(M,C)

**Theorem 1:** *The result of checking AG(p) where p is a Next_let_formula on an ASM D is the same as the result of checking AG(Flag=1) on the composite machine M built from D and Dp, where Dp is an ASM derived from the Next_let_formula p and constructed according to the algorithm in Chapter 7.*

PROOF. We prove this theorem by case splitting according to the definition of the Next_let_formula.

According to the algorithm in Section 7, to check a property in the template of **AG**($p$), we build an ASM of the next_let_formula $p'$ = *False* | $p$ .

- Case 1: if $p$ is a basic formula of $L_{MDG}$, i.e., $t_1 = t_2$, where $t_1$ is an ASM_variable of $D$, and if $t_2$ is also an ASM_variable, then the composite machine $M$ is as shown in Figure 16.

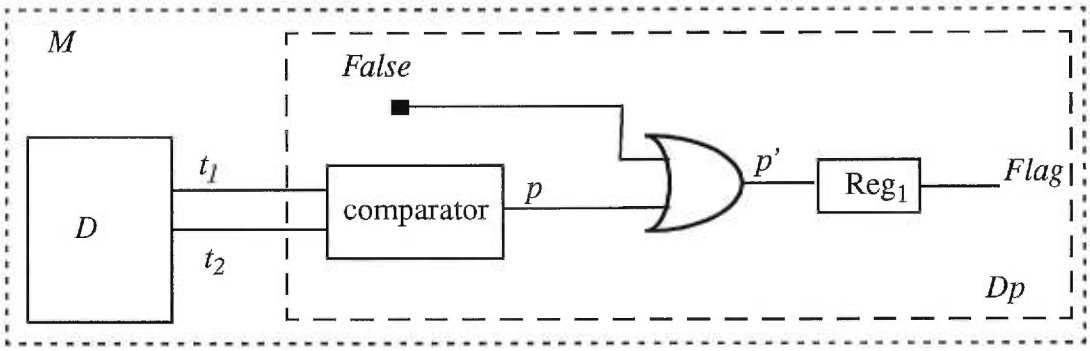



**Figure 16** - A composite machine for checking **AG**($t_1 = t_2$) when $t_2$ is an ASM_variable.

If $t_2$ is an individual or generic constant, then the composite machine M is as shown in Figure 17.

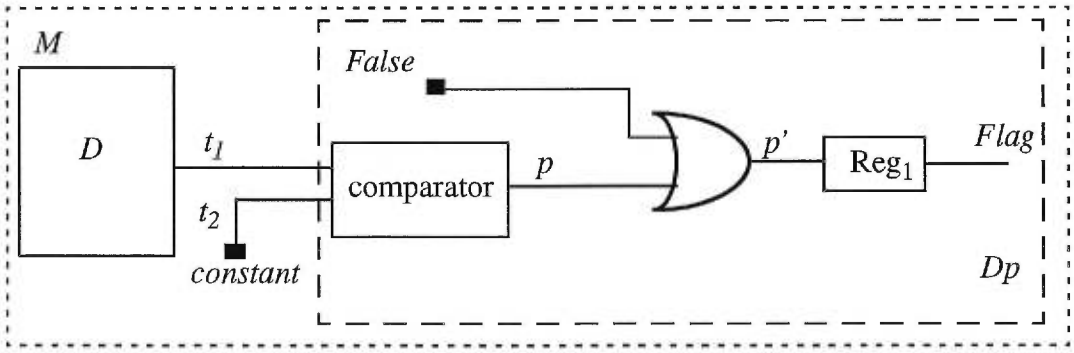



**Figure 17** - A composite machine for checking **AG**($t_1 = t_2$) when $t_2$ is a constant.

When $t_1 = t_2$ is true (or false) at a state $s$ of an ASM $D$, we can see from Figures 16 and 17 that signal *Flag* denotes truth (or falsity) of the equality in the subsequent state of the composite machine M. As $\text{Reg}_1$ is initialized to Boolean constant *True*,

the result of checking **AG**($t_1 = t_2$) on $D$ is the same as that of checking **AG**(*Flag*=1) on $M$. Without loss of generality, we assume that $t_1$ and $t_2$ are ASM_variables in the following proofs (the signal $t_2$ coould be tied to a constant value inside $D$).

- Case 2: if $p$ is !( $t_1 = t_2$) then the composite machine $M$ is as shown in Figure 18.

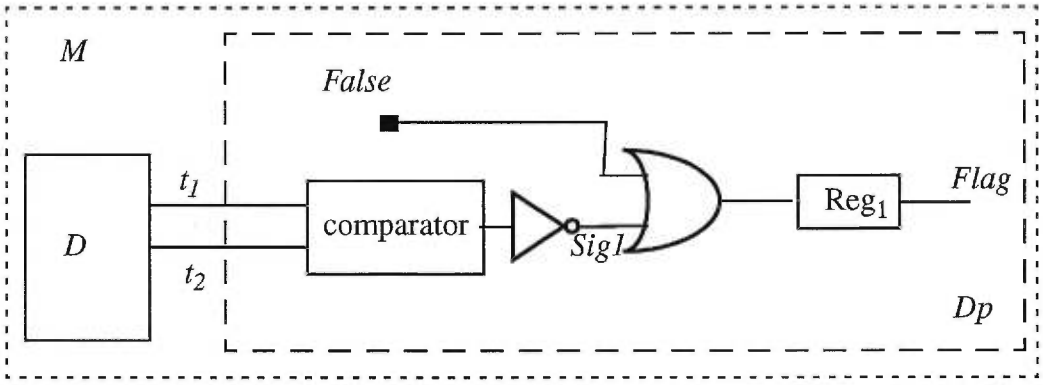



**Figure 18** - A composite machine for checking **AG**(!($t_1 = t_2$)) .

In Figure 18, *Sig1* denotes truth (or falsity) of !($t_1 = t_2$) in every state $s$ of $D$, *Flag* denotes truth (or falsity) of !($t_1 = t_2$) in the corresponding state following $s$ of the composite machine $M$. By initializing $\text{Reg}_1$ to *True*, the result of checking AG(!($t_1 = t_2$)) on $D$ is the same as that of checking **AG**(*Flag*=1) on $M$.

- Case 3: if $p$ is ($t_1 = t_2$) & ($t_3 = t_4$) then the composite machine $M$ is as shown in

Figure 19.

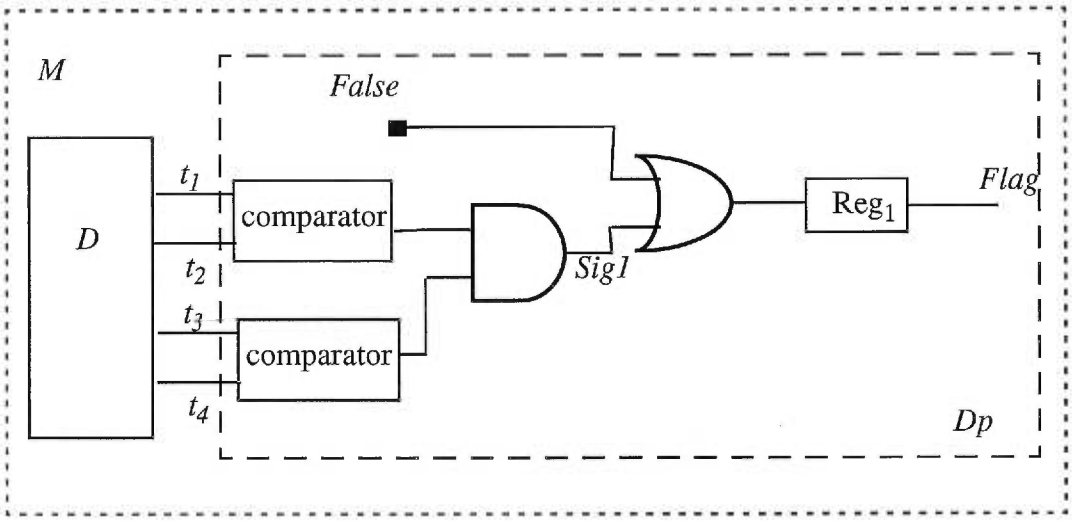



**Figure 19** - A composite machine for checking **AG**$((t_1 = t_2)$ & $(t_3 = t_4))$ .

In Figure 19, *Sig1* shows the truth (or falsity) of $(t_1 = t_2)$ & $(t_3 = t_4)$ in any state $s$ of $D$, *Flag* denotes truth (or falsity) of $(t_1 = t_2)$ & $(t_3 = t_4)$ in the corresponding next state of $s$ of the composite machine $M$. Therefore, if we initialize $\text{Reg}_1$ to *True*, the result of checking **AG**$((t_1 = t_2)$ & $(t_3 = t_4))$ on the original machine $D$ is the same as that of checking **AG**(*Flag*=1) on the composite machine M.

If $p = (t_1 = t_2) \mid (t_3 = t_4)$, then the AND gate is replaced by an OR gate; If $p = (t_1 = t_2) \text{->} (t_3 = t_4)$, we treat $p$ as $(!\ (t_1 = t_2)) \mid (t_3 = t_4)$.

- Case 4: if $p = \mathbf{X}(t_1 = t_2)$ then $p' = \mathit{False} \mid \mathbf{X}(t_1 = t_2)$. The composite machine $M$ is

as shown in Figure 20.

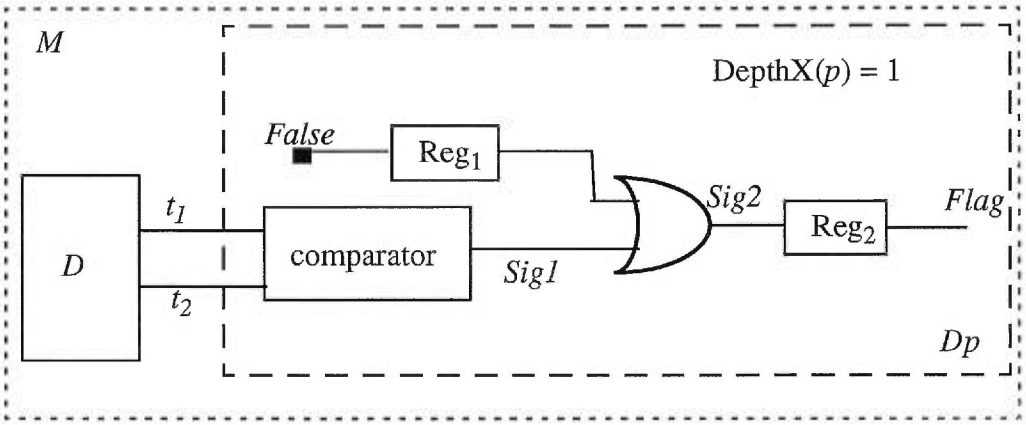



**Figure 20** - A composite machine for checking **AG**(**X**($t_1 = t_2$)) .

**AG**(**X**($t_1 = t_2$)) means that **X**($t_1 = t_2$) must hold on every state along all the computation paths, i.e., ($t_1 = t_2$) must hold on every state except that the truth (or falsity) of ($t_1 = t_2$) at the initial states is not in our concern. In the above figure, *Sig1* denotes truth (or falsity) of $t_1 = t_2$ in every state of D. By initializing $\text{Reg}_1$ to 1, Sig2 denotes truth (or falsity) of $t_1 = t_2$ in every state except that it denotes truth at the initial states. Therefore, with $\text{Reg}_2$ being initialized to *True*, the result of checking **AG**(**X**($t_1 = t_2$)) on $D$ is the same as that of checking **AG**($Flag$=1) on $M$.

- Case 5: if $p$ is of form **LET** ($v_1 = t_1$) **IN** (**X**($t_2 = v_1$)), then the composite machine is as shown in Figure 21. If $p$ is of form **LET** ($v_1 = t_1$) **IN** (**X**($t_2$ = function($v_1$))), i.e., **LET** ($v_1 = t_1$) **IN** (**X**($t_2$ = finc($v_1$))) , where finc represents the function "increase by i", then a component representing the function is added and the output of the function component becomes one of the inputs of the comparator.

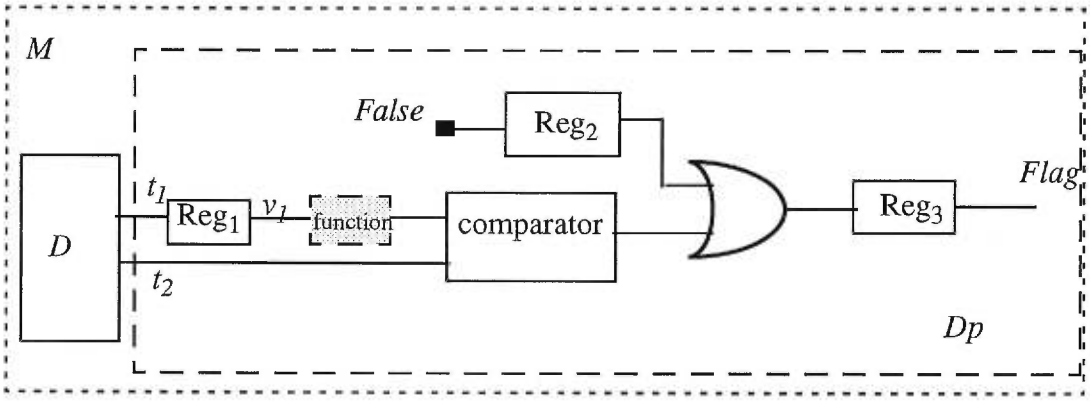



**Figure 21** - A composite machine for checking **AG**(**LET** $(v_1 = t_1)$ **IN** (**X**$(t_2 = v_1)$)) .

**AG**(**LET** $(v_1 = t_1)$ **IN** (**X**$(t_2 = v_1)$)) means that the value of $t_1$ in every state $s_i$ must be the same as that of $t_2$ in the next state of $s_i$. In the composite machine as shown in Figure 21, the output of the comparator indicates in each state (except the initial states as the initial value of $\text{Reg}_1$ can be any as long as it fits in the sort of $t_1$) the equality of comparing the value $t_1$ one transition earlier and the value of $t_2$ (an ordinary variable $\text{v}_1$ is used to remember the value of $t_1$ one transition earlier). By initializing $\text{Reg}_2$ and $\text{Reg}_3$ to *True*, *Flag* will indicate *True* for the initial set of states and the states following the initial states, and then the value of *Flag* depends on the output of comparator which depends on the value of $t_1$ and $t_2$ in $D$. Thus, the result of checking **AG**(**LET** $(v_1 = t_1)$ **IN** (**X**$(t_2 = v_1)$)) on the original machine $D$ is the same as that of checking **AG**(*Flag*=1) on the composite machine $M$.

In the above 5 cases, $p$ ranged over the basic structures of the Next_let_formulas. Below we show that when $p$ is a general Next_let_formula, the result of checking **AG**($p$) on an ASM **D** is still equivalent to the result of checking **AG**(*Flag*=1) on the composite machine M.

- Case 6: if $p$ is !($q$) where $q$ is a Next_let_formula, then $p'$ = *False* | !($q$). The

composite machine is as shown in Figure 22.

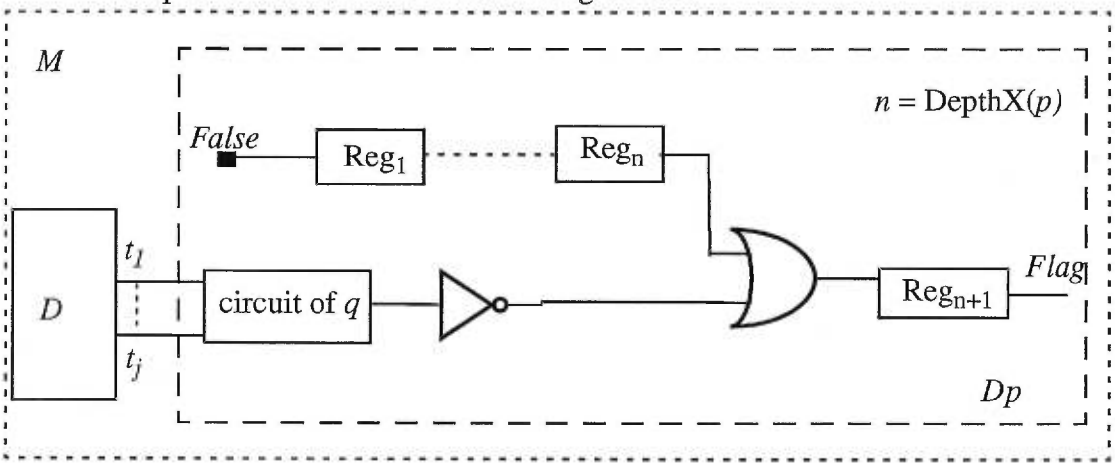



**Figure 22** - A composite machine for checking **AG**(!($q$)) when $q$ is a Next_let_formula.

$n$ = DepthX($p$) is the maximum depth of **X** operators in $p$ (this function DepthX is defined in Section 7). $n$ registers are added, with the primary input the constant *False*. This allows to ignore the output of !($q$) in the first $n$ clock cycles, i.e., to check !($q$) $n$ clock cycles later from the initial state. "circuit of $q$" is built using the basic components exposed in cases 1 to 5. Corresponding to the truth (or falsity) of !($q$) in a state $s$ of machine $D$, *Flag* denotes the truth (or falsity) of !($q$) in the corresponding ($n$+1)th state after $s$ of the composite machine $M$. As $\text{Reg}_1, \ldots, \text{Reg}_n$ and $\text{Reg}_{n+1}$ are initialized to *True*, the result of checking **AG**(!($q$)) on $D$ is exactly the same as that of checking **AG**(*Flag*=1) on $M$.

- Case 7: if $p$ is of the form $q_1$ & $q_2$, where $q_1$ and $q_2$ are Next_let_formulas, and assuming that DepthX($q_1$) ≤ DepthX($q_2$), then the composite machine is as

shown in Figure 23.

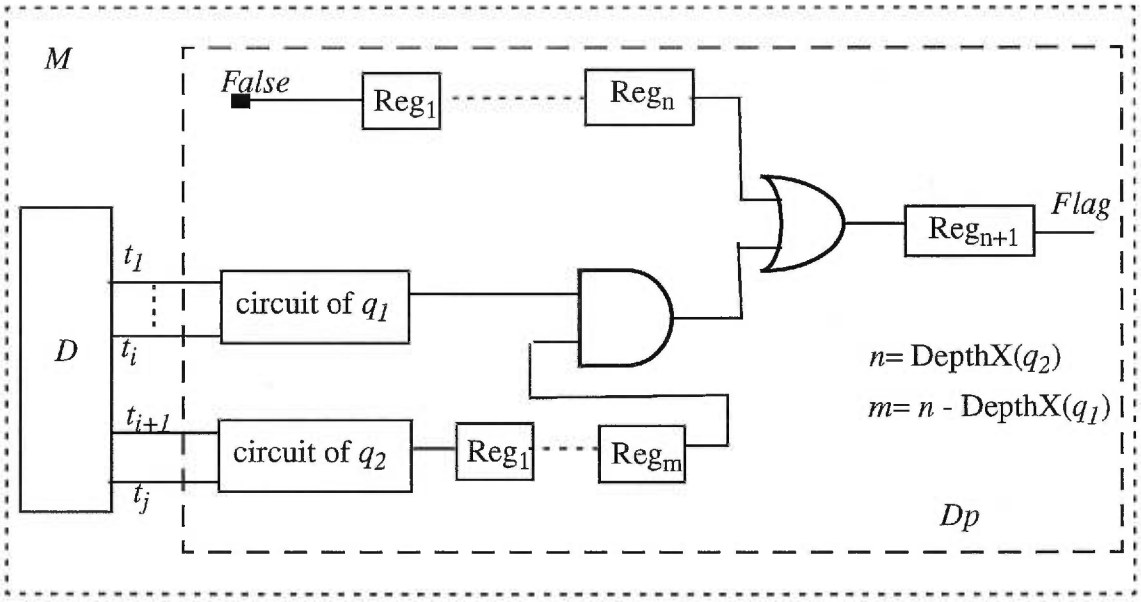



**Figure 23** - A composite machine for checking **AG**($q_1$ & $q_2$) when $q_1$ , $q_2$ are Next_let_formulas.

The blocks "circuit of $q_1$" and "circuit of $q_2$" are constructed using the basic components as shown in cases 1 to 5. The truth (or falsity) of $q_1$ & $q_2$ at a state $s$ of machine $D$ corresponds to the truth of *Flag* which denotes the truth (or falsity) of $q_1$ & $q_2$ in the corresponding ($n$+1)th state after $s$ of $M$, where $n$ is the maximum depth of **X** operators in $p$. Since $\text{Reg}_{1, \ldots,} \text{Reg}_n$ and $\text{Reg}_{n+1}$ are initialized to *True*, the result of checking **AG**($q_1$ & $q_2$) on $D$ is the same as that of checking **AG**(*Flag*=1) on $M$.

- Case 8: if $p$ is of form **X**($q$), where $q$ is also a Next_let_formula, then the

composite machine is as shown in Figure 24.

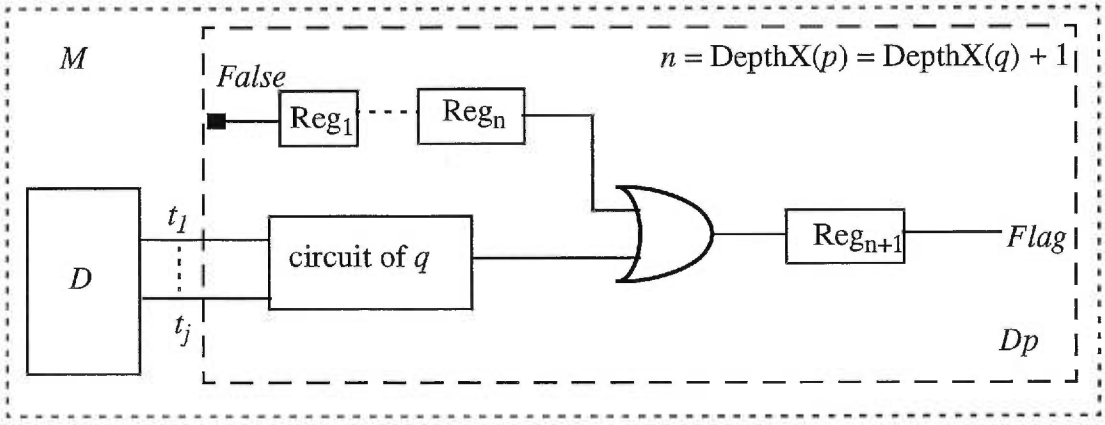



**Figure 24** - A composite machine for checking **AG**(**X**(*q*)) when *q* is a Next_let_formula.

Corresponding to the truth (or falsity) of **X**(*q*) in a state *s* of *D*, *Flag* denotes the truth (or falsity) of **X**(*q*) in (*n*+1)th state after *s* of the composite machine *M*. As $Reg_{1, \ldots}$, $Reg_n$ and $Reg_{n+1}$ are initialized to *True*, the result of checking **AG**(**X**(*q*)) on *D* is the same as that of checking **AG**(*Flag*=1) on *M*.

- Case 9: if *p* is of form **LET** $(v_1 = t_1)$ &..&$(v_m = t_m)$ **IN** (*q*), where *q* is a Next_let_formula containing the ordinary variables $v_1$, ... $v_m$, then *p'* = *False* | (**LET** $(v_1 = t_1)$ &..&$(v_m = t_m)$ **IN** (*q*) ). The composite machine is as shown in Figure 25.

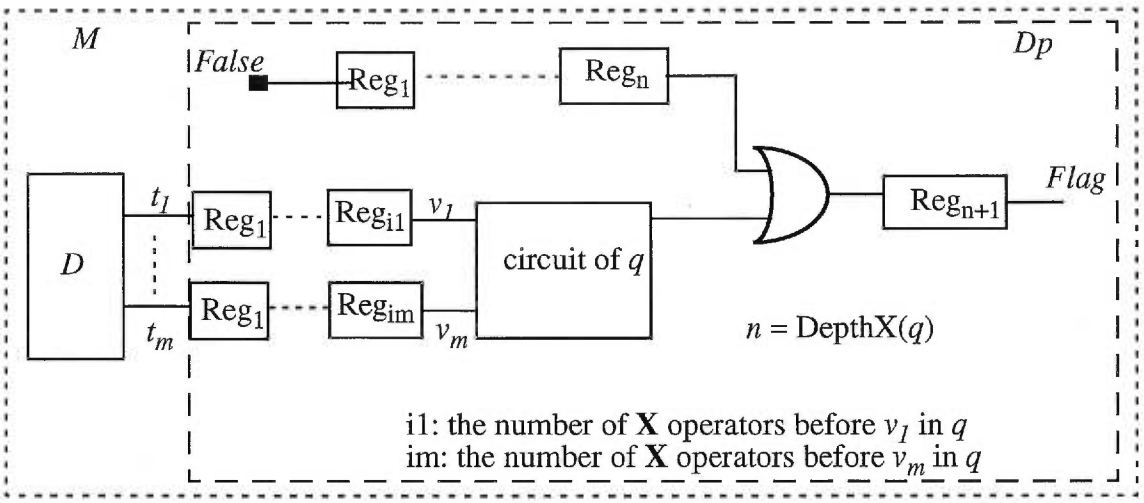



**Figure 25 -** A composite machine for checking **AG(LET** $(v_1 = t_1)$ &..&$(v_m = t_m)$ **IN** $(q)$)

Corresponding to the truth (or falsity) of "**LET** $(v_1 = t_1)$ &..&$(v_m = t_m)$ **IN** $(q)$)" *in* state $s$ of machine $D$, *Flag* denotes the truth (or falsity) of "**LET** $(v_1 = t_1)$ &..&$(v_m = t_m)$ **IN** $(q)$)" in the $(n+1)$th state after $s$ of the composite machine $M$. Hence, since $Reg_1$, ...., $Reg_n$, $Reg_{n+1}$ are initialized to *True*, the result of checking **AG(LET** $(v_1 = t_1)$ &..&$(v_m = t_m)$ **IN** $(q)$) on the original machine $D$ is the same as that of checking **AG**(*Flag*=1) on the composite machine $M$.

From Cases 1 to 9, according to the definition of the Next_let_formula, we have analyzed all the cases that a Next_let_formula can have. Therefore, we have proved Theorem 1.

We can also prove in a similar fashion that the result of checking **A**($p$), **AF**($p$), **A**($p$)**U**($q$) on the original machine $D$ where $p$, $q$ are Next_let_formulas is the same as that of checking **A**(*Flag*=1), **AF**(*Flag*=1), **A**(*FlagP*=1)**U**(*FlagQ*=1), respectively, on $M$

constructed using the algorithm in Section 7.

**Definition 9.1:** *A property checking algorithm is correct iff the algorithm succeeds (fails) when the property is true (not true) on the ASM being verified according to the semantics defined in Section 4.2.*

According to the semantics in Section 4.2, when a property is true, it is supposed to be true for all the interpretations of an ASM D. When the model checking algorithm reports failure, which means that the property is not true for all the interpretations, it is still possible that the property holds for a specific interpretation, hence our model checking algorithm could give a false negative result for that interpretation. However, this is not of our concern, because we consider the correctness according to definition 9.1.

**Theorem 2:** *The algorithm Check_AG(M,C) given in Section 6.2.1 is correct.*

PROOF. The algorithm **Check_AG**($M$, $C$) verifies if all the reachable states of the ASM $M$ satisfy the condition $C$ ($Flag$ =1). To prove the correctness of this algorithm, we do induction on the number of transition steps $K$.

•When $K = 0$, $Q$ and $R$ are both DFs of type $U \rightarrow Y_m$ representing the set of initial states. The following lines are executed:

(5) $P$ := **PbyS**($Q$, $C$);
(6) if $P \neq$ **F** then return failure;
(7) $K := K + 1$;
(8) $I$ := **Fresh**($X_m$, $K$); /*generate input values */
(9) $N$ := **RelP**($\{I, Q, G_T\}$, $X_m \cup Y_m$, $\eta'$ ); /* compute next states */
(10) $Q$ := **PbyS**($N$, $R$); /* compute frontier set of states */
(11) $Q$ = **F** then return success; /* if fixpoint reached, report success */
(12)$R$ := **PbyS**($R$, $Q$); /* simplify R by removing states subsumed by Q */
(13)$R$ := **Disj**($R$, $Q$); /* compute all states reached so far */

At Line (5), according to the definition of **PbyS** in Section 3.3, we get $P$ as a DF of type $U \rightarrow Y_m$ obtained by pruning the disjuncts in $Q$ that are subsumed by $C$ , i.e., Set($P$) = Set($Q$) \ Set($C$). At Line (6), if $P \neq \mathbf{F}$, meaning that it is not the case that all the initial states satisfy $Flag = 1$, then property AG($Flag$ =1) does not hold. Otherwise, if $P = \mathbf{F}$, then necessarily Set($Q$) $\subseteq$ Set($C$), meaning that all the initial states satisfy $Flag = 1$. In this case, the computation continues. In Lines (7), (8), (9), (10), the next states and the frontier set of states reached in one transition step are computed. At Line (11), if $Q = \mathbf{F}$, which means that set of states reached from the initial states by one transition step is already covered by the set of initial states, i.e., no new states have been generated, and it was already verified that all the initial states satisfy $Flag$ =1, hence AG($Flag$ =1) holds in this case. Otherwise, if new states are generated, the algorithm continues. Therefore, according to the above analysis, the algorithm gives the correct result at $K = 0$.

- Suppose that the algorithm produces the correct result up to $K = n$, then $R$ is a DF representing the set of all the reachable states after $n$+1 transition steps, $Q$ is a DF representing the frontier set of states generated after $n$+1 transition steps, and all the states in Set($R$) except those in Set($Q$) satisfy $Flag$ =1. At $K = n$+1, in Lines (5), (6), all the states in Set($Q$) are checked if they satisfy $Flag$ =1. If not, then AG($Flag$ =1) does not hold, and the algorithm does stop and report failure; if yes, meaning that all the reachable states in Set($Q$) satisfy $Flag$ =1, the algorithm continues to compute the frontier set of the states reachable in $n$+2 transition steps in Lines (7) - (10). At Line (11), if $Q = \mathbf{F}$, meaning that all the frontier states have been seen (the fixpoint has been reached), then the algorithm stops by reporting success. Otherwise, the algorithm continues. Hence, the algorithm gives a correct result at $K = n$+1.

From the above two cases, it follows by induction on $K$ that Theorem 2 holds.

However, if the reachability analysis of a particular ASM does not terminate, i.e., new next states are generated at every transition, the algorithm will not stop. The reasons of non-termination and a proposal for some solutions are addressed in [82].

## 9.2 Correctness of Algorithm Check_AF(*M*, *C*)

Before we can prove the correctness of the algorithm **Check_AF**(*M*, *C*), we need to prove the following lemmas.

**Lemma 1:** *Let a directed graph G = <V, E> such that V is a set of nodes and |V| = n is the number of elements in V, E is a set of node pairs <$v_i$, $v_j$> representing a directed edge from node $v_i$ to node $v_j$, and there is at least one edge leading to each node in V, then there exists at least one cycle in G and the cycle consists of less than n edges.*

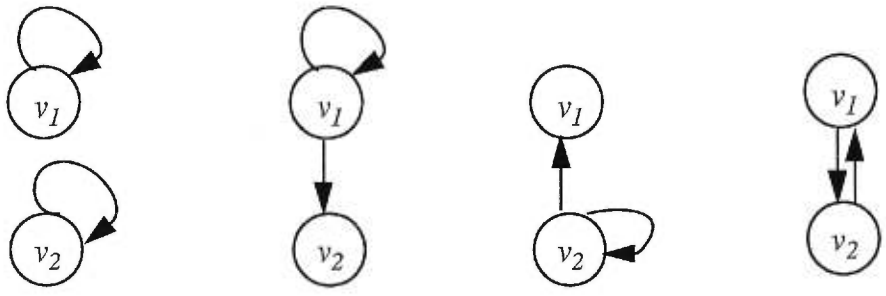



**Figure 26** - One case of Lemma 1 when |*V*| = 2 and only one edge leads to each node.

PROOF. To prove this lemma, we have to detect a cycle in the graph.

Take a node from $V$ and mark it as $v_1$, find the edge pointing to $v_1$, if the edge is $<v_1, v_1>$, then a cycle is found. Otherwise, we record the edge as $v_1$ <-- $v_2$. Then we search for an edge pointing to $v_2$. If the edge is $<v_1, v_2>$ or $<v_2, v_2>$, then a cycle is found. Otherwise, we have $v_1$ <-- $v_2$ <-- $v_3$. In general, we may have $v_1$ <-- $v_2$ <-- $v_3$ <-- ... <-- $v_{i-1}$ <-- $v_i$ ($i < n$). If $v_i = v_k$ ($k \leq i$), then a cycle exists with less than $n$ edges. Otherwise, we keep following the edges backword until $v_1$ <-- $v_2$ <-- $v_3$ <-- ... <-- $v_{n-1}$ <-- $v_n$. Since each node has at least one edge leading to it, there must exist an edge leading to $v_n$, i.e., there must exist a node $v_m$ ($1 \leq m \leq n$) such that $<v_m, v_n> \in E$ . Figure 27 illustrates such a case. Therefore, a cycle exists containing nodes $v_m$, $v_{m+1}$, ..., $v_{n-1}$, $v_n$, consisting of $n$-$m$+1 edges. When $m$ = 1, i.e., $v_m = v_1$, there are $n$ edges in the cycle.

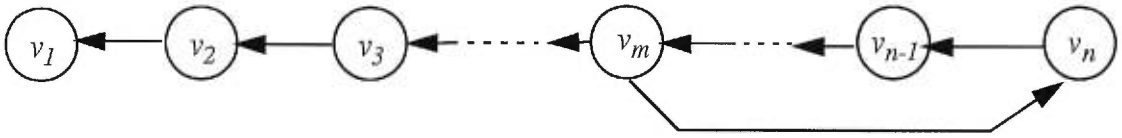



**Figure 27 -** One case of Lemma 1 when $|V| = n$ and only one edge points to each node.

**Lemma 2:** *Suppose that $S_1$ and $S_2$ are sets of states, $|S_1| = n$, every state s in $S_1$ has a next state s' (the next state s' could be the same as s), $S_2$ is derived from $S_1$ by one transition, and $S_1 \subseteq S_2$. Starting from any state in $S_1$, there exists a path that forms a cycle consisting of at most n transitions.*

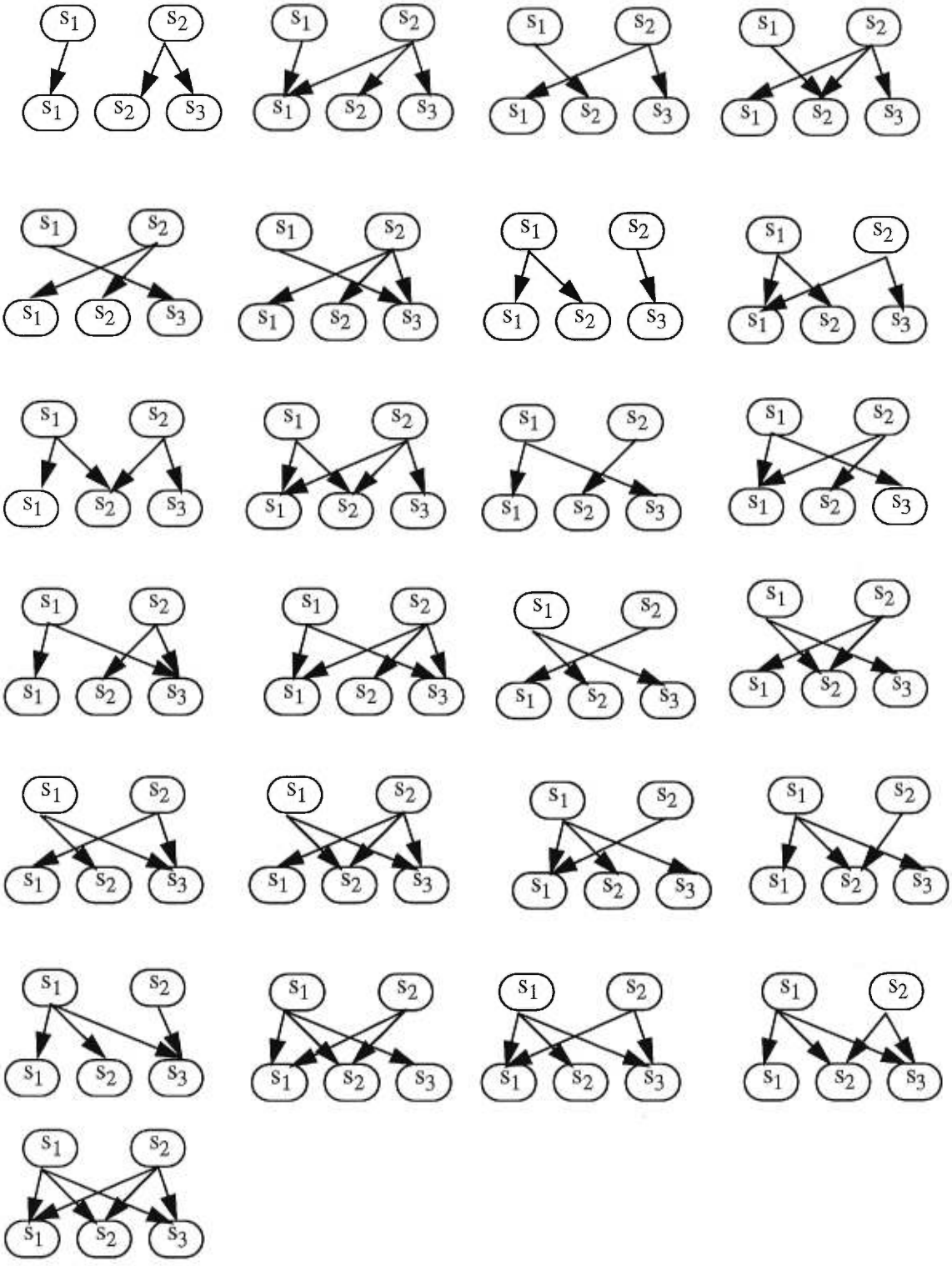



**Figure 28** - One case of Lemma 2 when $|S_1| = 2$ , $|S_2| = 3$.

PROOF. To prove this lemma, we show that such a cycle can always be constructed.

We build a graph $G = <V, E>$ such that $V$ is a set of nodes, a node represents a state in $S_1$, $|V| = n$; $E$ is a set of directed edges, with each edge $<s_i, s_j>$ ($1 \le i \le n$, $1 \le j \le n$) indicating that $s_j$ is reached from $s_i$ in one transition.

Since $S_2$ is derived from the states in $S_1$ by one transition and $S_1 \subseteq S_2$, then all the states in $S_1$ are again reached from the states in $S_1$ in one transition. In the other words, there is at least one edge leading to each node in V in G. Figure 28 shows such a case when $|S_1| = 2$ and $|S_2| = 3$. According to Lemma 1, there is at least one cycle in G and the cycle consists of at most $n$ edges. Suppose that the cycle consists of states $s_{i+1}, s_{i+2}, ..., s_{i+m}$ ($i \ge 0$, $i+m \le n$), starting from state $s_{i+1} \in S_1$, and following the edges in the cycle, after at most $n$ transitions, we get a path containing a cycle.

**Lemma 3:** *Suppose that $S_1$ and $S_2$ are sets of states, $S_2$ is reached from $S_1$ in m ($m \ge 1$) transitions, and $S_1 \subseteq S_2$. Starting from any state in $S_1$, after $n = |SI| \times m$ transitions, there exists a path that forms a cycle.*

PROOF. This lemma can be proven by viewing the $m$ transition steps from the states in $S_1$ to the states in $S_2$ as one "macro" transition and then by applying Lemma 2.

**Theorem3:** *The algorithm Check_AF(M, C) given in Section 6.2.3 is correct.*

PROOF. The algorithm **Check_AF**(*M*, *C*) verifies whether there exists a state satisfying *C* (*Flag* =1) along every path in the infinite computation forest derived from M. To prove the correctness of the algorithm, we do induction on the transition step number *K*.

•When $K = 0$, *P* represents the set of initial states, Σ is an empty list. The algorithm executes the following lines:

(7) $Q :=$ **PbyS**$(P, C)$;
(8) if $Q =$ **F** then return success;
(9) if $\exists T \in \Sigma$, **PbyS**$(T, Q) =$ **F** return failure;
(10) $\Sigma := \Sigma \| [Q]$;
(11) K := K+1;
(12) $I :=$ **Fresh**$(X_m, K)$;
(13) $P :=$ **RelP**$(\{I, Q, G_T\}, X_m \cup Y_m, \eta')$;

At Line (7), *Q* is a DF representing the set: Set(*Q*) = Set(*P*) \ Set(*C*). At Line (8), if $Q =$ **F**, which means Set(*P*) ⊆ Set(*C*), i.e., all the initial states satisfy $Flag = 1$ and the property **AF**$(Flag = 1)$ holds. If it is not true that $Q =$ **F**, which means that some of the initial states do not satisfy $Flag = 1$, the computation continues. Line (9) is skipped since Σ is empty. At Line 10, the algorithm records *Q* as an element in Σ and computes the next states derived from Set(*Q*) in Lines (11) (12) (13). The algorithm thus gives a correct result when $K = 0$.

•Suppose that the algorithm gives a correct result up to $K = n$. Σ is then a list containing *n* elements, the 1st element is the DF representing the initial states not satisfying $Flag = 1$, and the *i*th $(2 \le i \le n)$ element in Σ (which is a DF) represents the set of states that do not satisfy $Flag = 1$ and are reached in one transition from the states in the $(i\text{-}1)$th element of Σ. *P* is a DF representing the set of states generated after one transition from the states in the *n*th element of Σ.

When $K = n+1$, at Lines (7), (8), we check if all the states in Set(*P*) satisfy *Flag*

=1. If yes, then it means that for every computation path a state satisfying *Flag* =1 is found and **AF**(*Flag* = 1) holds. Otherwise, we check if Set($Q$) covers any set in $\Sigma$. At Line (9), if $\exists\, T \in \Sigma$, **PbyS**($T$, $Q$) = **F,** then Set($T$) $\subseteq$ Set($Q$). Since Set($Q$) is derived from Set($T$), then according to Lemma 3 there is at least one cycle in the computation paths, and the states along the cycle do not satisfy *Flag* = 1. Therefore, **AF**(*Flag* = 1) does not hold. If no cycle is detected, the computation continues. It follows that the algorithm gives a correct result at $K = n+1$.

From the above two cases, by induction on $K$, we have proven Theorem 3. However, there still exists the non-termination problem in this algorithm if the reachability analysis of a particular ASM does not terminate, and no cycle is detected among the states not satisfying (*Flag* = 1). Figure 29 shows such a case. The reasons of non-termination and a proposal for some solutions are addressed in [82].


Note: No state in $\{s_0, s_1, \ldots s_n, \ldots\}$ satisfies $C$.

**Figure 29 -** A case of non-termination of algorithm **Check_AF***(M, C)*

# Summary

We have proven the correctness of the algorithms **Check_AG***(M, C)* and **Check_AF***(M, C*. The proofs of correctness for **Check_A**(*M, n, C*), **Check_AU**(*M, C*), **Check_EX**(*M, n, C*), **Check_EG**(*M, C*), **Check_EF**(*M, C*), **Check_EU**(*M, C*), and for the algorithms for checking **AG** (*c* => (**F** *p*)) and **AG**(*c* => *p***U***q*) can be carried out in a similar way as the

proofs of Theorems 2 and 3.

In the next chapter, we illustrate the property checking procedures on two examples.

# 10 Experimental results

In this chapter, we apply the MDG-based model checker introduced earlier to two hardware design examples: the Island Tunnel Controller (ITC) [39], and the Abstract Counter [34]. Although the two examples are small and do not represent the scale of designs that MDG-based model checker can verify, they are ideal for illustration purposes. From the two examples, we can see how the ASMs are used to describe design models, and how the properties can be stated using $L_{\text{MDG}}$. We also carried out the same verification using the ROBDD-based verification tool VIS [83]. Both tools showed the same verification result. However, using the MDG-based method, we were able to use abstract variables that describe the data path and the first-order temporal logic to state properties, hence, the performance of the MDG-based model checker is much better than that of VIS.

## 10.1 Checking Properties of the Island Tunnel Controller

The Island Tunnel Controller was originally introduced by Fisler and Johhson [39] to

illustrate the notation of a heterogeneous logic system supporting diagrams as logic entities, however, no verification experiment were performed.

### 10.1.1 The Island Tunnel Controller

Generally speaking, the ITC controls the traffic lights at both ends of a tunnel based on the information collected by sensors installed at both ends of the tunnel: there is one lane tunnel connecting the mainland to an island, as shown in Figure 30. At each end of the tunnel, there is a traffic light. There are four sensors for detecting the presence of vehicles: one at the tunnel entrance (*ie*) and one at the tunnel exit on the island side (*ix*), and one at the tunnel entrance (*me*) and one at the tunnel exit on the mainland side (*mx*). It is assumed that all cars are finite in length, that no car gets stuck in the tunnel, that cars do not exit the tunnel before entering the tunnel, that cars do not leave the tunnel entrance without travelling through the tunnel, and that there is sufficient distance between two cars such that the sensors can distinguish the cars.

In [39], one more constraint is imposed: "at most sixteen cars may be on the island at any time". The number "sixteen" can be taken as a parameter and it can be any natural number. The constraint can thus be read as follows: "at most $n$ ($n \geq 0$) cars may be on the island at any time". In our ASM approach, we have the luxury to model an abstract data path, hence, we used an abstract variable to describe the counter $n$. For ROBDD-based verification methods, like VIS, a particular instance of $n$ has to be given.

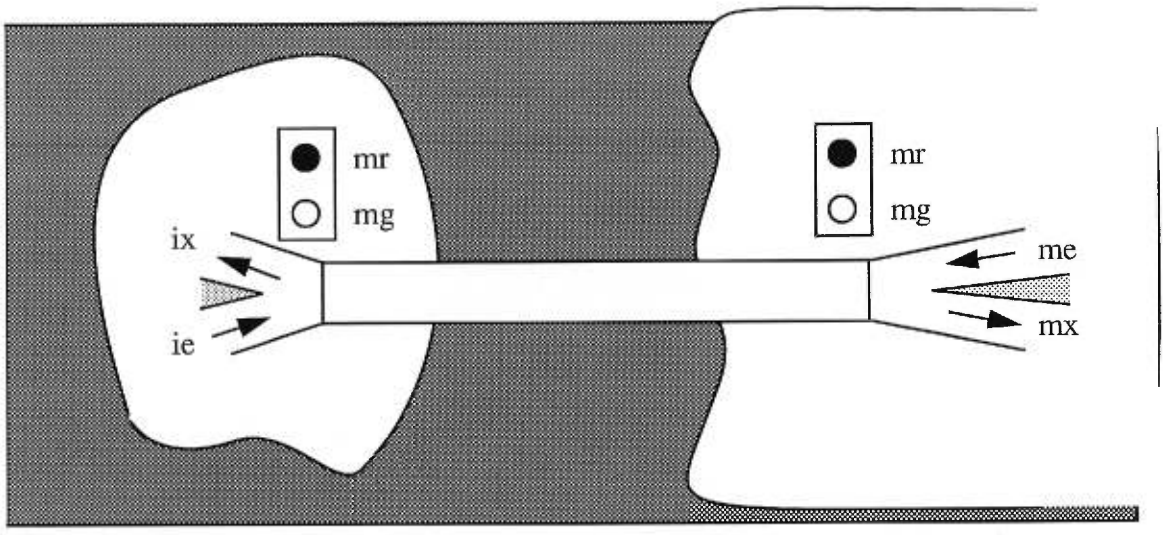



**Figure 30** - The Island Tunnel Controller

Fisler and Johnson proposed a specification of ITC using three communicating controllers and two counters as shown in Figure [31]. Their state transition diagrams are shown in Figure 32. The island light controller (ILC) has four states: *green*, *entering*, *red* and *exiting*. The outputs *igl* and *irl* control the green and red lights on the island side, respectively; *iu* indicates that the cars from the island side are currently occupying the tunnel, and *ir* indicates that ILC is requesting the tunnel. The input *iy* requests the ILC to release control of the tunnel, and *ig* grants control of the tunnel from the island side. A similar set of signals is defined for the mainland light controller (MLC). The tunnel controller (TC) processes the requests for access issued by the ILC and MLC. The island counter and the tunnel counter keep track of the numbers of cars currently on the island and in the tunnel, respectively. For the tunnel controller, at each clock cycle, the counter *tc* is increased by 1 depending on *tc+* or decremented by 1 depending on *tc-* unless it is already 0. The island counter operates in a similar way, except that the increment and

decrement signals are *ic+* and *ic-*, respectively.

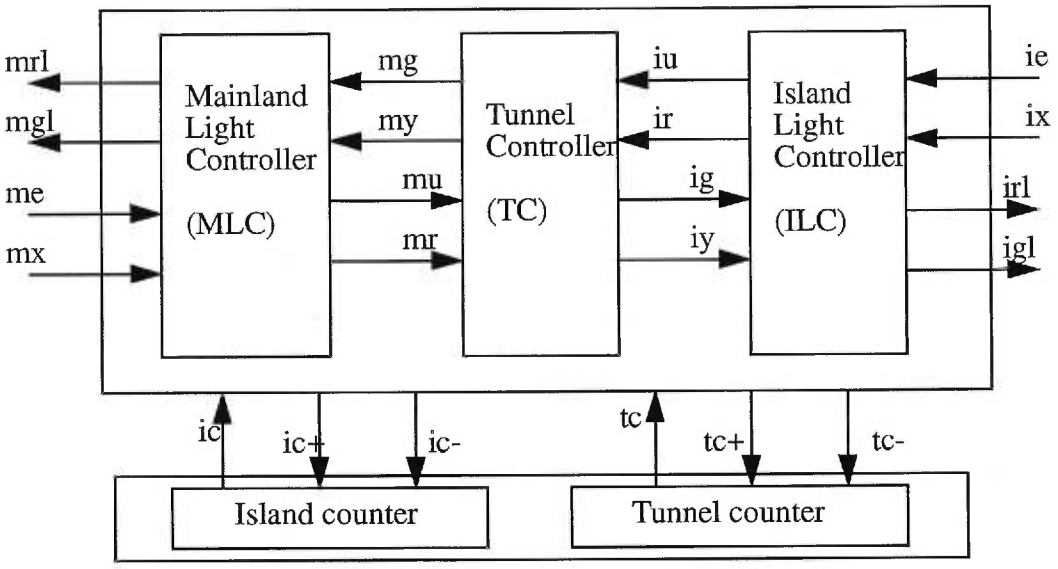



**Figure 31** - The specification of the Island Tunnel Controller

In [39], Fisler and Johnson proposed a set of properties that the ITC design should satisfy. In the next section, we will show how those properties are specified in $L_{MDG}$, and the CPU time and memory used for verifying the properties using the MDG package.

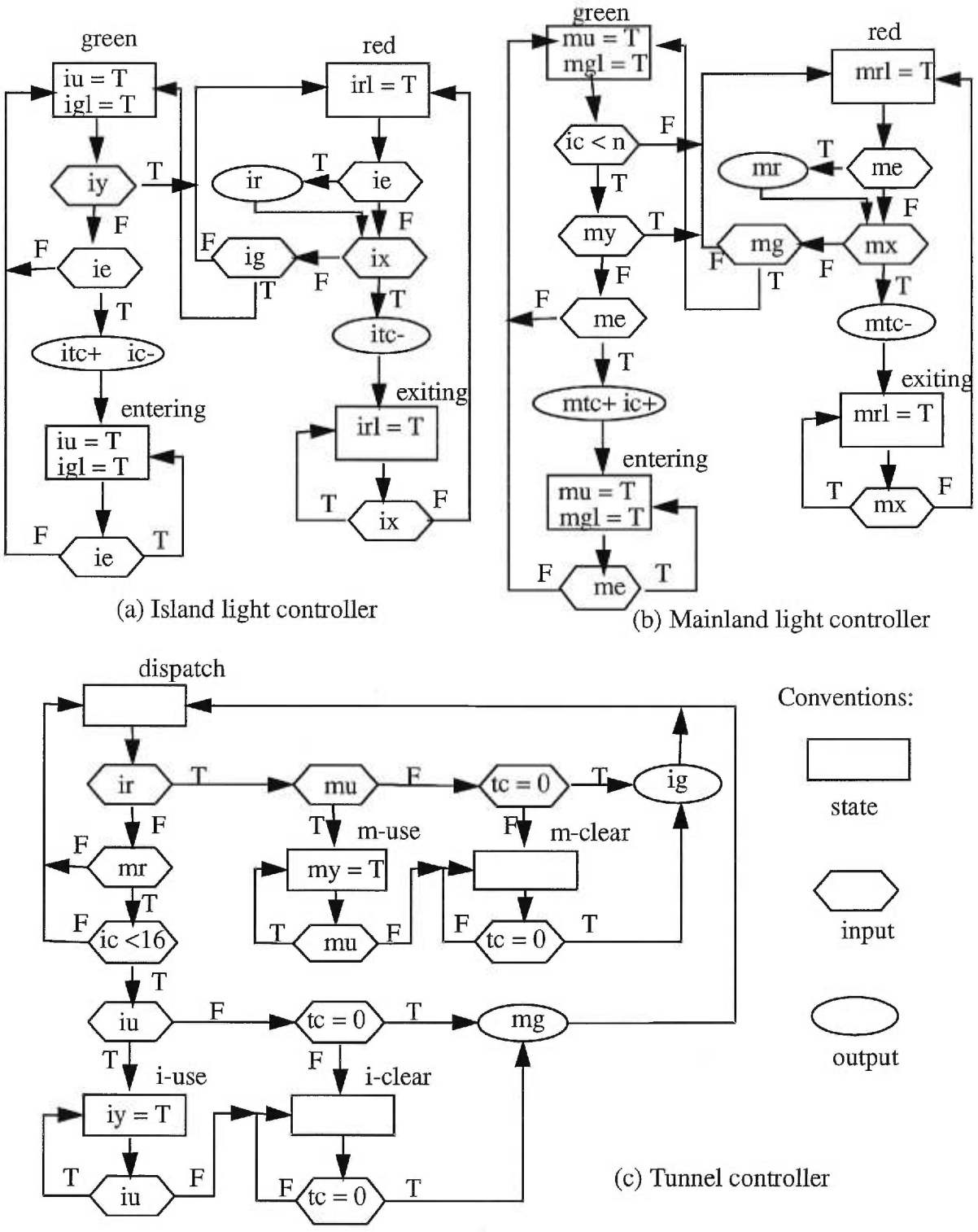



**Figure 32** - State Transition Graphs of the Island Tunnel Controller

### 10.1.2 Property checking using the MDG package

We first create an ASM model representing the ITC design which could be read by the MDG verification system. We created modules representing ILC, MLC, TC, and the counters as specified. All the signals are described using concrete variables, except that two state variables of abstract sort WORDN for $n$-bit word are used to describe the island counter (*ic*) and the tunnel counter (*tc*). The uninterprested function *inc* of type WORDN -> WORDN is used to describe the operation of incrementation by 1, and *dec* of the same type to describe the decrementation by 1. The environment (ENV) is built in such a way that it allows a non-deterministic choice of values on the primary inputs *ie*, *me*, *ix* and *mx*. Appendix A contains a listing of the ITC description in MDG-HDL, which is a language used for hardware description at the register transfer (RT) level. MDG-based symbolic reachability analysis requires 9 transition steps.

The following properties were verified on the ITC design:

**Property 1**: The lights at both entrances of the tunnel do not show green at the same time.

This is a typical safety property that a traffic light controller should satisfy. This property is described in the specification language $L_{\text{MDG}}$ as follows:

AG(! ((igl =1 ) & (mgl = 1)));

**Property 2**: The island counter is never ordered to increment and decrement simultaneously:

AG( ! ((ic- = 1) & (ic+ = 1)) );

**Property 3**. The tunnel counter behaves properly if ordered to increment and decrement simultaneously.

AG( ((tc+ = 1) &( tc- = 1)) -> (LET (v = tc) IN X (tc = v)) );

We used an ordinary variable *v* to remember the value of *tc* at the current state, and compare the value of *tc* at the next state with *v*. The property states that if both the signals *tc*+ and *tc*- are set, then the value of *tc* should not change from current state to the next state.

**Property 4**. The tunnel counter is never ordered to increment simultaneously by both the ILC and the MLC.

AG ( !((itc+ = 1) & (mtc- = 1)) );

Table 1 shows the CPU time and the memory used in building the composite machine and checking the simplified property regarding the signal *Flag* on the composite machine. The experiment was carried out on a SPARC Station 20 with 128 MB of memory.

**TABLE 1. Statistics for the ITC property verification in MDG.**

| Verification | Building the composite machine | | Checking the simplified property | |
|---|---|---|---|---|
| | CPU time (sec) | Memory (MB) | CPU time (sec) | Memory (MB) |
| Property 1 | 0.25 | 0.95 | 0.94 | 3.66 |
| Property 2 | 0.32 | 0.98 | 0.61 | 3.53 |
| Property 3 | 0.38 | 1.02 | 1.47 | 5.69 |
| Property 4 | 0.27 | 1.03 | 0.68 | 4.04 |

### 10.1.3 Property checking using VIS

Besides the ASM-based verification experiments, we also verified the same set of properties using VIS [83]. The same ITC behaviour model was recoded in a subset of Verilog HDL, accepted by VIS. However, since VIS is based on finite state machines, the counters *tc* and *ic* are now assigned concrete values which indicate the maximum number of cars that are allowed in the tunnel and on the island. We developed models according to the number of register bits used for the counters. For example, if 4 bits are used to describe *ic (tc)*, then the maximum of 16 cars are allowed on the island (in the tunnel). It takes 65 transition steps to compute all the reachable states when 4 bit counters are used. From Table 2, we can see that the number of transition steps increases when the counter width increases. Appendix B shows the ITC behavior model with 4 bit counters in Verilog HDL. The properties were described in CTL as follows:

**Property 1**: AG(!((igl=1 * mgl=1)));

**Property 2**: AG(!((ic_minus=1 * ic_plus=1)));

**Property 3**: In CTL, this property could be expressed as the conjunction of the following formulas. We have to enumerate all the possible values that *tc* could take, i.e., from 0 to 15.

AG( ((tc+ = 1) * ( tc- = 1) * (tc<0>=0 * tc<1>=0 * tc<2>=0 * tc<3>=0) )

-> (A X (tc<0>=0 * tc<1>=0 * tc<2>=0 * tc<3>=0) ) );

AG( ((tc+ = 1) * ( tc- = 1) * (tc<0>=1 * tc<1>=0 * tc<2>=0 * tc<3>=0) )

-> (A X (tc<0>=1 * tc<1>=0 * tc<2>=0 * tc<3>=0) ) );

.....

AG( ((tc+ = 1) * ( tc- = 1) * (tc<0>=1 * tc<1>=1 * tc<2>=1 * tc<3>=1) )

-> (A X (tc<0>=0 * tc<1>=0 * tc<2>=0 * tc<3>=0) ) );

**Property 4**. AG ( !((itc+ = 1) * (mtc- = 1)) );

Table 2 shows the CPU time and the memory used for verifying all the four properties on models with different counter widths. We also indicate the number of transition steps needed for the state exploration and the number of reachable states for the different models. The experiment was also carried out on a SPARC Station 20 with 128 MB of memory.

**TABLE 2. Statistics for the ITC property verification using VIS.**

| Counter Width | CPU time (sec) | Memory (MB) | Number of reachable states | Number of transition steps needed for state exploration |
|---|---|---|---|---|
| 4 bits | 4 | 5.67 | 59808 | 65 |
| 5 bits | 15 | 6.01 | 234400 | 129 |
| 6 bits | 46 | 6.70 | 927648 | 257 |
| 7 bits | 205 (3:25) | 8.35 | 3.69e+06 | 513 |
| 8 bits | 875(14:35) | 11 | 1.47e+07 | 1025 |
| 9 bits | 3097(51:37) | 22 | 5.88e+07 | 2049 |
| 10 bits | 12697(211:38) | 50 | 2.35e+08 | 4097 |

## 10.1.4 Discussion

From the experimental results shown in Tables 1 and 2, we can see that the MDG-based model checking can verify a parameterized implementation having $n$ bits, and it does so

very efficiently and independently of the datapath width. That is exactly the purpose behind the development of the ASM-based model checking methods. On the other hand, using the ROBDD-based tool VIS, the number of transition steps needed for state exploration and the number of states get doubled, and the resource usage (CPU time and memory) for the property verification increases exponentially with the counter width.

## 10.2 Verification of Properties of an Abstract Counter

In this section, we use the MDG-based model checker to verify both safety and liveness properties on a small design: an abstract counter which was introduced in [34]. The abstract counter was used in [34] as an example to show how formulas in Ground Temporal Logic can be used to describe state transitions and to specify design properties. Figure 33 shows the state transition graph of the counter. There are four control states: *c_Fetch*, *c_Load*, *c_Inc1*, and *c_Inc2*. Depending on the input, the counter *pc* will get a new value, or increase by one, or keep the previous value.

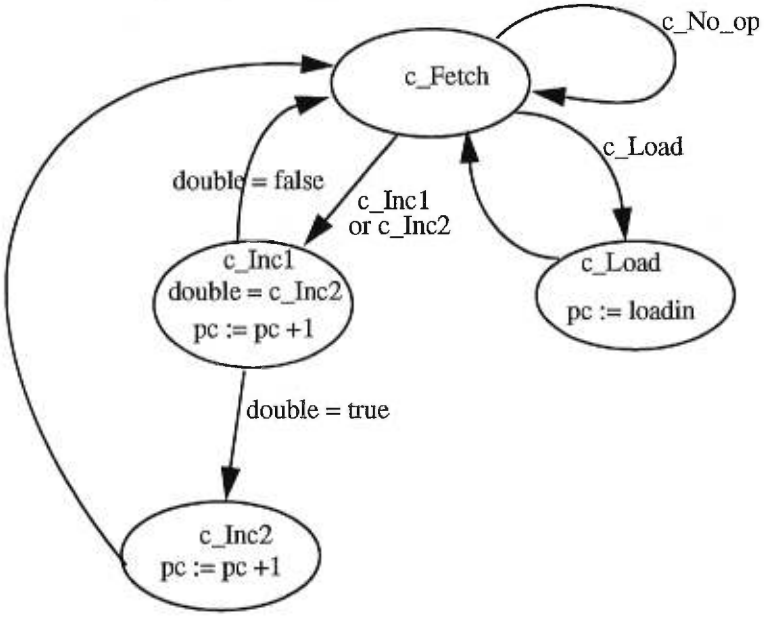



**Figure 33 -** An abstract counter

### 10.2.1 Property checking using the MDG package

To use our model checker, we first describe the behavior of the counter using the MDG-HDL language [80]. The description is shown in Appendix C. The counter *pc* is of abstract sort. The control state is initialized to *c_Fetch*, the initial value of *pc* is a free variable called *init_pc* (i.e., the initial state is generalized to any value). It takes 3 transition steps to compute all the reachable states. The following properties were verified:

**Property 1**: From state c_Fetch, if the input is c_Inc2, then the machine goes to the next state c_Inc1. This property is expressed in $L_{MDG}$ as follows:

AG( (state = c_Fetch & input = c_Inc2) -> (X(state = c_Inc1 )) );

**Property** 2: From state c_Fetch, if the input is c_Inc2, then the machine always reaches state c_Inc2 in two transition steps. This property is expressed in $L_{MDG}$ as follows:

AG( (state = c_Fetch & input = c_Inc2) -> (XX(state = c_Inc2 )) );

**Property** 3: From state c_Fetch, if the input is c_Inc2, then the machine reaches state c_Fetch in three transition steps and the counter pc has been increased by 2. This property is expressed in $L_{MDG}$ as follows:

AG( (state = c_Fetch & input = c_Inc2)
-> (LET (v1=pc) IN (XXX(state = c_Fetch & pc = finc(finc(v1)))) ) );

**Property** 4: From state c_Fetch, the machine will eventually reach state c_Load if the input is not c_No_op or c_Inc1 or c_Inc2 forever. The property is expressed in $L_{MDG}$ as:

AG( (state = c_Fetch) => (F(state = c_Load)));
under the following fairness constraint:
!( (state = c_Fetch)
-> ((input = c_Inc1) | (input = c_No_op) | (input = c_Inc2)) );

These properties were verified by our model checker using less than one second. Table 3 shows the CPU time in seconds used in building the composite machine and checking the simplified property regarding *Flag* on the composite machine. The experiment was carried out on a SPARC Station 20 with 128 MB of memory.

**TABLE 3. Statistics for the abstract counter verification in MDG.**

| Verification | Building the composite machine | | Checking the simplified property | |
|---|---|---|---|---|
| | CPU time (sec) | Memory (MB) | CPU time (sec) | Memory (MB) |
| Property 1 | 0.17 | 0.80 | 0.04 | 0.14 |
| Property 2 | 0.21 | 0.89 | 0.04 | 0.15 |
| Property 3 | 0.31 | 0.90 | 0.12 | 1.75 |
| Property 4 | 0.37 | 1.65 | 0.06 | 0.51 |

Using the decidable fragment of Ground Temporal Logic [34], Property 1, 2 and 3 could be checked, but Property 4 could not be verified since it is a liveness property. Using the "true symbolic model checking" [47], all the properties could be checked. But when verifying Property 3, as the abstract data *pc* appears in the property, we need to first strip the first-order parts in the formula to obtain a propositional formula G( (state = c_Fetch & input = c_Inc2) -> (XXX(state = c_Fetch ) ) ). After the propositional formula has been verified, a first-order verification condition need to be generated and verified. Using the ICS model [43][45], it happens that the abstract counter falls into the class of circuits where finite instantiation cannot be applied and thus it is not possible to compute all the reachable states; therefore, it appears that none of the above properties could be verified.

### 10.2.2 Property checking using VIS

To compare the performance of the MDG-based model checker to that of an FSM-based verification tool, and to partially verify the verification results, we carried out the same property verification using VIS. Again, for the counter *pc*, we have to give its upper

bound. We modeled the abstract counter in a subset of Verilog (see in Appendix D) using registers with different width for the counter *pc*, i.e., registers consisting of 4 bits, 8 bits, 16 bits, and 32 bits. On each model, we verified the same set of properties as in Section 10.2.1. The properties for the model with 4 bit *pc* register are stated in CTL as follows:

**Property 1**: AG( ((state = c_fetch) * (input_instruction = c_inc2))
-> (AX(state = c_inc1)));

**Property 2:** AG( ((state = c_fetch) * (input_instruction = c_inc2))
-> (AX(AX(state = c_inc2))));

**Property** 3: AG(((state = c_fetch) * (input_instruction = c_inc2)
* (pc<3> = 0 * pc<2> = 0 * pc<1> = 0 * pc<0> = 0))
-> (AX(AX(AX((state = c_fetch)
* (pc<3> = 0 * pc<2> = 0 * pc<1> = 1 * pc<0> = 0))))) );
with (pc<3> pc<2> pc<1> pc<0> ) ranging over from 0000 to 1111;

**Property** 4: AG((state = c_fetch) -> (AF(state = c_load)));
under the following fairness constraint:
!( (state = c_Fetch) -> ((input = c_Inc1) | (input = c_No_op) | (input = c_Inc2)) );

Table 4 shows the number of transitions it takes for each model to compute all the reachable states, the number of the reachable states, the CPU time, and the memory needed to verify Properties 1 to 4.

**TABLE 4. Statistics for the abstract counter verification using VIS.**

| Counter Width | CPU time (sec) | Memory (MB) | Number of reachable states | Number of transition steps needed for state exploration |
|---|---|---|---|---|
| 4 bits | 0.56 | 2.84 | 448 | 6 |
| 8 bits | 3 | 3.72 | 7168 | 6 |
| 16 bits | 7 | 4.80 | 1.83501e+06 | 6 |
| 32 bits | 12 | 6.12 | 1.20259e+11 | 6 |

### 10.2.3 Discussion

The statistics shown in Tables 3 and 4 again demonstrate that the MDG-based model checking can verify both safety and liveness properties on a parameterized implementation independent of the data path width very efficiently. However, from Table 4, we can see that with the counter width increasing, the number of reachable states increases exponentially, but the number of transition steps needed for state exploration stays the same and the usage of CPU time and memory only increases slightly, which was not the case in the Island Tunnel Controller. The reason is that in this particular example, the counter *pc* is independent of the state transitions, i.e., the state transitions are not gated by the value of *pc*. Every time when loading in a new value of *pc* it can take any value within its range, hence, the node *pc* will not appear in the BDD expression of the sets of states. Therefore, no matter how large the width of *pc* is, the time and memory usage will not grow significantly. Nevertheless, the MDG-based model checking still outperforms the ROBDD-based model checker in the sense that one ASM model of the Abstract Counter and one set of properties automatically cover all the possible *pc* widths. Using VIS on the other hand, we have to build separate models and to develop separate sets of properties for *pc* instances of different widths.

# Summary

In this chapter, we performed property verifications on two examples: the Island Tunnel Controller and the Abstract Counter. We illustrated how safety and liveness properties can be described in $L_{\text{MDG}}$. Using MDG_based model checking, we were able to use only one abstract variable instead of a number of Boolean variables for representing a data value. Hence, the performance of the MDG-based model checker was better than that of the ROBDD-based model checker when there is a data path involved in the design.

# 11 Conclusions and Future Work

## 11.1 Conclusions

BDD-based symbolic model checking has proven to be a successful verification technique that can be applied to real life designs. However, since it requires the design to be described at the Boolean logic level, the state explosion problem caused by large datapath is often the bottleneck in applying symbolic model checking technique.

In this thesis, we studied model checking for a first-order temporal logic based on the Abstract description of State Machines (ASMs). Since a data value is represented by a single variable of abstract type, rather by a vector of Boolean variables, and a data operation is represented by an uninterpreted function symbol, the width of a datapath of a design has no affect to the description model of the design. We can then alleviate the state explosion problem in symbolic model checking caused by a large datapath.

We defined a very general first-order branching-time temporal logic: Abstract_CTL*. We then defined $L_{\mathrm{MDG}}$, a subset of Abstract_CTL*, as the property

specification language and developed property checking algorithms for $L_{MDG}$. To check a property of $L_{MDG}$ on an ASM $M$, we first build additional ASMs for all the Next_let_formulas (which contain the temporal operator X) that appear in the property. Then we compose the additional ASMs with $M$, and finally verify a simpler property on the composite machine. We only allow universal path quantifier and limited nesting of temporal operators (other than X) in $L_{MDG}$, however, useful safety and liveness properties can be expressed with or without fairness constraints. We use MDGs to encode sets of states and the transition relations. The property checking procedures are based on implicit state enumeration and are carried out fully automatically. We have also demonstrated the soundness of our verification procedures.

We have implemented a parser in the C language using Yacc and Lex to check the property specification and to automatically build ASMs for the Next_let_formulas. All the model checking algorithms were implemented using the MDGs operations implemented in Quintus Prolog V3.2.

We illustrated the application of our model checker on the Island Tunnel Controller and the Abstract Counter benchmarks. The experimental results demonstrate that the MDG-based model checking can verify both safety and liveness properties on parameterized implementations independent of the data path width very efficiently.

## 11.2 Future work

The ASMs-based model checking for a first-order temporal logic presented in this thesis showed its potential of automatically verifying properties on designs with large data path. However, there are areas in which the present work could be improved or extended. Listed below are some of the future research directions:

- Developing a counter-example facility:

A counter-example facility showing a trace from the initial state to a state causing the property to fail will certainly make debugging of designs a lot easier. Several ROBDD-based model checkers possess this feature [86][83]. In our MDG package, states are described in DFs represented by MDGs. When abstract state variables are involved, we cannot distinguish one computation path from another during the state exploration, since even a one-disjunct DF represents a set of states. Hence, we cannot provide the same counter-example facility as an ROBDD-based model checker can.

However, in the MDG package, when a property fails, it is possible to provide traces from a set of initial states to sets of states in which the property was not satisfied. One possible solution is as follows: when checking a property, in addition to computing the next states using **RelP**, we add a DF $S_{total}$ (initialized to the initial set of states) to record the current states, all the previous states, and the inputs. This can be achieved by adding the following two statements within the iteration of each property checking algorithm: $S_{total}$ = **RelP**($\{I, pre\text{-}S_{total}, G_T\}$, {}, $\eta_{new}$ ); $pre\text{-}S_{total} = S_{total}$; where $I$ are the inputs, $G_T$ the transition relation, and $\eta_{new}$ a renaming function which substitutes $Y'$ to $Y_n$ ($n$ is iteration number). If a safety property (in the template of **AG, AX**) fails, a DF **Conj**($S_{total}$, $\{Flag_n = 0\}$) contains several counter-example traces. It a liveness property fails, a DF **Conj**($S_{total}$, $Flag_n = 0$) when the property is in the template of **AF** ( or a DF **Conj**($S_{total}$, $FlagQ_n = 0$) when the property is in the template of **AU**), and the DF $T$ (when if $\exists\ T \in \Sigma$, **PbyS**($T$, $Q$) = **F** is detected) should help the user to find the cycles along which $Flag = 1$ (or $FlagQ = 1$) never becomes true.

- Link to theorem provers:

Combining both theorem proving and model checking to resolve the verification tasks involving large designs becomes an interesting topic in the formal verification application community[71]. Theorem proving can be used at a higher level of abstraction than model checking and can augment the verification coverage of the design hierarchically. In a sense, model checking can be used to verify the low-level

modules until it cannot go higher in the hierarchy.

It is thus desirable to explore the linkages between MDG tools and a theorem prover (e.g., HOL). The MDG-based model checker can be used as a decision procedure in a theorem proving system. Namely, when using the theorem prover to verify a large goal, some of the sub-goals or lemmas could be proved using the MDG-based model checker.

- Experimental verification of the method using industrial and academic benchmarks:

  The MDG-based verification package (including the model checker) used variables of abstract type to represent data and uninterpreted functions to describe data operation. The data width is no longer the bottleneck to cause the state explosion. This is ideal for verification of designs with large data path. It would be valuable to test a large number of industrial scale designs with large data path (most telecommunication circuits happen to fall into this category) and academic benchmarks using the MDG-based model checker, in order to evaluate and to improve its performance.

- Solving the non-termination problem:

  Some early research has shown that two approaches could solve the non-termination problem in some situations. The first one is based on the use of ρ-terms which can finitely represent infinite sets of state [64]. An extension of the syntax of MDGs and MDG-based algorithms could incorporate ρ-terms to solve the non-termination problem when the generated set of states exhibit certain repetitive patterns. The second approach is to modify the original ASM structural description according to certain rules to avoid the non-termination problem[65]. It would be valuable to explore a more general method that could automatically analyze the ASM description, modify the design description and infer ρ-terms. Furthermore, implementing these ideas in the package would extend the applicability of the MDG-based verification techniques.

- Automatic node ordering:

  It is possible to develop an automatic node ordering procedure based on the current variable ordering heuristics for the MDG package and the experience from ROBDDs and other decision graphs [19][35][36][66][70]. Automatic node ordering would make the MDG package easier to use and improve the performance of the model checking procedures.

# Bibliography


[1] C. M. Angelo, D. Verkest, L. Claesen, H. De Man. On the Comparison of HOL and Boyer-Moore for Formal Hardware Verification. In Journal *Formal Methods in System Design*, vol 2: pp. 45-72, 1992.

[2] K.D. Anon, N. Boulerice, E. Cerny, F. Corella, M. Langevin, X. Song, S. Tahar, Y. Xu, Z. Zhou. MDG Tools for the Verification of RTL Designs. In *Proceedings of Conference on Computer-Aided Verification* (CAV'96). New Jersey, USA, July 1996.

[3] J. P. Billon and J. C. Madre. Original concepts of PRIAM, an industrial tool for efficient formal verification of combinational circuits. In *Fusion of Hardware Design and Verification*, G. J. Milne (ed.). pp. 487-501. North-Holland, Amsterdam, 1988.

[4] G.V. Bochmann, Hardware specification with temporal logic: An example. In *IEEE Transactions on Computers*, C-31(3): pp.223-231, March 1982.

[5] J. Bormann, J. Lohse, M. Payer, G. Venzl. Model Checking in Industrial Hardware Design. In *Proceedings of the 32th Design Automation Conference (DAC'95)*. June 1995.

[6] S. Bose and A.L. Fisher. Automatic verification of synchronous circuits using symbolic simulation and temporal logic. In *Proceedings of the IFIP International Workshop on Applied Formal Methods for Correct VLSI Design*, Leuven, Belgium, 1989, L.J.M. Claesen, (ed), pp.759-764. North-Holland, Amsterdam, 1990.

[7] M.C. Browne, E. M. Clarke, D.L. Dill and B. Mishra. Automatic Verification of Sequential Circuits Using Temporal Logic. In *IEEE Transactions on Computers*, December 1986.

[8] R.S. Boyer and J.S.Moore. A Computational Logic Handbook. Academic Press, Boston, 1998.

[9] R.K. Brayton et. al. VIS: A system for verification and synthesis. Technical Report. UCB/ERL M95. Electronics Research Lab, University of California, Berkeley. December 1995.

[10] R.K. Brayton et. al. VIS: A system for Verification and Synthesis. In the *Proceedings of the 8th International Conference on Computer Aided Verification*, pp.428-432, Springer Lecture Notes in Computer Science, #1102, Edited by R. Alur and T. Henzinger, New Brunswick, NJ, July 1996.

[11] R. E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Transactions on Computers*, 35(8):pp.677-691, August 1986.

[12] R.E. Bryant and C.-J.H. Seger. Formal verification of digital circuits using symbolic ternary system models. In *Proceedings of the Workshop on Computer-Aided Verification (CAV 90)*, E.M. Clarke and R.P. Kurshan (eds.). volume 3 of *DIMACS Series in Discrete Mathematics and Theoretical Computer Science*. American Mathematical Society, Spring-Verlag, New York, NY, 1991.

[13] R.E. Bryant, D. Beatty, K. Brace, K. Cho, and T. Sheffler. COSMOS: A compiled simulator for MOS circuits. In *Proceedings of the 24th ACM/IEEE Design*

*Automation Conference*, pp.9-16. IEEE Computer Society Press, Los Alamitos, CA, June 1987.

[14] J. R. Burch, E. M. Clarke, and D. E. Long. Symbolic model checking with partitioned transition relations. In VLSI 91, Edinburgh, Scotland, 1990.

[15] J. R. Burch, E. M. Clarke, D. E. Long, K. L. McMillan, and D. L. Dill. Symbolic model checking for sequential circuit verification. In *IEEE Transactions on Computer-Aided Design*, 13(4):401-424, April 1994.

[16] J. R. Burch, E.M. Clarke, and K. L. McMillan. Symbolic model checking: $10^{20}$ States and Beyond. In *Proceedings of LICS*, 1990.

[17] J. Burch, E.M. Clarke, K.L. McMillan, and D.L.Dill. Sequential circuit verification using symbolic model checking. In *Proceedings of the 27th ACM/IEEE Design Automation Conference*, pp.46-51. IEEE Computer Society Press, Los Alamitos, CA, June 1990.

[18] J.R. Burch and D.L. Dill. Automatic Verification of Pipelined Microprocessor Control. In *Computer Aided Verification. 6th International Conference,*1994.

[19] Ney Calazans, Q. Zhang, R. Jacobi, B. Yernaux and A.M.Trullemans. Advanced Ordering and Manipulation Techniques for Binary Decision Diagrams. In *Proceedings of the 29th Design Automation Conference (DAC'29),* June 1992.

[20] A.J. Camilleri, M.J.C. Gordon, and T.F. Melham. Hardware verification using higher-order logic. In *From HDL Descriptions to Guaranteed Correct Circuit Designs*, pp.43-67. D. Borrione(ed.). North-Holland, Amserdam, 1987.

[21] B. Chen, M. Yamazaki, M. Fujita. Bug Identification of a Real Chip Design by Symbolic Model Checking. In *Proceedings of International Symposium on Circuits and Systems (ISCAS'94),* 1994.

[22] E.M. Clarke and E. A. Emerson. Design and Synthesis of synchronization skeletons using branching time temporal logic. In *Proceedings of the Workshop on Logics of Programs*, Volume 131 of *Lecture Notes in Computer Science*. Springer-Verlag, pp. 52-71. New York, 1981.

[23] E.M. Clarke and E. A. Emerson. Synthesis of synchronization skeletons for branching time temporal logic. In Dexter Kozen, editor, *Logic of Programs: Workshop*, volume 131 of *Lecture Notes in Computer Science*, Yorktown Heights, New York, May 1981. Springer-Verlag.

[24] E.M. Clarke, E. A. Emerson, and A.P. Sistla. Automatic verification of finite state concurrent systems using temporal logic specifications. *ACM transactions on Programming Languages and Systems*, 8(2):pp.244-263 (April 1986).

[25] E. M. Clarke, O. Grumberg, and D. E. Long. Model checking and abstraction. In *Proceedings of the Nineteenth Annual ACM Symposium on Principles of Programming Languages*, January 1992.

[26] R. L. Constable, et al. Implementing Mathematics with the Nuprl Proof Development System. PrenticeHall, Englewood cliffs, New Jersy,1986.

[27] F. Corella, M. Langevin, E. Cerny, Z.Zhou, X. Song. State enumeration with abstract descriptions of state machines. In *Proceedings IFIP WG10.5 Advanced Research Working Conference on Correct Hardware Design and Verification Methods(Charme'95)*, Frankfurt, Germany, October 1995.

[28] F. Corella, Z.Zhou, X. Song, M. Langevin, E. Cerny. Multiway decision graphs for automated hardware verification. *Formal Methods in System Design*. 10(1): 7-46, February 1997.

[29] O. Coudert, C. Berthet, and J. C. Madre. Verification of synchronous sequential machines using symbolic execution. In *Proceedings of the International Workshop on Automatic Verification Methods for Finite State Systems*, Grenoble, France,

volume 407 of Lecture Notes in Computer Science. pp. 365 - 373. Spring-Verlag, New York, 1989.

[30] O. Coudert, C. Berthet, and J. C. Madre. Verification of sequential machines using boolean functional vectors. In L. Claesen, editor, *Proceedings IFIP International Workshop on Applied Formal Methods for Correct VLSI Design,* pp. 111--128, Leuven, Belgium, November 1989. North-Holland.

[31] O. Coudert, J.C. Madre, and C. Berthet. Verifying temporal properties of sequential machines without building their date diagrams. In *Proceedings of the Workshop on Computer-Aided Verification (CAV 90)*, E.M. Clarke and R.P. Kurshan (eds.). volume 3 of *DIMACS Series in Discrete Mathematics and Theoretical Computer Science*. American Mathematical Society, Spring-Verlag, New York, NY, 1991.

[32] O. Coudert and J. C. Madre. A unified framework for the formal verification of sequential circuits. In *Proceedings of International Conference on Computer-Aided Design (ICCAD'90)*, 1990.

[33] Paul Curzon. The formal verification of the Fairisle ATM switching element. Technical Report No.328, No.329, University of Cambridge Computer Laboratory, 1994.

[34] D. Cyrluk and P. Narendran. Ground Temporal Logic: A logic for hardware verification. In *Proceedings on Computer-Aided Verification*, 1994.

[35] Rolf Drechsler, Nicole Drechsler, Wolfgang Günther. Fast Exact Minimization of BDDs. In *Proceedings of the 34th Design Automation Conference (DAC'98),* pp.200. San Francisco, California. June 1998.

[36] Rolf Drechsler, Wolfgang Günther. Linear Transformations and Exact Minimization of BDDs. In *Great Lakes Symposium on VLSI (GLSV'98)*, pp. 325-330. Lafayette, 1998.

[37] E. Allen Emerson. Temporal and Modal Logic, 16th chapter of *HANDBOOK OF THEORETICAL COMPUTER SCIENCE*, edited by J.van Leeuwen. Elsevier Science Publishers B.V., 1990.

[38] E. A. Emerson, C. L. Lei. Modalities for Model Checking: Branching Time Logic Strikes Back. Science of Computer Programming 8, pp. 275-306, Elsevier Science Publishers, 1987.

[39] K. Fisler and S. Johnson. Integrating design and Verification Environments Through A Logic Supporting Hardware Diagrams. In *Proceedings of IFIP Conference on Hardware Description Languages and their Applications (CHDL'95)*. August 1995, Chiba, Japan.

[40] M. J. C. Gordon and J. Herbert. Formal hardware verification methodology and its application to a network interface chip. *IEE Proceedings,* 133, Part E(5): pp.255-270 (September 1986).

[41] M. J. C. Gordon, T. F. Melham. Introduction to HOL: A theorem proving environment for higher order logic. Cambridge University Press, Cambridge, UK, 1993.

[42] A. Gupta. Formal Hardware Verification Methods: A Survey. In Journal *Formal Methods in System Design*, vol 1, pp. 151-238 (1992).

[43] R. Hojati, R. K. Brayton. Automatic datapath Abstraction In Hardware Systems. In *Proceedings of Conference on Hardware Description Language (CHDL'95),* Tokyo, Japan, August 1995.

[44] R. Hojati, A. Isles, D. Kirkpatrick, R. K. Brayton, Verification Using Uninterpreted Functions and Finite Instantiations, Formal Methods in Computer-Aided Design (FMCAD), November 1996.

[45] R. Hojati, D. L. Dill, R. K. Brayton. Verifying linear temporal properties of data insensitive controllers using finite instantiations. In *Proceedings of IFIP Conference on Hardware Description Languages and their Applications (CHDL'97)*. Spain, April 1997.

[46] J. E. Hopcroft and J. D. Ullman. *Introduction to Automata Theory, Languages and Computation*. Addison-Wesley, Reading, March 1979.

[47] H. Hungar, O. Grumberg, and W. Damm. What if Model Checking Must Be Truly Symbolic. In *Workshop on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'95)*, Aarhus, Denmark, May 1995.

[48] W. A. Hunt, Jr. FM8501: A verified microprocessor. Ph.D thesis, Technical Report ICSCA-CMP-47, University of Texas at Austin, 1985.

[49] W. A. Hunt, Jr. Microprocessor design verification. *Journal of Automated Reasoning*, 5(4): 429-460 (1989).

[50] A. Isles, R. Hojati, R. K. Brayton. Reachability Analysis of ICS Models, SRC Techcon, September 1996.

[51] Jae-Young Jang, Shaz Qadeer, Matt Kaufmann, Carl Pixley. Formal Verification of FIRE: A Case Study. In *Proceedings of the 34th Design Automation Conference (DAC'97)*. Anaheim, CA, June 1997.

[52] P. B. Jackson. Nuprl and its Use in Circuit Design. In *Proceedings of the IFIP TC10/WG10.2 International Conference on Theorem Provers in Circuit Design: Theory, Practice and Experience*, V. Stavridou, T.F. Melham and R.T. Boute (Editors), pp. 311-336, North-Holland, The Netherlands, 1992.

[53] P. B. Jackson. The Nuprl Proof Development System, Version 4.1 Reference Manual and User's Guide. Cornell University, Ithaca, NY, 1994.

[54] J. Joyce, G. Birtwistle, and M. Gordon. Proving a computer correct in higher order logic. Technical Report 100, University of Cambridge, Computer Laboratory, December 1986.

[55] R. P. Kurshan. Reducibility in Analysis of Coordination. In LNCIS, volume 103, pp. 19-39, Springer-Verlag, 1987.

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

[57] R. P. Kurshan. Formal Verification in a Commercial Setting. In *Proceedings of the 34th Design Automation Conference (DAC'97)*. Anaheim, California. July 1997.

[58] O. Lichtenstein and A. Pnueli. Checking that finite state concurrent programs satisfy their linear specifications. In *Proceedings of the Twelfth Annual ACM Symposium on Principles of Programming Language*, pp. 196-218. ACM, New York, 1985.

[59] Z. Manna and P. Wolper. Synthesis of communicating processes from temporal logic specifications. *ACM Transactions on Programming Languages and Systems*, 6:pp.68-93 (1984).

[60] M. C. McFarland. Formal Verification of Sequential Hardware: A Tutorial. *IEEE Transaction on Computer-Aided Design of Integrated Circuits and Systems*. Vol.12, No.5, May 1993.

[61] K. L. McMillan. *Symbolic model checking, An approach to the state explosion problem*. Ph.D. thesis, School of Computer Science, Carnegie Mellon University, Pittsburgh, PA,1992.

[62] K. L. McMillan and J. Schwalbe. Formal verification of the Encore Gigamax cache consistency protocol. In *Proceedings of the International Symposium on Shared Memory Multiprocessing*, 1991. pp.242-251.

[63] S. Owre, N. Shankar and J. Rushby. PVS: A Prototype Verification System. In Proc of 11th International Conference on Automated Deduction. D. Kapur, Ed. Saratoga, NY, 1992.

[64] A. Otmane, X. Song, E. Cerny. On the non-termination of MDG-based abstract state enumeration. In *Proceedings of IFIP International Conference on Correct Hardware Design and Verification Methods, (CHARME'97)*, pp.218-235, Montreal, Canada, 1997.

[65] A. Otmane, E. Cerny, X. Song. MDGs-based Verification by Retiming and Combinational Transformations. In *Proceedings of the IEEE 8th Great Lakes Symposium on VLSI*, Louisiana, USA, 1998.

[66] S.Panda, F.Somenzi. Who are the variables in your neighbourhood? In *Proceedings of International Conference on Computer-Aided Design (ICCAD'*95), 1995.

[67] A. Pnueli. Applications of temporal logic to the specification and verification of reactive systems: A survey of current trends. In *Current Trends in Concurrency*, J.W. de Bakker, W.-P.de Roever, and G. Rozenberg (eds.), volume 224 of *Lecture Notes in Computer Science*, pp. 510-584. Springer-Verlag, New York, 1986.

[68] A. Pnueli. The temporal logic of programs. In *Proceedings of the Eighth Annual Symposium on Foundations of Computer Science,* pp. 123-144. IEEE, New York, 1984.

[69] J. P. Quielle and J. Sifakis. Specification and verification of concurrent systems in CESAR. In *Proceedings of the Fifth International Symposium in Programming*, 1981.

[70] Rajeev K. Ranjan, Wilsin Gosti . Speeding up Variable Reordering of OBDDs. In *Proceedings of International Conference on Computer Design (ICCD'97),* Austin,TX, USA 1997.

[71] C. Seger. Combining Theorem Proving and Model Checking: How Much Theorem Proving Is Needed? Invited talk. In *Proceedings of Conference on Computer-Aided Verification* (CAV'98). Vancouver, BC, Canada, July 1998.

[72] C. Seger and R. E. Bryant, "Formal Verification by Symbolic Evaluation of Partially-Ordered Trajectories", UBC Department of Computer Science Technical Report 93-8, April 1993.

[73] A. P. Sistla and E.M. Clarke. Complexity of propositional linear temporal logic. *Journal of the ACM*, 32(3): 733-749 (July 1985).

[74] S. Tahar and R. Kumar: Implementing a Methodology for Formally Verifying RISC Processors in HOL (Higher Order Logic); In: Joyce, J. and Seger, C. (Eds.), Higher Order Logic Theorem Proving and Its Applications, Lecture Notes in Computer Science 780, pp. 281-294. Springer Verlag, 1994.

[75] G. Thuau, B. Berkane. A Unified Framework for Describing and Verifying Hardware Synchronous Sequential Systems. *Formal Methods in System Design*, vol 2: pp 259-276, 1993.

[76] H. J. Touati, H. Savoj, B. Lin, R. K. Brayton, and A. Sangiovanni-Vincentelli. Implicit state enumeration of finite state machines using BDDs. In International Conference on Computer-Aided Design, 1990.

[77] Y. Xu, E. Cerny, A. Silburt, and R. B. Hughes. Property Verification Using Theorem Proving and Model Checking. *Integrated System Design*, November 1997.

[78] Y. Xu, E. Cerny, X. Song, F. Corella, O. Ait Mohamed. Model Checking for a First-Order Temporal Logic using Multiway Decision Graphs. In *Proceedings of Conference on Computer-Aided Verification* (CAV'98). Vancouver, BC, Canada, July 1998.

[79] Z. Zhou. *Multiway Decision Graphs and Their Applications in Automatic Formal Verification of RTL Designs*. PhD thesis, D'IRO, University of Montreal, 1997.

[80] Z. Zhou and N. Boulerice. MDG Tools (V1.0) User's Manual. D'IRO, University of Montreal, June 1996.

[81] Z. Zhou, X. Song, F. Corella, E. Cerny and M. Langevin. Description and verification of RTL designs using Multiway Decision Graphs. In *Proceedings of the Conference on Computer Hardware Description Languages and their applications* (CHDL'95). Chiba, Japan. August, 1995.

[82] Z. Zhou, X. Song, S. Tahar, F. Corella, E. Cerny and M. Langevin. Formal verification of the island tunnel controller using multiway decision graphs. In *Proceedings of International Conference on Formal Methods in Computer Aided Design (FMCAD'96)*, pp. 233 - 247, Palo Alto, CA, USA, 1996.

[83] The VIS Group web page. VIS: Verification Interacting with Synthesis, 1995. http://www-cad.eecs.berkeley.edu/Respep/Research/vis.

[84] CMU - School of Computer Science Formal Methods - Model Checking web page. http://www.cs.cmu.edu/~modelcheck/

[85] FormalCheck web page. http://www.bell-labs.com/org/blda/product_formal.html

[86] FormalCheck User's Guide. Bell labs Design Automation, Lucent Technologies, V1.1, 1997.

[87] CheckOff User Guide, Siemens Nixdorf Informations Systemen AG & Abstract Hardware Limited, January, 1996.

# Appendix A - ITC behavioral description in MDG-HDL

```
%-----------------------------------------------
% File:  itc_ret_s.pl
% Title: ITC specification.
%-----------------------------------------------

% Multifile declaration required by the Prolog system.
%
:- multifile component/2.
:- multifile signal/2.
:- multifile next_state_partition/1.
:- multifile output_partition/1.
:- multifile init_val/2.
:- multifile init_var/2.
:- multifile init_con/2.
:- multifile st_nxst/2.
:- multifile outputs/1.
:- multifile par_strategy/2.

%======================== Inputs and Outputs ======================

%--- Input signals---
%
signal(ie,bool).
signal(ix,bool).
signal(me,bool).
signal(mx,bool).

%--- Outputs ---
signal(irl_A,bool).
signal(igl_A,bool).
signal(itc_plus_A,bool).
signal(itc_min_A,bool).
signal(ic_min_A,bool).

signal(mrl_A,bool).
signal(mgl_A,bool).
signal(mtc_plus_A,bool).
signal(mtc_min_A,bool).
signal(ic_plus_A,bool).
```

```
signal(tc_plus_A,bool).
signal(tc_min_A,bool).

%--- Outputs ---

signal(ie,bool).
signal(ix,bool).
signal(me,bool).
signal(mx,bool).

%======================== Island Light Controller ====================

%--- Input signals---
%
signal(ig_A,bool).
signal(iy_A,bool).

%--- Outputs ---
%
signal(ir_A,bool).
signal(iu_A,bool).

%--- State variables---
%
signal(is_A,mi_sort).

%----- Behavioral description for the island light controller----
%
component(is_comp_A,table([[is_A,ig_A,iy_A,ie,ix,n_is_A],
                           [green,*,0,0,*,green],
                            [green,*,0,1,*,entering],
                            [green,*,1,*,*,red],
                            [entering,*,*,0,*,green],
                            [entering,*,*,1,*,entering],
                            [red,  0,*,*,0,red],
                            [red,  1,*,*,0,green],
                            [red,  *,*,*,1,exiting],
                            [exiting, *,*,*,0,red],
                            [exiting, *,*,*,1,exiting]])).

component(ir_comp_A,table([[is_A,ie,ir_A],
                           [red,1,1]|0])).
```

```
component(irl_comp_A,table([[is_A,irl_A],
                            [red,1],
                            [exiting,1]|0])).

component(igl_comp_A,table([[is_A,igl_A],
                            [green,1],
                            [entering,1]|0])).

component(iu_comp_A,table([[is_A,iu_A],
                            [green,1],
                            [entering,1]|0])).

component(itc_plus_comp_A,table([[is_A,iy_A,ie,itc_plus_A],
                            [green,0,1,1]|0])).

component(itc_minus_comp_A,table([[is_A,ix,itc_min_A],
                            [red,1,1]|0])).

component(ic_min_comp_A,table([[is_A,iy_A,ie,ic_min_A],
                            [green,0,1,1]|0])).


%================ Mainland Light Controller ====================

%--- Input signals---
%
signal(mg_A,bool).
signal(my_A,bool).

%--- Outputs ---
%
signal(mr_A,bool).
signal(mu_A,bool).

%--- State variables---
%
signal(ms_A,mi_sort).

%----- Behavioral description for the mainland light controller----
%
component(ms_comp_A,table([[ms_A,mg_A,my_A,me,mx,lessn(ic_A),n_ms_A],
                            [green,*,*,*,*,0,red],
                            [green,*,0,0,*,1,green],
                            [green,*,0,1,*,1,entering],
```

```
                                    [green,*,1,*,*,1,red],
                                    [entering,*,*,0,*,*,green],
                                    [entering,*,*,1,*,*,entering],
                                    [red,  0,*,*,0,*,red],
                                    [red,  1,*,*,0,*,green],
                                    [red,  *,*,*,1,*,exiting],
                                    [exiting, *,*,*,0,*,red],
                                    [exiting, *,*,*,1,*,exiting]])).

component(mr_comp_A,table([[ms_A,me,mr_A],
                                    [red,1,1]|0])).

component(mrl_comp_A,table([[ms_A,mrl_A],
                                    [red,1],
                                    [exiting,1]|0])).

component(mgl_comp_A,table([[ms_A,mgl_A],
                                    [green,1],
                                    [entering,1]|0])).

component(mu_comp_A,table([[ms_A,mu_A],
                                    [green,1],
                                    [entering,1]|0])).

component(mtc_plus_comp_A,table([[ms_A,my_A,me,lessn(ic_A),mtc_plus_A],
                                        [green,0,1,1,1]|0])).

component(mtc_minus_comp_A,table([[ms_A,mx,mtc_min_A],
                                        [red,1,1]|0])).

component(ic_plus_comp_A,table([[ms_A,my_A,me,lessn(ic_A),ic_plus_A],
                                        [green,0,1,1,1]|0])).

%======================== Tunnel Controller ======================
/*
%--- Input signals---
%
signal(ir_A,bool).
signal(iu_A,bool).

signal(mr_A,bool).
signal(mu_A,bool).
```

```
%--- Outputs ---
%
signal(ig_A,bool).
signal(iy_A,bool).

signal(mg_A,bool).
signal(my_A,bool).
*/
%--- State variables---
%

signal(ts_A,ts_sort).

%----- Behavioral description for the tunnel controller----
%
component(ts_comp_A,
table([[ts_A,ir_A,mr_A,lessn(ic_A),eqz_ret1,iu_A,mu_A,n_ts_A],
        [dispatch,0,0,*,*,*,*,dispatch],
        [dispatch,0,1,0,*,*,*,dispatch],
        [dispatch,0,1,1,*,1,*,iuse],
        [dispatch,0,1,1,0,0,*,iclear],
        [dispatch,0,1,1,1,0,*,dispatch],
        [dispatch,1,*,*,0,*,0,mclear],
        [dispatch,1,*,*,1,*,0,dispatch],
        [dispatch,1,*,*,*,*,1,muse],
        [iuse,*,*,*,*,0,*,iclear],
        [iuse,*,*,*,*,1,*,iuse],
        [muse,*,*,*,*,*,*,0,mclear],
        [muse,*,*,*,*,*,*,1,muse],
        [iclear,*,*,*,0,*,*,iclear],
        [iclear,*,*,*,1,*,*,dispatch],
        [mclear,*,*,*,0,*,*,mclear],
        [mclear,*,*,*,1,*,*,dispatch]])).

component(ig_comp_A,table([[ts_A,ir_A,eqz_ret1,mu_A,ig_A],
                           [dispatch,1,1,0,1],
                           [mclear,  *,1,*,1]|0])).

component(iy_comp_A,table([[ts_A,iy_A],
                           [iuse,1]|0])).

component(mg_comp_A,
table([[ts_A,ir_A,mr_A,lessn(ic_A), eqz_ret1,iu_A,mu_A,mg_A],
        [iclear,    *,*,*,1,*,*,1],
```

```
        [dispatch,  0,1,1,1,0,*,1]|0])).

component(my_comp_A,table([[ts_A,my_A],
                           [muse,1]|0])).

%----- Behavioral description for Counters -------
%
%inputs and outputs added
signal(eqz_s1,bool).
signal(eqz_ret1,bool).
signal(s1,wordn).
signal(tc_A_s1,wordn).
signal(sig1,bool).
signal(eqz_choice1,bool).

component(signal_1, constant_signal(value(1), signal(sig1))).
component(itc_A_reg,reg(control(sig1),input(s1),output(tc_A_s1))).
component(equz_comp1,transform(inputs([s1]),function(equz),output(eqz_s1))).
component(eqz_reg1,reg(control(sig1),input(eqz_choice1),output(eqz_ret1))).

component(choice,
table([ [itc_plus_A, mtc_plus_A, itc_min_A, mtc_min_A, eqz_choice1],
        [0, 0, 1, 0,eqz_s1],
        [0, 0, 0, 1,eqz_s1],
        [1, 0, 0, 0,eqz_s1],
        [0, 1, 0, 0,eqz_s1]|eqz_ret1])).

component(ctrl_tc_A,
table([[eqz_ret1,itc_plus_A, mtc_plus_A, itc_min_A, mtc_min_A, s1],
        [0, 0, 0, 1, 0, dec(tc_A_s1)],
        [0, 0, 0, 0, 1, dec(tc_A_s1)],
        [*, 1, 0, 0, 0, inc(tc_A_s1)],
        [*, 0, 1, 0, 0, inc(tc_A_s1)]|tc_A_s1])).

component(ctrl_tc_plus_A, table([[itc_plus_A, mtc_plus_A, tc_plus_A],
                                 [*, 1, 1],
                                 [1, *, 1]|0])).

component(ctrl_tc_minus_A, table([[itc_min_A, mtc_min_A, tc_min_A],
                                  [*, 1, 1],
                                  [1, *, 1]|0])).

st_nxst(tc_A_s1,n_tc_A_s1).
st_nxst(eqz_ret1,n_eqz_ret1).
```

```
%counter ic_A

signal(ic_A,wordn).

component(ctrl_ic_A, table([[equz_new(ic_A),ic_plus_A, ic_min_A, n_ic_A],
                                    [0, 0, 1, dec(ic_A)],
                                    [*, 1, 0, inc(ic_A)]|ic_A])).

%--- Initial states ---
%

init_val(is_A,red).
init_val(ms_A,red).
init_val(ts_A,dispatch).

init_val(tc_A_s1,init_tc).
init_val(ic_A,init_ic).

init_val(eqz_ret1,1).
init_var(init_tc,wordn).
init_var(init_ic,wordn).

%--- Outputs ---
%
outputs([irl_A, igl_A, mrl_A, mgl_A, itc_plus_A, mtc_plus_A, ic_plus_A, ic_min_A]).

%--- Partitions ---
%
output_partition([[[irl_A]],[[igl_A]],[[mrl_A]],[[mgl_A]],
                  [[itc_plus_A]], [[mtc_plus_A]],
                   [[ic_plus_A]], [[ic_min_A]] ]).


next_state_partition([
[[n_is_A]],
[[n_ms_A]],
[[n_ts_A]],
[[n_ic_A]],
[[n_tc_A_s1]],
[[n_eqz_ret1]]
 ] ).
```

```
%--- State variable, next state variable mapping---
%
st_nxst(is_A,n_is_A).
st_nxst(ms_A,n_ms_A).
st_nxst(ts_A,n_ts_A).
st_nxst(ic_A,n_ic_A).


%--- Partition strategy---
%
par_strategy(auto, auto).




%-------------------------------------------------------------
% File:  itc_alg.pl
% Title: Algebraic specification file for the ITC example
%-------------------------------------------------------------

% Multifile definition for Prolog predicates
%
:- multifile  abs_sort/1.
:- multifile  conc_sort/2.
:- multifile  function/3.
:- multifile  gen_const/2.

:- multifile  rr/3.
:- multifile  ucrr/2.

% Algebraic specification
%
conc_sort(mi_sort,[green,red,exiting,entering]).
conc_sort(ts_sort,[dispatch,iuse,muse,iclear,mclear]).

% Functions
%

function(lessn,[wordn],bool).
function(equz,[wordn],bool).
function(equz_new,[wordn],bool).

function(absComp,[wordn,wordn],bool).
```

```
% Conditional rewrite rules;
% To rewrite the terms of abstract functions, only the conditional
% rewrite rules can be used. For rules which don't have explicit conditions,
% as the following two rules, the condition set is "[]".
%
rr([], dec(inc(X)), X).
rr([], inc(dec(X)), X).

% Xterm rewrite rules;
%
xtrr([],equz(zero),1).
xtrr([],lessn(zero),1).
xtrr([],absComp(X,X),1).



%-----------------------------------------------------------
% File itc_o.pl: Variable order specification file for the ITC example
%-----------------------------------------------------------

order_main([
rand_choice1,
rand_choice2,
rand_choice3,
rand_choice4,

sig1,
signal0,
x,

init_is,
init_ms,
init_ts,

tc,
ic,

ts_A,
is_A,
ms_A,

ie,
ix,
```

me,
mx,

cx_A,

n_ts_A,
n_is_A,
n_ms_A,

n_ie,
n_ix,
n_me,
n_mx,

is_entering,
is_exiting,
ms_entering,
ms_exiting,

lessn_ic_A,
equz_tc_A,

tc_A_s1,
inc_s1,
dec_s1,
s1,
n_tc_A_s1,
eqz_s1,
eqz_choice1,
eqz_ret1,
n_eqz_ret1,

ic_A_s2,
inc_s2,
dec_s2,
s2,
%rand_choice1,
%rand_choice2,
%rand_choice3,
%rand_choice4,
n_ic_A_s2,
eqz_s2,
eqz_choice2,
lessn_s2,

lessn_choice,
eqz_ret2,
n_eqz_ret2,
lessn_ret,
n_lessn_ret,

tc_A,
tc1_A,
tc0_A,
ic_A,
ic1_A,
ic0_A,
n_tc_A,
n_tc1_A,
n_tc0_A,
n_ic_A,
n_ic1_A,
n_ic0_A,

irl_A,
igl_A,
itc_plus_A,
itc_min_A,
ic_min_A,

mrl_A,
mgl_A,
mtc_plus_A,
mtc_min_A,
ic_plus_A,

tc_plus_A,
tc_min_A,

ig_A,
iy_A,
ir_A,
iu_A,

mg_A,
my_A,
mr_A,
mu_A,

```
lessn,
equz,
equz_new,
inc,
dec
]).
```

# Appendix B - ITC behavioral model with 4 bit counters in Verilog HDL

```
/*==========================================================*/
/* itc_4b.vl: ITC specification in Verilog  */
/*Counters are instantiated to 4 bits */
/*==========================================================*/

/* Enumerate type definition. This is an extension of Verilog allowed by VIS */
typedef enum {green, entering, red, exiting} ms_sort;
typedef enum {dispatch, iuse, muse, iclear, mclear} ts_sort;

/*============== Main module ===============*/

module main(clk,igl,irl,mgl,mrl);
input clk;
output igl,irl,mgl,mrl;

wire ie,ix,me,mx,igl,irl,mgl,mrl;
wire ic_plus,ic_minus,itc_plus,itc_minus,mtc_plus,mtc_minus,tc_plus,tc_minus;
wire [3:0] tc,ic;

ms_sort wire is, ms;

sensor sensor(clk, ie,ix,me,mx,is,ms);
counter counter(clk,tc,ic, ic_plus, ic_minus, itc_plus, itc_minus, mtc_plus, mtc_minus,
tc_plus, tc_minus);
island island(clk,ie,ix,igl,irl,ic_minus,itc_plus,itc_minus,iu,ir,ig,iy,is);
mainland      mainland(clk,me,mx,mgl,mrl,ic,ic_plus,mtc_plus,mtc_minus,mu,mr,mg,my,
ms);
tunnel tunnel(clk,iu,ir,ig,iy,mu,mr,mg,my,tc,ic);

endmodule

/* Sensors.
  For VIS, a variable in a CTL formula should not have primary
  inputs as its supporting variables. So module Sensor simply
  simulates the enviroment by modelling the inputs as state
  variables in an enviroment state machine.
*/

module sensor(clk, ie,ix,me,mx,is, ms);
```

```
input clk,is,ms;
output ie,ix,me,mx;

ms_sort wire is, ms;

wire rand_choice1,rand_choice2,rand_choice3,rand_choice4;
reg ie,ix,me,mx, ie_delay1, me_delay1;


initial ie = 0;
initial ix = 0;
initial me = 0;
initial mx = 0;
initial ie_delay1 = 0;
initial me_delay1 = 0;

assign rand_choice1 = $ND(0,1);
assign rand_choice2 = $ND(0,1);
assign rand_choice3 = $ND(0,1);
assign rand_choice4 = $ND(0,1);

always @(posedge clk) begin
   if (rand_choice1 == 0)
      ie = 0;
   else
ie = 1;


   if (rand_choice2 == 0)
      ix = 0;
   else
ix = 1;

   if (rand_choice3 == 0)
      me = 0;
   else
me = 1;

if (rand_choice4 == 0)
      mx = 0;
   else
mx = 1;

end
```

```
endmodule

/*========================== Counters ====================*/

module counter(clk, tc, ic, ic_plus, ic_minus, itc_plus, itc_minus, mtc_plus, mtc_minus,
tc_plus, tc_minus);
input clk;
input ic_plus,ic_minus,itc_plus,itc_minus,mtc_plus,mtc_minus;
output tc,ic,tc_plus,tc_minus;

reg [3:0] tc,ic;
wire ic_plus,ic_minus,itc_plus,itc_minus,mtc_plus,mtc_minus, tc_plus, tc_minus;

initial tc = 0;
initial ic = 0;


assign tc_plus = (itc_plus || mtc_plus) ? 1 : 0;    //added
assign tc_minus = (itc_minus || mtc_minus) ? 1 :0;  // added
always @(posedge clk) begin
   if ((ic_minus == 1) && (ic > 0) && (ic_plus == 0)) ic = ic - 1;
   else if ((ic_plus == 1) && (ic < 15) && (ic_minus == 0)) ic = ic + 1;
   else ic = ic;

   if     ((tc_minus == 1) && (tc > 0) && (tc_plus == 0))  tc = tc - 1;
   else if ((tc_plus == 1)  && (tc < 15) && (tc_minus == 0)) tc = tc + 1;
   else tc = tc;
end
endmodule


/*==================== Island Light Controller ======================*/

module island(clk,ie,ix,igl,irl,ic_minus,itc_plus,itc_minus,iu,ir,ig,iy,is);

input clk;
input ie,ix,ig,iy;
output igl,irl,ic_minus,itc_plus,itc_minus,iu,ir,is;

wire ie,ix,ig,iy,igl,irl,iu,ir;
wire ic_minus,itc_plus,itc_minus;

ms_sort reg is;
```

```
initial is = red;

always @(posedge clk) begin
        case (is)
        green:if ((iy==0)&&(ie==0)) is = green;
        else if ((iy==0)&&(ie==1)) is = entering;
        else is = red;
        entering: if (ie==0) is = green;
        else is = entering;
        red: if ((ix==0)&&(ig==0)) is = red;
        else if ((ix==0)&&(ig==1)) is = green;
        else is = exiting;
        exiting: if (ix==0) is = red;
        else is = exiting;
        endcase
end

assign ir = ((is == red)&&(ie == 1)) ? 1 : 0;
assign iu = ((is == green) || (is == entering)) ? 1 : 0;
assign irl = ((is == red) || (is == exiting)) ? 1 : 0;
assign igl = ((is == green) || (is == entering)) ? 1 : 0;
assign itc_plus = ((is == green) && (iy == 0) && (ie == 1)) ? 1 : 0;
assign itc_minus = ((is == red) && (ix == 1)) ? 1 : 0;
assign ic_minus = ((is == green) && (iy == 0) && (ie == 1)) ? 1 : 0;

endmodule

/*============= Mainland Light Controller ================*/

module mainland(clk,me,mx,mgl,mrl,ic,ic_plus,mtc_plus,mtc_minus,mu,mr,mg,my,ms);

input clk;
input [3:0] ic;
input me,mx,mg,my;
output mgl,mrl,ic_plus,mtc_plus,mtc_minus,mu,mr,ms;

wire [3:0] ic;
wire me,mx,mg,my;
wire mgl,mrl,ic_plus,mtc_plus,mtc_minus,mu,mr;

ms_sort reg ms;

initial ms = red;
```

```
always @(posedge clk) begin
        case (ms)
        green:if (ic >= 15) ms = red;
        else if ((my==0)&&(me==0)) ms = green;
        else if ((my==0)&&(me==1)) ms = entering;
        else ms = red;
        entering: if (me==0) ms = green;
        else ms = entering;
        red: if ((mx==0)&&(mg==0)) ms = red;
        else if ((mx==0)&&(mg==1)) ms = green;
        else ms = exiting;
        exiting: if (mx==0) ms = red;
        else ms = exiting;
        endcase
end

assign mr = ((ms == red)&&(me == 1)) ? 1 : 0;
assign mu = ((ms == green) || (ms == entering)) ? 1 : 0;
assign mrl = ((ms == red) || (ms == exiting)) ? 1 : 0;
assign mgl = ((ms == green) || (ms == entering)) ? 1 : 0;
assign mtc_plus =((ms == green) && (my == 0) && (me == 1) && (ic < 15)) ? 1 : 0;
assign mtc_minus = ((ms == red) && (mx == 1)) ? 1 : 0;
assign ic_plus = ((ms == green) && (my == 0) && (me == 1) && (ic < 15)) ? 1 : 0;

endmodule


/*================== Tunnel Controller ======================*/

module tunnel(clk,iu,ir,ig,iy,mu,mr,mg,my,tc,ic);

input clk;
input [3:0] ic, tc;
input iu,ir,mu,mr;
output ig,iy,mg,my;

wire [3:0] ic, tc;
wire iu,ir,mu,mr;
wire ig,iy,mg,my;

ts_sort reg ts;

initial ts = dispatch;
```

```
always @(posedge clk) begin
        case (ts)
        dispatch: if ((ir==0)&&(mr==0)) ts = dispatch;
        else if ((ir==0)&&(mr==1)&&(ic>=15)) ts = dispatch;
        else if ((ir==0)&&(mr==1)&&(ic<15)&&(iu==1)) ts = iuse;
        else if ((ir==0)&&(mr==1)&&(ic<15)&&(iu==0)&&(tc!=0)) ts=iclear;
            else if ((ir==0)&&(mr==1)&&(ic<15)&&(iu==0)&&(tc==0)) ts=dispatch;
        else if ((ir==1)&&(mu==1)) ts=muse;
        else if ((ir==1)&&(mu==0)&&(tc!=0)) ts=mclear;
        else ts=dispatch;
        iuse:   if (iu==0) ts = iclear;
        else ts = iuse;
        muse:   if (mu==0) ts = mclear;
        else ts = muse;
        iclear: if (tc!=0) ts = iclear;
        else ts = dispatch;
        mclear: if (tc!=0) ts = mclear;
        else ts = dispatch;
        endcase
end

assign ig = (((ts==dispatch)&&(ir==1)&&(tc==0)&&(mu==0)) ||
    ((ts==mclear)&&(tc==0))) ? 1 : 0;

assign iy = (ts == iuse) ? 1 : 0;
assign mg = (((ts==dispatch)&&(ir==0)&&(mr==1)&&(ic<15)&&(tc==0)&&(iu==0)) ||
    ((ts==iclear)&&(tc==0))) ? 1 : 0;
assign my = (ts == muse) ? 1 : 0;

endmodule
```

# Appendix C - Behavioral description of the Abstract Counter using MDG-HDL

```
%-----------------------------------------------------
% File: counter_s.pl
% Title: Behavioral description of the Abstract Counter
%-----------------------------------------------------

% Multifile declaration required by Prolog system.
%
:- multifile component/2.
:- multifile signal/2.
:- multifile next_state_partition/1.
:- multifile output_partition/1.
:- multifile init_val/2.
:- multifile init_var/2.
:- multifile init_con/2.
:- multifile st_nxst/2.
:- multifile outputs/1.
:- multifile par_strategy/2.

%====================== Inputs and Outputs ====================

signal(input,instructionSort).
signal(state,stateSort).
signal(double,bool).
signal(load_in,wordn).
signal(pc,wordn).

component(sta_comp,table([ [state,input, double, n_state],
                           [c_fetch,c_no_op, *,c_fetch],
                           [c_fetch,c_load,*,c_load],
                           [c_fetch, c_inc1, *, c_inc1],
                           [c_fetch, c_inc2, *,c_inc1],
                           [c_load, *,*, c_fetch],
                           [c_inc1, *, 0, c_fetch],
                           [c_inc1, *, 1, c_inc2],
                           [c_inc2, *, *, c_fetch]
                           ])).
component(double_comp,table([ [input,n_double],
                              [c_inc2,1]
                              |0])).
```

```
component(pc_comp,table([ [state,n_pc],
                          [c_load, load_in],
                          [c_inc1, finc(pc)],
                          [c_inc2, finc(pc)]
                          |pc])).


st_nxst(double,n_double).
st_nxst(pc,n_pc).
st_nxst(state,n_state).

%--- Initial states ---
%
init_var(init_pc,wordn).
init_val(pc,init_pc).
init_val(double,0).
init_val(state,c_fetch).

%--- Outputs ---
%
outputs([]).

%--- Partitions ---
%
output_partition([]).

next_state_partition([
[[n_double]],
[[n_pc]],
[[n_state]]
]).

%--- Partition strategy---
%
par_strategy(auto, auto).
%-------------------------------------------------------------
% File:  itc_alg.pl
% Title: Algebraic specification file for the ITC example
%-------------------------------------------------------------

% Multifile definition for Prolog predicates
%
:- multifile  abs_sort/1.
```

```
:- multifile  conc_sort/2.
:- multifile  function/3.
:- multifile  gen_const/2.

:- multifile  rr/3.
:- multifile  ucrr/2.

% Algebraic specification
%
conc_sort(stateSort,[c_fetch,c_load,c_inc1,c_inc2]).
conc_sort(instructionSort,[c_no_op,c_load,c_inc1,c_inc2]).

% Functions
%

function(equz,[wordn],bool).

function(finc,[wordn],wordn).

function(absComp,[wordn,wordn],bool).
rr([], dec(inc(Y)), Y).
rr([], inc(dec(Y)), Y).

% Xterm rewrite rules;
%
xtrr([],equz(zero),1).


xtrr([],absComp(X,X),1).


%-------------------------------------------------------------
% File: counter_alg.pl
% Title: Algebraic specification file for the Abstract Counter example
%-------------------------------------------------------------

% Multifile definition for Prolog predicates
%
:- multifile  abs_sort/1.
:- multifile  conc_sort/2.
:- multifile  function/3.
:- multifile  gen_const/2.

:- multifile  rr/3.
```

```
:- multifile  ucrr/2.

% Algebraic specification
%
conc_sort(stateSort,[c_fetch,c_load,c_inc1,c_inc2]).
conc_sort(instructionSort,[c_no_op,c_load,c_inc1,c_inc2]).

% Functions
%

function(equz,[wordn],bool).
function(finc,[wordn],wordn).

function(absComp,[wordn,wordn],bool).
rr([], dec(inc(Y)), Y).
rr([], inc(dec(Y)), Y).

% Xterm rewrite rules;
%
xtrr([],equz(zero),1).
xtrr([],absComp(X,X),1).


%-----------------------------------------------------------------
% File counter_o.pl: Variable order specification file for the Abstract Counter example
%-----------------------------------------------------------------

order_main([
init_pc,
load_in,
input,
double,
n_double,
state,
n_state,
pc,
n_pc,
finc
]).
```

# Appendix D - Behavioral description of the Abstract Counter in Verilog HDL

```
// A behavioral model of the Abstract Counter

/* Enumerate type definition. This is an extension of Verilog allowed by VIS */
typedef enum {c_fetch, c_load, c_inc1, c_inc2} state_sort;
typedef enum {c_no_op, c_load, c_inc1, c_inc2} instruction_sort;

/*============== Main module ==============*/
module main(clk);
input     clk;

// as input_instruction needed in th eproperty, it has to be eventually driven by a
// latch. So it cannot be a primary input.

wire [1:0] random_choice;
wire [3:0] load_in;
instruction_sort wire input_instruction;

reg    double;
reg [3:0] pc;

state_sort reg state;
instruction_sort  reg     input_instruction;

initial double = 0;
initial state = c_fetch;
initial pc = 0;
initial input_instruction = c_no_op;

assign random_choice[1] = $ND(0,1);
assign random_choice[0] = $ND(0,1);
assign load_in[3] = $ND(0,1);
assign load_in[2] = $ND(0,1);
assign load_in[1] = $ND(0,1);
assign load_in[0] = $ND(0,1);

  always @(posedge clk)
    begin
    if (random_choice == 0)
          input_instruction = c_no_op;
    else if (random_choice == 1)
```

```
            input_instruction = c_inc1;
    else if (random_choice == 2)
            input_instruction = c_inc2;
    else
            input_instruction = c_load;
    end

always @(posedge clk)
begin
 case (state)
  c_fetch:
    begin
         if (input_instruction == c_load) state = c_load;
         else if (input_instruction == c_inc1) state = c_inc1;
         else if (input_instruction == c_inc2) state = c_inc1;
         else state = c_fetch;
          pc = pc;
    end // case: c_fetch

  c_load:
    begin
         state = c_fetch;
         pc = load_in;
    end // case: c_load

  c_inc1:
    begin
         if (double == 1) state = c_inc2;
         else state = c_fetch;
         pc = pc +1;
    end // case: c_inc1

  c_inc2:
    begin
         state = c_fetch;
         pc = pc + 1;
    end // case: c_inc2

endcase // case (state)


if (input_instruction == c_inc2)
   double = 1'b1;
else
```

```
    double = 1'b0;

end
endmodule
```