# MODEL CHECKING:

# TWO DECADES OF NOVEL TECHNIQUES AND TRENDS

PHD GENERAL EXAM REPORT

Ashish Sabharwal
Computer Science and Engineering
University of Washington, Box 352350
Seattle, Washington 98195-2350
ashish@cs.washington.edu

*Supervisory Committee*: Paul Beame (Advisor), Elizabeth Thompson (GSR),
Henry Kautz, Richard Ladner, Daniel Weld

May 8, 2002

# Contents

**Abstract**

Model checking is a fully automatic and complete technique for verifying whether a finite state transition system satisfies a set of desired properties. It involves the process of creating a formal model for the given system, using mechanisms such as temporal logics for specifying the desired properties succinctly, and developing algorithms for testing if the model satisfies its specification. Tools available in the early 1980's for performing this task used explicit representation of systems as directed, acyclic graphs. This motivated the tools developed for finite automata on infinite words to be used for model checking. Although helpful for understanding the problem and designing the very first algorithms, techniques based on explicit representation became unfeasible as systems one was interested in verifying became larger.

A major breakthrough was made in the late 1980's with the use of symbolic representations of sets of states as Binary Decision Diagrams (BDDs). This allowed one to analyze systems much bigger than what was possible before. Another approach that began developing in mid 1990's was the use of well researched propositional satisfiability (SAT) algorithms for model checking. This provided a new way of looking at the problem and turned out to be even better than BDD based algorithms on certain domains. At around the same time, SAT procedures also began being used for solving notoriously hard planning problems occurring in artificial intelligence and very similar in nature to model checking. Smart encodings and stochastic strategies allowed generalized SAT procedures to outperform traditional specialized planning systems on certain hard domains. A final development in the late 1990's was the creation of a simple unifying representation called Boolean Expression Diagram (BED). With this, one is able to exploit the advantages of both BDD and SAT based techniques depending on the domain at hand. This paper presents a summary of these approaches.

# 1 Introduction

Today computers are used to perform a variety of complex and sensitive tasks such as medical instrumentation, e-commerce and air traffic control. For many of these applications, it is critical to have a high degree of confidence in the correctness of the underlying automated procedure. As computational power increases at an exponential rate and increasingly complex systems are created, the process of design validation becomes harder. Methods traditionally used for this purpose include *simulating* the system on an abstraction thereof and *testing* it on some real world examples. These methods, though still widely used in practice, suffer from the disadvantage of being manual, incomplete and relatively ineffective in later stages of the design process when the number of bugs has dropped down.

*Formal verification* presents an attractive alternative. While simulation and testing only explore some of the possible behaviors of the system, formal verification involves exhaustively going through all possible scenarios, looking for one where the system fails. Several approaches, manual and automated, have been proposed for performing such complete verification. *Theorem provers* are based on deductive reasoning to prove the correctness of systems starting from a set of base axioms and rules. They, however, typically require manual intervention to guide the validation procedure in the correct direction.

A second approach to formal verification is *model checking*, which is the subject of this paper. It is a technique for verifying whether a given finite state concurrent system evolving over time satisfies a set of desired properties. Although the restriction to finitely many states rules out certain classes of systems, model checking can be applied to several other very important ones. Two things that make this approach interesting for practical purposes are complete automatizability and the ability to generate a counterexample in case the given system does not satisfy the desired specification.

The process of model checking can be broken down into three major steps. First, the given finite state system has to be modeled as a formal structure understandable by an automated verification engine. Kripke structures are typically used for this. Second, the properties desired of the system must be specified in a formal way. We will see how temporal logics such as CTL and LTL can be used for succinct specification of properties of interest. Finally, the model of the system needs to be checked against the specification to see if it satisfies the desired properties. This last step should also generate a counterexample in case the system fails to meet the specification.

Various representations used for modeling systems and specifying properties lead to different model

checking procedures with varying time complexity. It is natural to represent finite state systems as graphs with nodes as states and edges as transitions. This leads to *explicit* state exploration procedures based on reachability analysis. Results and algorithms from automata theory can be applied to reason about such procedures [VW86, Pel98]. Typical implementations using this approach can handle systems with around $10^6$ states. However, most interesting real world systems have far too many states to be represented and manipulated in an explicit fashion. The use of a simple data structure for Boolean functions called Binary Decision Diagram or BDD [Bry86] as a *symbolic* representation of sets of states was a novel idea that gave model checking far more credibility and hope [BCM$^+$92]. This new canonical representation avoids the exponential state space blowup by dealing with sets of states in a succinct, implicit manner. Use of BDDs took the limit of practically feasible systems up to $10^{20}$ states, and with further refinements and increased computational power, now allows one to handle systems with more than $10^{120}$ states.

Like many interesting problems, model checking can also be cast as a propositional formula satisfiability (SAT) problem. Increasingly powerful SAT solvers [MSS96, Zha97, SK93] resulted in exploration of the use of SAT algorithms for model checking. For certain domains where intermediate or even initial BDD representations are huge, model checking procedures based on satisfiability testing [BCCZ99] allow us to perform verification in a reasonable time. However, for some other domains, BDD based algorithms are still better. A natural step, then, is to try to combine the advantages of BDD and SAT based approaches. A yet another interesting data structure called Boolean Expression Diagram or BED [AH97] provides exactly what is needed. Clever representation and efficient manipulation algorithms for BEDs allow one to combine the advantages of both BDD and SAT based techniques to obtain a unified procedure for model checking [WBCG00]. This is a relatively new idea still being explored.

An area studied extensively by the artificial intelligence community and closely related to model checking is *planning*. We briefly discuss this toward the end of the paper. Given a set of fluents or predicates that hold at certain states and a set of actions that can be taken to transform the state, the planning problem is to come up with a strategy (a sequence of actions) so that the system moves from any initial state to the goal states, respecting all given mutual exclusion and other constraints. Examples of such systems appearing frequently in literature are Blocks World, Logistics, and Rocket Controllers. Even though it was believed that use of domain specific knowledge was essential for the construction of plan-generation systems, use of fast SAT solvers and good encodings showed that for certain domains, general purpose satisfiability solvers

can outperform specific planning systems [KS96, KMS96, KS99]. A crucial element for this approach is the encoding used to convert the standard STRIPS-style [FN71] representation to SAT instances. It turns out that performance is highly affected by the encoding used.

# 2 Preliminaries

The problem of model checking involves taking a finite state transition system and verifying whether it satisfies certain desired properties. The first step in doing this is to construct a *formal model* for the system of interest. Ideally, this model should capture all details that must be considered to check if the desired properties hold, and it should also abstract away any details that are irrelevant to these properties. The second vital component of model checking is a formalism for *specifying properties* that are then checked against the model created for the system to see if the system satisfies them. This, as we will see, is typically done using various kinds of *temporal logics*.

## 2.1 Modeling Systems

Finite state systems can often be characterized by their input-output behavior. These include many hardware and software systems. However, there are applications that don't necessarily have a terminal output. What matters for them is how the system behaves over time. In many cases, the system might never terminate. We will be concerned with such *reactive systems* in this paper, though it should be remembered that the formalism described here can be used to model standard input-output systems as well.

### 2.1.1 Kripke Structures

A reactive system is thought of as being in a *state* at any instant of time. This captures values of all variables in the system at that instant. As time progresses, the system *transitions* into another state as a result of some action. An infinite sequence of such state transitions characterizes a *computation* of the reactive system. We capture this intuition formally in the form of a state transition graph called a *Kripke structure*. Paths in this graph model computations of the system and are checked against desired properties.

Let $AP$ be a set of atomic propositions. A *Kripke structure* $M$ over $AP$ is a 4-tuple $M = (S, S_0, R, L)$ where

1. $S$ is a finite set of states,

2. $S_0 \subseteq S$ is a set of initial states,

3. $R \subseteq S \times S$ is a transition relation that is total, *i.e.* for every state $s \in S$, there is a state $s' \in S$ such that $R(s, s')$, and

4. $L : S \to 2^{AP}$ is a function that labels each state with the set of atomic propositions true in that state.

A 3-tuple $(S, R, L)$ might sometimes be used to denote a Kripke structure when the set of initial states is irrelevant. A *path* in the structure $M$ from a state $s$ is an infinite sequence of states $\pi = s_0 s_1 s_2 \ldots$ such that $s_0 = s$ and $R(s_i, s_{i+1})$ holds for all $i \geq 0$.

### 2.1.2 Representation as a Boolean Formula

Kripke structures can be represented as Boolean formulas in a natural way. This is useful because Boolean formulas in turn have other compact symbolic representations that make model checking algorithms efficient enough to be used in practice. Given a Kripke structure $M = (S, S_0, R, L)$ with $|S| = 2^m$, the corresponding Boolean formula has $2m$ state variables. The first set of $m$ variables (denoted $\bar{x}$) represents starting states of transitions while the second set (denoted $\bar{x}'$) represents ending states. Note that any transition relation $R(s_1, s_2)$ can also be thought of as a Boolean relation $\hat{R}(\bar{x}, \bar{x}')$, where $\bar{x}$ represents the state $s_1$ in the first set of variables and $\bar{x}'$ represents the state $s_2$ in the second set of variables. $R$ is then represented as the Boolean function $I_{\hat{R}}$ which is simply the characteristic function of $\hat{R}$, *i.e.* $I_{\hat{R}}(\bar{x}, \bar{x}') = 1$ iff $\hat{R}(\bar{x}, \bar{x}')$. The set $S_0$ of initial states is similarly represented by its characteristic function $I_{S_0}$ over variables $\bar{x}$. For $L : S \to 2^{AP}$, each $p \in AP$ is represented by the characteristic function of the set of states it is true in.

## 2.2 Temporal Logics

We now describe a formalism for specifying interesting properties of state transition systems. Properties of a given state can be effectively described in any standard logic that uses atomic propositions and combines them using Boolean connectives such as conjunction, disjunction and negation. For reactive systems that our Kripke structures represent, we are typically also interested in temporal properties. For instance, we might want our system to *never* enter a certain set of bad states. Or we might require that a particular

atomic proposition *eventually* becomes true. These properties are specified using *temporal logics*, which, in addition to Boolean operators over atomic propositions, also support *temporal operators*.

### 2.2.1 The (Generalized) Computation Tree Logic: CTL*

CTL* [CES86] is a powerful temporal logic whose sublogics CTL and LTL are widely used to succinctly express interesting properties of real life state transition systems. It works on *computation trees* which are formed by simply starting with an *initial state* in a given Kripke structure as the root and unwinding the structure into an infinite tree. Paths in this tree correspond to all possible executions starting from the initial state.

CTL* contains the usual Boolean operators AND $\wedge$, OR $\vee$ and NOT $\neg$. To specify branching properties, CTL* has two *path quantifiers* describing branching structure in the computation tree: $\mathbf{A}$ for "all computation paths" and $\mathbf{E}$ for "some computation path." For example, a property expressed by the CTL* formula $\mathbf{A}\, g$ holds at state $s$ of the computation tree if the property expressed by $g$ for holds along *all* paths starting at $s$. Henceforth, we will use $g$ to denote both the formula and the property it expresses. In addition, CTL* has five *temporal operators* that describe what it means for a property to hold along a path: the *next state* operator $\mathbf{X}$ that requires the property to hold at the second state of the path, the *eventually* operator $\mathbf{F}$ that requires the property to hold at some state along the path, the *always* or *globally* operator $\mathbf{G}$ that asserts that the property must hold at every state along the path, the binary *until* operator $\mathbf{U}$ which requires the second property to hold at some state of the path and the first to hold at every preceding one, and the binary *release* operator $\mathbf{R}$ that also combines two properties and asserts that the second property must hold at all states along the path up to and including the first state where the first property holds.

Formulas in CTL* can be categorized into two types: *state formulas* specifying properties of a given state (denoted here by $f, f_1, f_2, \ldots$) and *path formulas* specifying properties of a path (denoted here by $g, g_1, g_2, \ldots$). State formulas include all atomic propositions $p \in AP$ as well as formulas of the form $\neg f, f_1 \vee f_2, f_1 \wedge f_2, \mathbf{A}\, g$ and $\mathbf{E}\, g$. Path formulas include every state formula as well as formulas of the form $\neg g, g_1 \vee g_2, g_1 \wedge g_2, \mathbf{X}\, g, \mathbf{F}\, g, \mathbf{G}\, g, g_1\, \mathbf{U}\, g_2$ and $g_1\, \mathbf{R}\, g_2$.

One can easily check that just like Boolean operators $\vee$ and $\wedge$, path operators $\mathbf{E}$ and $\mathbf{A}$ as well as temporal operators $\mathbf{F}$ and $\mathbf{G}$, and $\mathbf{U}$ and $\mathbf{R}$ are logical duals of each other. More precisely, $\mathbf{A}\, g \equiv \neg\mathbf{E}\, \neg g$, $\mathbf{G}\, g \equiv \neg\mathbf{F}\, \neg g$, and $g_1\, \mathbf{R}\, g_2 \equiv \neg(\neg g_1\, \mathbf{U}\, \neg g_2)$. This allows us to focus only on a subset of these operators when

designing model checking algorithms.

### 2.2.2 CTL and LTL: Weaker but Useful Sublogics

The power of CTL* in expressing a variety of useful properties also makes it a difficult logic to reason about in practice. It turns out that two of its sublogics, CTL and LTL, are good enough to express many interesting properties of real life state transition systems. Their relative simplicity lends them to better analysis and faster model checking techniques.

*Computation Tree Logic* or CTL [BMP83] puts the restriction that each of the five temporal operators $\mathbf{X}, \mathbf{F}, \mathbf{G}, \mathbf{U}$ and $\mathbf{R}$ be immediately preceded by a path quantifier $\mathbf{A}$ or $\mathbf{E}$. This, for instance, rules out the CTL* formula $\mathbf{A}[\mathbf{FG}\, p]$ that expresses the property that along every path, there is some state from which $p$ will hold forever. Given operator dualities, all formulas in CTL can in fact be expressed in terms of three compound operators $\mathbf{EX}, \mathbf{EG}$ and $\mathbf{EU}$ along with Boolean connectives $\neg$ and $\wedge$.

*Linear Temporal Logic* or LTL [Pnu81] is another useful sublogic of CTL* that restricts the use of path quantifiers. All formulas in LTL are of the form $\mathbf{A}\, g$ or $\mathbf{E}\, g$ where $g$ is a *restricted* path formula without any path quantifiers $\mathbf{A}$ or $\mathbf{E}$. This, for example, rules out the formula $\mathbf{AG}[\mathbf{EF}\, p]$ expressing the property that along every state in every path, one can choose a path where $p$ will eventually hold. Again, given operator dualities, $g$ in an LTL formula $\mathbf{A}\, g$ can be thought of as any restricted path formula constructed using temporal operators $\mathbf{X}, \mathbf{G}$ and $\mathbf{U}$ along with Boolean connectives $\neg$ and $\wedge$.

Due to its severe restriction on path quantifiers, LTL expresses properties that are *linear* in time. This is in contrast with CTL which allows a richer computation *tree* structure but instead restricts the use of temporal operators. These two logics actually have different expressive powers (neither is subsumed by the other) and are both strictly weaker than CTL*. The examples we saw above that were not expressible in CTL and LTL, respectively, are in fact expressible in the other sublogic, while their disjunction $\mathbf{A}[\mathbf{FG}\, p] \vee \mathbf{AG}[\mathbf{EF}\, p]$, written here as a CTL* formula, is not expressible in either CTL or LTL.

## 3 Explicit State Model Checking

Given a Kripke structure $M = (S, S_0, R, L)$ representing a finite state system and a temporal logic formula $f$ specifying desired properties, the *model checking problem* is to check whether all initial states $s \in S_0$ satisfy $f$. In practice, one ends up computing the set $S_f$ of all states in which $f$ holds and determining if

$S_0 \subseteq S_f$. It is worth noting here that $f$ holding in the initial states indirectly represents future properties of the system because of the way temporal operators behave. An example of a property that one might require to hold at all initial states of the system is that from all states reachable from this point, there is always a path which leads to a pre-specified "halt" state.

## 3.1 The Basic Model Checking Procedure

A natural way to solve the model checking problem is by using an *explicit* representation of the Kripke structure $M = (S, S_0, R, L)$ as a labeled, directed graph whose nodes represent the states in $S$, edges represent the transition relation $R$, and node labels describe the labeling function $L$. We briefly discuss a basic model checking procedure that uses this graph representation to recursively compute the set $S_f$ of all states where $f$ holds. At each stage of the recursion, this procedure explicitly maintains the set of states where the corresponding sub-function of $f$ holds. The rest of the paper will discuss how automata theory tools and clever representations of Boolean functions can be used to improve this basic procedure.

The base case of the recursion occurs when $f$ is an atomic proposition. Here we simply construct the set of all states where this atomic proposition holds. Boolean operators $\vee$ and $\wedge$ applied on two functions correspond to union and intersection, respectively, of the sets of states associated with these two functions. Boolean negation $\neg$ corresponds to set complement. For $\mathbf{EX}\, g$, we compute the set of all states which have a successor in the set corresponding to $g$. For $\mathbf{E}[g_1\, \mathbf{U}\, g_2]$, we start with all states where $g_2$ holds and then work backwards using the converse of the transition relation $R$ to compute all states that can be reached by a path which has $g_1$ true in each of its states. Other temporal operators are handled in a similar way, though operators $\mathbf{G}$ and $\mathbf{R}$ specifying global properties are a little more involved, requiring computation of strongly connected components. See [CGP99] for details. The whole process has complexity $O(|M|^{|f|})$.

## 3.2 Automata on Infinite Words

The theory of finite automata has been extensively studied over the years and finds natural uses in various fields such as compilation, hardware design and structural complexity. It also provides a convenient tool for designing model checking algorithms. The basic idea [VW86, Pel98, CGP99] is to represent possible behaviors of the system as well as all acceptable behaviors as (non-deterministic) $\omega$-regular automata and use tools from automata theory to check if the language of the former is contained in that of the latter. Any

word violating this language containment serves as a witness for system failure.

$\omega$-regular automata work on *infinite* words over a finite alphabet, and are therefore a natural choice for representing behaviors of a reactive system as an infinite sequence of states. Their structure is the same as that of regular automata – a set of states including specifically marked initial, accepting and rejecting ones, and a set of transitions labeled with symbols from the underlying alphabet. They also proceed by reading input symbols one at a time and moving from state to state according to them. However, the accepting condition is usually a variation of the following: an infinite word given as input is accepted if it makes the automaton pass through some accepting state infinitely often. This particular condition defines a *Büchi automaton* [Büc60]. It will be convenient to use a *generalized* version of Büchi automata where one has a number of accepting sets and the accepting condition requires that one visits a state from *each* of these sets infinitely often.

### 3.2.1 Model Checking Using Automata Theory

A Kripke structure $M = (S, S_0, R, L)$ with $AP$ as the set of atomic propositions is represented in a natural way using a Büchi automaton $\mathcal{A}$. The set of states of $\mathcal{A}$ is $S \cup \{q_i\}$ with $q_i$ as the initial state, and the alphabet is $2^{AP}$. There is a transition $(s, \alpha, s')$ from state $s \in S$ to state $s' \in S$ on input symbol $\alpha$ iff $R(s, s')$ holds and $L(s') = \alpha$. There are also initial transitions $(q_i, \alpha, s)$ where $s \in S_0$ and $\alpha = L(s)$. Finally, we make *all* states of $\mathcal{A}$ accepting. This results in the language $\mathcal{L}(\mathcal{A})$ accepted by $\mathcal{A}$ being exactly the set of possible behaviors of $M$.

Desired properties can also be specified using another automaton $\mathcal{S}$. We will shortly discuss how to convert any LTL specification into an equivalent Büchi automaton. Given $\mathcal{A}$ and $\mathcal{S}$, the model checking problem reduces to the language containment problem: Is $\mathcal{L}(\mathcal{A}) \subseteq \mathcal{L}(\mathcal{S})$? This can be rephrased as testing whether $\mathcal{L}(\mathcal{A}) \cap \overline{\mathcal{L}(\mathcal{S})} = \phi$. Since Büchi automata are closed under intersection and complementation, we can model check the original system by performing these two operations and checking if the language of the resulting automaton is empty. Emptiness testing can be done by identifying strongly connected components of the automaton and checking if the initial state is in the same component as some accepting state. If the resulting language is not empty, we need to produce a counter example, which is actually an infinite word. This, however, turns out to be feasible because if there is a counterexample, then there also exists one with a succinct description of the form $uv^{\omega}$, where $u$ and $v$ are finite words and $\omega$ denotes infinite repetition.

The complementation step in this process raises an efficiency issue. Although Büchi automata are closed under complementation, the resulting automata can be exponentially larger. This problem is overcome by making the user specify the complement automaton $\overline{\mathcal{S}}$ directly, giving *bad behaviors* of the system as input instead of acceptable ones. In the case where $\mathcal{S}$ was obtained automatically from a temporal logic formula $f$, one can straight away obtain $\overline{\mathcal{S}}$ by starting with $\neg f$ instead of $f$.

### 3.2.2 Translating LTL Specification into Automaton

In this section, we briefly discuss an algorithm for translating a given LTL specification into an equivalent generalized Büchi automaton [GPVW95]. Given a restricted path formula $g$, we first convert it into *negation normal form* by using operator dualities to push all negations $\neg$ to the innermost level, next to atomic propositions. All operators are then rewritten in terms of $\mathbf{U}, \mathbf{R}$ and $\mathbf{X}$. The algorithm starts with a node *init* that has associated with it three sets of subformulas of $g$. *New* $= \{g\}$ is the target set for this node, *Old* $= \phi$ denotes what has already been processed when reaching this node, and *Next* $= \phi$ is what we require successors of this node to have as their target. For a concrete example, let $g = A\,\mathbf{U}\,(B\,\mathbf{U}\,C)$. The algorithm proceeds by taking any node whose *New* field is not a simple atomic proposition and *expanding* it. This is best explained by the example at hand. The node *init* here would expand to have two successor nodes, one with fields (*New* $= \{A\}$, *Old* $= \{A\,\mathbf{U}\,(B\,\mathbf{U}\,C)\}$, *Next* $= \{A\,\mathbf{U}\,(B\,\mathbf{U}\,C)\}$) representing the case where $A$ holds in the next state and $g$ recursively holds in the state after it, and a second one with fields (*New* $= \{(B\,\mathbf{U}\,C)\}$, *Old* $= \{A\,\mathbf{U}\,(B\,\mathbf{U}\,C)\}$, *Next* $= \phi$) representing the case where $(B\,\mathbf{U}\,C)$ holds in the next state and nothing is required of the states after it. The other two temporal operators $\mathbf{R}$ and $\mathbf{X}$ and the three Boolean operators are handled in a similar fashion.

Starting with the *init* node with $g$ as the *New* field, the algorithm recursively expands nodes until all of the unexpanded ones have simple atomic propositions as their target. All these nodes form the states of our generalized Büchi automaton. The alphabet consists of sets of atomic propositions, thought of as indicating those propositions that are true. There is a transition from state $r$ to state $r'$ labeled $\alpha$ if and only if $r'$ is a successor of $r$ from the expansion process and $\alpha$ satisfies all propositions in *Old*$(r')$. There is one accepting set for each occurrence of $\mathbf{U}$ in $g$. For a subformula $g' = g_1\,\mathbf{U}\,g_2$, the accepting set contains all states $r$ such that either $g_2 \in$ *Old*$(r)$ implying that this subformula has already been satisfied, or $g' \notin$ *Old*$(r)$ implying that $g'$ was never required to hold in the previous state.

### 3.2.3 Complexity

We first note that union and intersection of two Büchi automata require size the product of the sizes of the two using standard product construction. Language emptiness can be checked in linear time using depth first search type algorithms to identify strongly connected components. Hence, this model checking procedure requires time $O(|\mathcal{A}| \times |\mathcal{S}|)$, where $|\mathcal{A}|$ and $|\mathcal{S}|$ are the sizes of the system and the specification automata, respectively. $|\mathcal{A}|$ is linear in the size of the Kripke structure $M$ it represents. If $\mathcal{S}$ is obtained from an LTL formula $f$, $|\mathcal{S}| = 2^{2|f|}$, resulting in net complexity $O(|M| \times 2^{2|f|})$.

# 4 Symbolic Model Checking

Explicit state representation of Kripke structures as labeled, directed graphs results in intuitive algorithms for checking temporal properties based on standard graph operations. Although the graph operations needed for these are efficient in terms of the size of the graph, the state transition graph itself becomes enormous for most practical problems. This calls for relatively concise *symbolic representations*. Instead of working with explicit states and transitions, we use compact structures that manipulate *sets* of states in an indirect way.

## 4.1 Binary Decision Diagram: BDD

One of the most influential ideas that made model checking practical was the use of Binary Decision Diagrams or BDDs [Bry86] for symbolically encoding state transition graphs and specifications [BCM$^+$92]. BDDs are like binary decision trees where nodes represent variables branched upon, 0/1 edge labels represent the value of the variable in the corresponding subtree, and 0/1 leaf labels denote the value of the function on any assignment to the variables consistent with the path from the root to that leaf. The only difference from decision trees is that BDDs are allowed to *reuse* nodes. Their underlying structure is a directed acyclic graph instead of a tree. For our discussion, we will assume BDDs are *ordered*, meaning that there exists a total order on the variables that every path respects. These are also referred to as OBDDs in the literature.

The size of a BDD representing a given Boolean function varies substantially with the order of variables used to branch upon. However, if we fix the order, there is a *canonical* BDD for any given function (up to trivial simplifications). This makes BDDs highly suitable for satisfiability and equivalence checks. Furthermore, under good variable orderings, Boolean formulas often have a much smaller representation as

BDDs than as conjunctive and disjunctive normal forms. A property that often makes them more attractive than circuits (which are also more compact than CNF and DNF forms) is the complexity of evaluating the underlying formula on a given variable assignment. While this is linear in the size of the circuit, it is linear only in the number of variables of the BDD irrespective of the size of the BDD itself.

### 4.1.1 Operations on BDDs

For many applications including model checking, it is important to be able to start with BDDs representing two Boolean functions $f_1$ and $f_2$ and combine them efficiently to obtain one that represents $f_1 \star f_2$, where $\star$ denotes any one of $2^{2^{|\{0,1\}|}} = 16$ logical operators. Projection operators such as $f_1 \star f_2 \stackrel{\text{def}}{=} f_1$ are trivially handled by picking the right BDD. Negation, $\neg f_1$, which is also one of the 16 two-argument logical operators, can actually be handled very easily – we simply negate the value of each terminal node of the BDD representing $f_1$.

For all other binary operators, we use a uniform recursive method called *Apply* [Bry86] that uses Shannon expansion of Boolean functions: $f = (\neg x \wedge f|_{x \leftarrow 0}) \vee (x \wedge f|_{x \leftarrow 1})$. If one of $f_1$ and $f_2$, say $f_1$, is the constant function, *i.e.* it is represented by a single terminal node BDD, *Apply* simply relabels each terminal node $t$ of the BDD representing $f_2$ with $value(f_1) \star value(t)$. Otherwise, let $x_1$ and $x_2$ be the variables branched upon at the roots of the BDDs representing $f_1$ and $f_2$, respectively. Recall that we have a branching order on the variables. If $x_1 = x_2 = x$, *Apply* outputs a BDD branching on $x$ with $f_1|_{x \leftarrow 0} \star f_2|_{x \leftarrow 0}$ and $f_1|_{x \leftarrow 1} \star f_2|_{x \leftarrow 1}$ as its two children. BDDs for these subformulas are computed in a recursive manner. If $x_1 < x_2$, the function $f_2$ must be independent of $x_2$ because of the fixed branching order. In this case, the root branches upon $x_1$ and the two children are $f_1|_{x_1 \leftarrow 0} \star f_2$ and $f_1|_{x_1 \leftarrow 1} \star f_2$. The case where $x_1 > x_2$ is handled similarly.

Even though each call above generates two recursive subproblems, *Apply* can be performed in time $O(|f_1| \times |f_2|)$ using dynamic programming. Here $|f_i|$ denotes the size of the BDD representation of $f_i$. The dynamic programming table has one entry for each pair $(g_1, g_2)$, where $g_i$ denotes the function represented by a sub-BDD of the BDD for $f_i$. Such a sub-BDD is uniquely determined by its root, for which there are $|f_i|$ choices.

Two other operations often used in model checking applications are *substitution* and *existential* quantification. Given a BDD for $f$, a BDD for $f|_{x \leftarrow 0}$ or $f|_{x \leftarrow 1}$ is created by simply replacing every node, if any,

branching on $x$ with its 0 or 1 successor, respectively. Existential quantification $\exists x \,.\, f$ is handled by creating a BDD for the equivalent formula $f|_{x \leftarrow 0} \vee f|_{x \leftarrow 1}$.

### 4.1.2 Model Checking Using BDDs

The basic tool required for model checking using BDDs is the *fixpoint* characterization of temporal logic operators. Let $M = (S, S_0, R, L)$ be a Kripke structure. A set $S' \subseteq S$ is called a fixpoint of a function $\tau : \mathcal{P}(S) \rightarrow \mathcal{P}(S)$ if $\tau(S') = S'$. Here $\mathcal{P}(S)$ denotes the power set of $S$. For this section, we will identify $S'$ with its characteristic function $I_{S'}$, which can also be thought of as a *predicate* on $S$ true exactly for the states that belong to $S'$. Accordingly, $\tau$ can be thought of as a *predicate transformer*. The elements of $\mathcal{P}(S)$ form a partial ordering under set inclusion. The least element of this ordering, the empty set, can also be thought of as the FALSE predicate. Similarly, the greatest element, which is $S$ itself, can be thought of as the TRUE predicate.

We will use simple $\mu$-calculus notation for fixpoint characterization. It can be shown that if the predicate transformer $\tau$ is *monotonic, i.e.* $P \subseteq Q$ implies $\tau(P) \subseteq \tau(Q)$, then it always has a unique least fixpoint $\mu Z \,.\, \tau(Z)$ and a unique greatest fixpoint $\nu Z \,.\, \tau(Z)$. Assuming certain continuity properties that always hold for finite state Kripke structures we are concerned with, it is further true that

$$\begin{aligned} \mu Z \,.\, \tau(Z) &= \bigcap \{Z \mid \tau(Z) \subseteq Z\} = \bigcup_i \tau^i(\text{false}) \\ \nu Z \,.\, \tau(Z) &= \bigcup \{Z \mid \tau(Z) \supseteq Z\} = \bigcap_i \tau^i(\text{true}) \end{aligned}$$

where $\tau^i$ denotes $i$ repeated applications of $\tau$. Given monotonicity of $\tau$, this characterization gives us trivial iterative algorithms to compute its least and greatest fixpoints – start with FALSE or TRUE depending on what you want to compute and keep applying $\tau$ until there is no change in the predicate. This convergence testing is where canonicity of BDDs comes handy. All we need to check is whether the BDDs representing consecutive approximations are *exactly* the same.

We now describe how one uses fixpoint computations to check whether a Kripke structure satisfies a given specification. For this section, we will assume that the specification is given in CTL form, using Boolean connectives along with compound operators $\mathbf{EX}, \mathbf{EG}$ and $\mathbf{EU}$. We will identify a CTL formula $f$ with the set of all states in which $f$ holds and recursively create a BDD for the characteristic function of this

set. The BDD for an atomic proposition simply represents the set of states where that atomic proposition is true. Given BDDs representing the sets of states where $f_1$ and $f_2$ hold, respectively, we can use the *Apply* operation described earlier to obtain a BDD representing the set of states where Boolean combinations such as $f_1 \wedge f_2$ hold. The three compound operators are a little more involved. Given a BDD for $g$, the BDD representing the set of states where $\mathbf{EX}\, g$ holds is the BDD for the function $\exists \bar{x}' \,.\, g(\bar{x}') \wedge R(\bar{x}, \bar{x}')$, which can be constructed using standard BDD operations seen earlier. Next, we claim that the set of states where $\mathbf{EG}\, g$ holds is exactly the greatest fixpoint of the predicate transformer $\tau(Z) = g \wedge \mathbf{EX}\, Z$, that is, $\mathbf{EG}\, g = \nu Z \,.\, g \wedge \mathbf{EX}\, Z$. To see this, we first observe that $\tau$ is monotonic. One can also verify that $\mathbf{EG}\, g$ is *a* fixpoint of $\tau$ by simply using the definition of the three operators $\mathbf{E}, \mathbf{G}$ and $\mathbf{X}$. That it is in fact the *greatest* fixpoint of $\tau$ can be shown using the characterization of the greatest fixpoint as $\cap_i \tau^i(true)$. We will omit the details here (see [CGP99]). One can similarly show that $\mathbf{E}[g_1\, \mathbf{U}\, g_2]$ is the least fixpoint of the predicate transformer $\tau(Z) = g_2 \vee (g_1 \wedge \mathbf{EX}\, Z)$. Combining everything, we have an algorithm for generating a BDD that characterizes exactly those states of $M$ where the CTL specification $f$ holds.

### 4.1.3 Complexity

Fixpoint computations take at most as many iterations as there are states in the fixpoint set, and each iteration requires $O(|M|)$ time. The complexity of fixpoint computations can therefore be bounded by $O(|M|^2)$. Boolean operations can be performed using *Apply* which takes time the product of the sizes of the two underlying subformulas. Hence, in the worst case, model checking using BDDs has complexity $O(|M|^2 \times 2^{|f|})$. However, in practice, it turns out to be feasible for many real world domains where any method based on explicit state representation fails.

## 4.2 Propositional Formula Satisfiability: SAT

Propositional formula satisfiability or SAT is a well studied NP-complete problem for which several heuristic algorithms have been developed over the last few decades. These include systematic local search procedures such as the *tableau* method [CA93], SATO [Zha97] and GRASP [MSS96] based on the Davis Putnam procedure [DP60, DLL62], as well as stochastic methods employing randomized search for solutions such as GSAT [SLM92] and WALKSAT [SK93].

### 4.2.1 Bounded Model Checking

We saw earlier a way to represent any Kripke structure $M = (S, S_0, R, L)$ as a Boolean formula over two sets of state variables. The technique of Bounded Model Checking [BCC+99] extends this idea. Here we seek counterexamples of a *fixed* length $k$ in the system represented by $M$. Accordingly, we have $k+1$ sets of state variables $\bar{x}_0, \bar{x}_1, \ldots, \bar{x}_k$ representing $k$ steps of the system. Using terminology from Section 2.1.2 and *unrolling* the transition relation $R$ $k$ times, we have a Boolean formula

$$[\![M]\!]_k \stackrel{\text{def}}{=} I_{S_0}(\bar{x}_0) \wedge \bigwedge_{i=0}^{k-1} I_{\hat{R}}(\bar{x}_i, \bar{x}_{i+1})$$

over the state variables $\bar{x}_i$ which is satisfiable iff $\bar{x}_0$ represents an initial state and the system proceeds in a valid manner according to $R$ for the initial $k$ steps.

To use SAT procedures for model checking $M$, we must first encode the given specification as a propositional formula over the $\bar{x}_i$'s. In this section, we will examine the *existential LTL model checking problem*, namely the case where the specification is given as an LTL formula $\mathbf{E}\, g$. We will show how to generate a propositional formula $[\![M, g]\!]_k$ which is satisfiable if and only if there exists a valid path $\pi$ in $M$ along which $g$ holds. Note that if we know an upper bound on $k$, a solution to this problem also implies a solution to any *universal* LTL model checking problem because of the duality $\mathbf{A}\, g = \neg \mathbf{E} \neg g$. Recall from LTL restrictions that $g$ is not allowed to have any path quantifiers $\mathbf{A}$ or $\mathbf{E}$, and is defined recursively in terms of Boolean operators and the five temporal operators. Boolean operators are handled in a trivial way when constructing the propositional form. We briefly discuss how temporal properties are translated.

We first make a crucial observation that influences the translation below. While the two existential properties $\mathbf{F}$ and $\mathbf{U}$ can be verified by identifying a "witness" state within the bound $k$, the two global properties $\mathbf{G}$ and $\mathbf{R}$ cannot. Bounded model checking gets around this obstacle by using the fact that information about an entire infinite path may be contained in a finite prefix thereof if there is a *back loop* in the corresponding state sequence. This allows us to check for global properties by looking only at finite path prefixes.

We will specify two different translations depending on whether the witness path $\pi$ for the existential problem has a back loop or not. For a restricted path formula $h$ in LTL, let $[\![h]\!]^i_k$ denote the result of its translation into a propositional formula when $\pi$ does not have a loop, and ${}_l[\![h]\!]^i_k$ denote the corresponding

translation when $\pi$ does have a loop starting at the $l^{th}$ state along $\pi$, *i.e.* $R(k,l)$ holds. Here $i$ denotes a temporary index referring to the $i^{th}$ state along $\pi$ and is used for recursive definitions. The translation will be on the conservative side, *i.e.* if the translated formula is true, the original LTL formula would hold even without any bound $k$, though the converse may not be true.

We will first do the case where there is no loop. Here $[\![\mathbf{G}\, h]\!]_k^i = \text{FALSE}$ because no global properties can be asserted confidently. $[\![\mathbf{X}\, h]\!]_k^i$ is FALSE if $i = k$ and $[\![h]\!]_k^{i+1}$ otherwise. $[\![h_1\, \mathbf{U}\, h_2]\!]_k^i$ says that there exists $j, i \leq j \leq k$, such that $[\![h_2]\!]_k^j$ as well as all of $[\![h_1]\!]_k^p$, $i \leq p < j$ are true. Operators $\mathbf{F}$ and $\mathbf{R}$ are translated in a similar way. Note that truth of these propositional translations implies that the corresponding LTL formulas hold, but not vice versa. For the loop case, we know the complete information about the path and hence the translation is exact. ${}_l[\![\mathbf{G}\, h]\!]_k^i$ is now the conjunction of all ${}_l[\![h]\!]_k^j$ for $min(i,l) \leq j \leq k$. ${}_l[\![\mathbf{X}\, h]\!]_k^i = {}_l[\![f]\!]_k^{succ(i)}$ where the successor $succ(i)$ of state $i$ along $\pi$ is defined in the natural cyclic way. The other three operators are also defined in a similar manner. Finally, we have formulas ${}_lL_k \stackrel{\text{def}}{=} I_{\hat{R}}(\bar{x}_k, \bar{x}_l)$ indicating whether there is a loop starting at state $l$ along $\pi$. $L_k$ is defined to be the disjunction of all these formulas.

We are now ready to give the complete translation of an existential LTL model checking problem into a propositional formula:

$$[\![M,g]\!]_k \stackrel{\text{def}}{=} [\![M]\!]_k \wedge \left( \left( \neg L_k \wedge [\![g]\!]_k^0 \right) \vee \bigvee_{l=0}^{k} \left( {}_lL_k \wedge {}_l[\![g]\!]_k^0 \right) \right)$$

Intuitively, this formula asserts that $M$ behaves in a valid way for the first $k$ steps and that the correct translation of $g$ is used depending on whether there is a loop or not.

### 4.2.2 Complexity

The size of $[\![M,g]\!]_k$ is polynomial in the size $|g|$ of the formula if common subformulas are shared, which is done in the implementation BMC given by [BCCZ99]. It is quadratic in the unrolling length $k$ and linear in the size $|M|$ of the Kripke structure. Bounded model checking proceeds by incrementing the bound $k$ in an exponential manner, looking for a value for which $[\![M,g]\!]_k$ is satisfiable. As such, this procedure is incomplete – one will never know when to stop and declare that $[\![M,g]\!]_k$ is unsatisfiable for all $k$ in case it is. However, one can show that an LTL formula $\mathbf{E}\, g$ holds in $M$ iff there exists $k \leq |M| \times 2^{|g|}$ such that $[\![M,g]\!]_k$ is satisfiable. This makes the procedure complete. There are also better bounds in terms of the

*loop diameter* of a $M$ [BCCZ99], which is the least number $d$ such that if a state $s'$ of $M$ is reachable from another state $s$ via some path, then it is also reachable via a path of length at most $d$.

## 4.3 Boolean Expression Diagram: BED

BDDs have been traditionally used for model checking because of their compactness, canonicity and ease of manipulation. However, for many real world problems, the intermediate and sometimes even the initial BDDs are too large to handle. In some of these harder domains, general SAT procedures using stochastic or other techniques turn out to be more useful. However, they are worse than BDDs on certain other domains. Boolean Expression Diagrams or BEDs [AH97] can be used to combine the advantages of both these techniques [WBCG00, WAH01]. The novelty comes from using a simple new representation very similar to BDDs, except that nodes are now allowed to have Boolean connectives instead of the usual one variable branch. Any Boolean function $f$ represented by a BDD with $x$ as the first variable branched on can be written recursively in its Shannon expansion form $(\neg x \wedge f|_{x \leftarrow 0}) \vee (x \wedge f|_{x \leftarrow 1})$. We can use this expansion and a few more Boolean operators corresponding to new $\vee, \wedge$ and $\neg$ nodes to obtain the Boolean function represented by any given BED. This allows us to use standard SAT procedures for BED model checking. On the other hand, as we will shortly see, there is a simple *up-one* procedure whose repeated application can convert any BED to an equivalent BDD. This lets us utilize the canonicity and compactness of BDDs to do BED model checking as well.

Unlike BDDs, BEDs are not canonical. This makes direct satisfiability and equivalence checks harder. In particular, the convergence test used in fixpoint algorithms is now more complicated. However, lack of canonicity also often makes the representation much more compact and allows us to deal with problems for which even the initial BDDs are exceedingly huge. An example of this is the multiplier function which requires exponential size BDD representation under any variable ordering [Bry86] but has a quadratic size BED computing it. This latter result follows from the facts that there is a small circuit for multiplication and Boolean circuits can be trivially converted into BEDs without any size blowup. One should remember, however, that although BEDs are more succinct than BDDs for many natural functions, a simple counting argument shows that almost all Boolean functions require exponential size BEDs [AH97].

### 4.3.1 Operations on BEDs

We now describe how to carry out useful operations on BEDs in an efficient manner. Given two BEDs representing Boolean functions $f_1$ and $f_2$, and a binary Boolean operator $\star$, we no longer need the relatively complicated *Apply* operation used in combining BDDs. Instead, we simply create a node labeled $\star$ and make the two given BEDs its children. Negation $\neg$ can be handled in a similar way.

A key operation whose repeated application converts any BED to a BDD is *up-one* [AH97]. This is useful when one wishes to perform satisfiability, equality and other checks for which BDDs are highly suited. Let $x \rightarrow f_1, f_0$ denote the Boolean function $(x \wedge f_1) \vee (\neg x \wedge f_0)$. This naturally corresponds to a BDD branch on a variable $x$ if $f_1 = f|_{x \leftarrow 1}$ and $f_0 = f|_{x \leftarrow 0}$ for some function $f$. One can easily verify that for any Boolean operator $\star$,

$$(x \rightarrow f_1, f_0) \star (x \rightarrow g_1, g_0) \equiv x \rightarrow (f_1 \star g_1), (f_2 \star g_2)$$

The *up-one* operation uses this identity to move the branching variable $x$ up one level above the operator $\star$. Repeated applications eventually bring $\star$ down to the lowest level, at which point it is directly applied to the values at the two terminal nodes and eliminated from the BED.

BEDs allow for a more general substitution operation. Given a BED for $f$, a variable $x$ and any Boolean formula $\varphi$ represented by a BED with root $u$, the BED for $f|_{x \leftarrow \varphi}$ is obtained by first using *up-one* to bring $x$ to the root $v$ with children $l$ and $h$, and then replacing $v$ with the BED representing $(\neg u \wedge l) \vee (u \wedge h)$. Existential quantification on a variable $x$ is handled by first performing *up-one* on the BED for $f$ to bring $x$ to the root. The new BDD for $\exists x \,.\, f$ is obtained by simply relabeling the root with the operator $\vee$ instead of the branch variable $x$. This is valid because $\exists x \,.\, x \rightarrow f, g$ is equivalent to $f \vee g$.

### 4.3.2 Model Checking Using BEDs

BEDs can be used for model checking finite state transition systems in a way similar to BDDs by utilizing fixpoint characterization of temporal logic operators. The only difference is that we now can use slightly more efficient and succinct BED operations to handle Boolean and temporal operators. For instance, a BED for $f_1 \vee f_2$ is simply a node labeled $\vee$ with the BEDs for $f_1$ and $f_2$ as its two children. Similarly, a BED for $\mathbf{EX}\, g = \exists \bar{x}' \,.\, g(\bar{x}') \wedge R(\bar{x}, \bar{x}')$ is obtained by applying existential quantification to the BED consisting of a

node labeled $\wedge$ with BED representations of $R$ and $g$ as its two children.

This last example motivates a new encoding for the transition relation $R$ that leads to faster model checking algorithms. Instead of specifying $R$ in a purely relational form, we can also specify parts of it as *functional assignment* to the next state variables. More precisely, for a partition ${}_{rel}\bar{x} \oplus {}_{fun}\bar{x}'$ of the next state variables $\bar{x}'$,

$$R(\bar{x}, \bar{x}') \;=\; {}_{rel}R(\bar{x}, {}_{rel}\bar{x}') \;\wedge\; \bigwedge_i \left({}_{fun}x_i' \Leftrightarrow f_i(\bar{x})\right)$$

Suppose the complete transition relation is specified in the functional form. Then the representation of $\mathbf{EX}\, g$ as $\exists \bar{x}' \,.\, g(\bar{x}') \wedge R(\bar{x}, \bar{x}')$ translates into $\exists \bar{x}' \,.\, g(\bar{x}') \wedge \bigwedge_i (x_i' \Leftrightarrow f_i(\bar{x}))$. This can now be re-written in a much simpler way as $g[f_i(\bar{x})/x_i']$, converting existential quantification into substitution. This *quantification-by-substitution* rule $\exists x \,.\, g \wedge (x \Leftrightarrow f) \equiv g[f/x]$ turns out to be a useful tool in a couple of other places also such as set inclusion testing and iterative squaring to reach fixpoints faster. For details see [WBCG00].

# 5 The Planning Problem and SAT procedures

A problem related to model checking and well studied in the artificial intelligence community because of its hardness as well as practical applications is *Planning*. The planning problem captures finite state systems with a set of *fluents* or predicates that may or may not hold at a given state, and a set of available *actions* that can be performed at any time their *precondition* is met, resulting in a set of *effects*. There are *goal* states that one tries to reach, respecting conflict rules that specify which actions or fluents cannot occur simultaneously. The aim is to come up with a *control strategy* or a plan specifying a sequential or parallel set of actions that will take the system from an initial state to the goal states. This generic formulation is used widely to study planning problems and has a succinct representation based on the language used in the STRIPS system [FN71]. The challenge in planning comes from the vastness of the underlying space of possible plans and the large number of options available at each step of the search.

Planning problems have been traditionally handled using specialized planning systems that utilize domain knowledge in a heuristic way. They typically proceed by viewing the state space as a (symbolic) graph with fluents labeling nodes and actions labeling edges. Heuristics are used to guide a backtrack or forward-chaining search looking for a valid path from the initial to the goal states. These heuristics typically use domain specific knowledge to estimate the distance to goal states. A drawback of most of these heuristic

methods is that if the goal is not achievable, the only way to detect this is to exhaustively enumerate the whole state space, which is usually not feasible. Moreover, heuristics require manual insight and may not always work to our benefit.

A more systematic as well as automatizable approach is that of the GRAPHPLAN system [BF97]. It converts STRIPS style planning problems into a new data structure called a *planning graph*, which is a layered representation of the underlying state space. Each layer denotes a time step from the initial state. Nodes are either actions or fluents, and edges connect action nodes with their preconditions as well as effects. A planning graph can be viewed as the state transition graph unrolled from the initial states a fixed number $t$ of steps. GRAPHPLAN proceeds by picking a value for $t$ and searching for a plan of length $t$. If it is successful, it tries for a shorter plan, and if not, a higher value of $t$ is checked. Plan search is done by generating *mutex* constraints that say which pairs of fluents or actions cannot occur simultaneously, and using them to prune the search space.

## 5.1 Planning Using SAT Procedures

Until mid-1990's, it was widely believed that planning problems require specialized systems performing a systematic state space search using domain knowledge. The development of fast satisfiability solvers and efficient encodings of planning problems as propositional formulas changed this. Although the general STRIPS-style plan existence problem is PSPACE-complete [Byl91], the corresponding problem where one seeks plans of polynomial length is (only) NP-complete. Hence there exists an encoding of planning problems as propositional satisfiability problems [KS92], from now on called SAT instances. It however turns out that the efficiency of SAT based planning procedures varies tremendously with the encoding used [KMS96].

SAT encodings for planning problems use a standard clausal representation. No syntactic distinction is made between fluent and action variables. Constraints are specified as simple clauses on these variables and there is no indication of what is a goal or what is a precondition. This allows SAT solvers to explore strategies other than state based backtracking or forward chaining. This leads to a potentially different kind of constraint propagation mechanism than what GRAPHPLAN like algorithms achieve. A second way in which SAT based planning methods differ from traditional specialized procedures is in the use of stochastic search. While planning is usually thought of as a systematic search through a huge space of possible plans,

randomized SAT algorithms such as WALKSAT [SK93] are in some cases able to lead the search away from a space of incorrect plans. As a combined effect of general encodings and randomization, SAT procedures often outperform specialized systematic planning algorithms on hard instances from many domains including blocks world, logistics and rocket controllers [KS96]. One should remember, however, that stochastic SAT procedures are inherently incomplete in that they never prove that a given formula is not satisfiable. One must revert to systematic procedures, SAT based or otherwise, to assert statements like no plan of a given length exists.

## 5.2 SAT Encodings

We briefly describe how STRIPS-style problems can be encoded as a propositional formula. We will use as an example the blocks world domain where one deals with stacks of blocks and can perform at any stage one of four block operations: `pickup, putdown, stack` and `unstack`. The aim is to find a plan, which is a sequence of these operations, that takes the blocks from a given initial state to a given goal state. In STRIPS notation, there are fluents such as `on(blockA,blockB,5)` holding at a particular instant and actions such as `pickup(blockC,4)` starting at the specified instant and ending in one time step. Fluents and actions can be thought of as describing axiom schemas such as `on(x,y,t)` and `pickup(x,t)`. When translating into a SAT instance, the maximum length of the plan sought is fixed and these axiom schemas are instantiated as propositional variables such as `pickup_blockC_4`. Desired properties of the system are specified as a propositional formula over these variables, and is then converted into a standard clausal form. For instance, to assert that every `pickup` is immediately followed by a `stack`, one can write a general schema `pickup(x,i)` $\Rightarrow$ $\exists$ `y.stack(x,y,i+1)` and instantiate it into constraints like `pickup_blockA_4` $\Rightarrow$ (`stack_blockA_stackP_5` $\vee$ `stack_blockA_stackQ_5`).

In addition to axioms specifying desired behavior, SAT framework often requires many additional constraints to rule out unintended models. For example, one might have pairs of conflicting actions that require a mutual exclusiveness constraint of the form $\neg$ `opA(t)` $\vee \neg$ `opB(t)`. Similarly, the fact that action `opA` requires fluents `flC` and `flD` as its precondition should be encoded explicitly as `opA(t+1)` $\Rightarrow$ (`flC(t)` $\wedge$ `flD(t)`). Such rules can be either written down manually by looking at the problem specification as done in *linear encodings* of [KS92], or generated automatically from the planning graph representation [KS96]. Yet another way, referred to as *state-based* encoding [KS96], specifies the same information

by focusing on what it means for a state to be valid rather than describing what each action does. As an example, instead of specifying explicitly what a `stack` operation does, we can have axiom schemas that generate constraints saying: if `on_blockA_blockB_3` is FALSE but `on_blockA_blockB_4` is TRUE, then `stack_blockA_stackP_3` must have occurred, given `blockB` was in `stackP`.

# 6 Summary

Model checking presents a promising tool that can be used for design validation as the need for high confidence in the correctness of automated systems rises with increasing computational power and more widespread use of such systems in critical domains. As a field of research, it has seen tremendous progress in the last twenty years, growing from tools that could verify systems with a million states to procedures that can now handle more than $10^{120}$ states. In the process, connections with automata theory, propositional satisfiability, logic and efficient data structures have been explored.

Finite automata on infinite words correspond naturally to finite state transition systems. Desired temporal properties can also be usually specified in terms of these automata. This reduces the problem of model checking to testing language containment given two automata, a solution to which is well known. What makes this approach even more attractive are efficient procedures to convert standard temporal logic specification into an equivalent automaton. Transition systems with several independent components can be specified as a group of component automata whose product, which never needs to be computed explicitly, represents the whole system. This leads to a relatively succinct representation.

While techniques based on explicit representation of reactive systems as graphs suffer from the state space explosion problem, they do provide a good understanding of model checking and form the basis for efficient algorithms we now have. BDD representation of Boolean functions corresponding to sets of states gives a much more succinct structure to handle and allows us to go well past the limit of earlier techniques. At the heart of this lie concise representation provided by BDDs and efficient algorithms to manipulate them. While canonicity of BDDs leads to trivial satisfiability and equality testing used frequently in model checking applications, it also results in requiring exponentially large size for BDD representation of some useful functions, such as multipliers.

Propositional satisfiability testing has seen many developments in the last thirty years and has always been a hot area of research. Dealing with an NP-complete problem, it naturally finds applications in many

fields of computation. Employing state of the art SAT solvers to do model checking has proved successful. The key idea here is a good propositional representation of both the system model as well as the specification. These representations typically fix the maximum number $t$ of system steps considered. This has twofold consequences: one, we might not be able to prove consistency of the system within $t$ steps, and two, if the system is not consistent, then we cannot infer this until we have tried all values for $t$ within a bound based on the system. This bound may or may not be small depending on the domain. Nevertheless, SAT procedures are able to handle some of the hard instances where BDD based techniques fail. In most such cases, canonicity of BDDs ends up requiring huge structures in the intermediate stages of model checking.

Model checking tools based on BDDs and SAT procedures complement each other, often providing fast solutions in domains where the other fails. BED representation unifies these two techniques through the use of a new data structure that is suitable for representing both BDDs and Boolean formulas. Clever but simple operations allow conversion from a given BED to either an equivalent BDD or a SAT instance. As a result, both BDD and SAT based techniques or a combination thereof, can be used with BEDs depending on the domain. An obvious issue here is the selection of the right sub-procedure given a model checking instance. BEDs are not canonical, making satisfiability and equality testing harder. In fact, one has to resort to relatively complicated SAT procedures for these. On the other hand, lack of canonicity also makes BEDs much more succinct even for functions that require huge BDD representation. BEDs therefore end up combining the advantages of the two underlying techniques and extending feasibility of model checking.

A related area is the use of general SAT procedures for solving planning problems. Planning is one of the heavily studied fields in artificial intelligence. Over the years, researchers have developed specialized systematic procedures for solving planning problems on particular domains. These use domain specific knowledge to guide the search in a vast space of potential plans. The key ideas that made the use of generalized stochastic SAT solvers possible for planning were smart encodings of planning problems as SAT instances and the inherently different search mechanism underlying randomized SAT procedures. Over the years, planning instances have come to use a very specialized notation that relates well to real life planning systems of interest but is far from being as general as a propositional formula. Conversion of this notation into a good propositional encoding, done either manually based on the domain or automatically from a planning graph representation, along with the development of fast SAT solvers have allowed SAT based techniques to compete, and in some cases even outperform, traditional systematic planning engines.

# References

[AH97] H. R. Anderson and H. Hulgaard. Boolean expression diagrams (extended abstract). In *12th Annual IEEE Symposium on Logic in Computer Science*, pages 88–98, July 1997.

[BCC+99] A. Biere, A. Cimatti, E. M. Clarke, M. Fujita, and Y. Zhu. Symbolic model checking using sat procedures instead of bdds. In *Proceedings of the 36th Conference on Design Automation*, pages 317–320, June 1999.

[BCCZ99] A. Biere, A. Cimatti, E. M. Clarke, and Y. Zhu. Symbolic model checking without bdds. In *Proceedings of the 5th International Conference on Tools and Algorithms for Construction and Analysis of Systems*, pages 193–207, Amsterdam, The Netherlands, Mar. 1999.

[BCM+92] J. R. Burch, E. M. Clarke, K. L. McMillan, D. L. Dill, and L. J. Hwang. Symbolic model checking: $10^{20}$ states and beyond. *Information and Computation*, 98:142–170, 1992.

[BF97] A. Blum and M. L. Furst. Fast planning through planning graph analysis. *Artificial Intelligence*, 90(1/2):281–300, 1997.

[BMP83] M. Ben-Ari, Z. Manna, and A. Pnueli. The temporal logic of branching time. *Acta Informatica*, 20:207–226, 1983.

[Bry86] R. E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Transactions on Computers*, 35(8):677–691, 1986.

[Büc60] J. R. Büchi. On a decision method in restricted second order arithmetic. In *International Congress on Logic, Methodology and Philosophy of Science*, pages 1–11. Stanford University Press, 1960.

[Byl91] T. Bylander. Complexity results for planning. In *Proceedings of the 12th International Joint Conference on Artificial Intelligence*, pages 274–279, Aug. 1991.

[CA93] J. M. Crawford and L. D. Auton. Experimental results on the crossover point in satisfiability problems. In *Proceedings, AAAI-93: 11th National Conference on Artificial Intelligence*, pages 21–27, Washington, DC, July 1993.

[CES86] 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):244–263, 1986.

[CGP99] E. M. Clarke, O. Grumberg, and D. A. Peled. *Model Checking*. The MIT Press, 1999.

[DLL62] M. Davis, G. Logemann, and D. Loveland. A machine program for theorem proving. *Communications of the ACM*, 5:394–397, 1962.

[DP60] M. Davis and H. Putnam. A computing procedure for quantification theory. *Communications of the ACM*, 7:201–215, 1960.

[FN71] R. E. Fikes and N. J. Nilsson. Strips: A new approach to the application of theorem proving to problem solving. *Artificial Intelligence*, 2(3/4):198–208, 1971.

[GPVW95] R. Gerth, D. Peled, M. Y. Vardi, and P. Wolper. Simple on-the-fly automatic verification of linear temporal logic. In *Protocol Specification, Testing and Verification*, volume 38, pages 3–18. Chapman & Hall, 1995.

[KMS96] H. A. Kautz, D. A. McAllester, and B. Selman. Encoding plans in propositional logid. In *Proceedings of the 5th International Conference on Principles of Knowledge Representation and Reasoning*, pages 374–384, Nov. 1996.

[KS92] H. A. Kautz and B. Selman. Planning as satisfiability. In *Proceedings of the 10th European Conference on Artificial Intelligence*, pages 359–363, Aug. 1992.

[KS96] H. A. Kautz and B. Selman. Pushing the envelope: Planning, propositional logic, and stochastic search. In *Proceedings, AAAI-96: 13th National Conference on Artificial Intelligence*, pages 1194–1201, Portland, OR, Aug. 1996.

[KS99] H. A. Kautz and B. Selman. Unifying sat-based and graph-based planning. In *Proceedings of the 16th International Joint Conference on Artificial Intelligence*, pages 318–325, Aug. 1999.

[MSS96] J. P. Marques-Silva and K. A. Sakallah. Grasp – a new search algorithm for satisfiability. In *Proceedings of the International Conference on Computer-Aided Design*, pages 220–227, San Jose, CA, Nov. 1996. ACM/IEEE.

[Pel98] D. A. Peled. Model checking using automata theory. In *Verification of Digital and Hybrid Systems, Editor M. Kemal Inan*. Springer-Verlag, 1998.

[Pnu81] A. Pnueli. The temporal semantics of concurrent programs. *Theoretical Computer Science*, 13:45–60, 1981.

[SK93] B. Selman and H. Kautz. Domain-independent extensions to GSAT: Solving large structured satisfiability problems. In *Proceedings of the 13th IJCAI*, pages 290–295, 1993.

[SLM92] B. Selman, H. J. Levesque, and D. G. Mitchell. A new method for solving hard satisfiability problems. In *Proceedings, AAAI-92: 10th National Conference on Artificial Intelligence*, pages 440–446, San Jose, CA, July 1992.

[VW86] M. Y. Vardi and P. Wolper. An automata-theoretic approach to automatic program verification. In *1st Annual IEEE Symposium on Logic in Computer Science*, pages 332–344, June 1986.

[WAH01] P. F. Williams, H. R. Anderson, and H. Hulgaard. Satisfiability checking using boolean expression diagrams. In *7th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, volume 2031 of *Lecture Notes in Computer Science*, pages 39–51. Springer-Verlag, 2001.

[WBCG00] P. F. Williams, A. Biere, E. M. Clarke, and A. Gupta. Combining decision diagrams and sat procedures for efficient symbolic model checking. In *Proceedings of the 12th International Conference on Computer Aided Verification*, pages 124–138, Chicago, IL, July 2000.

[Zha97] H. Zhang. Sato: An efficient propositional prover. In *Proceedings of the International Conference on Automated Deduction, LNAI*, volume 1249, pages 272–275, July 1997.