HELSINKI UNIVERSITY OF TECHNOLOGY
Department of Computer Science
Laboratory for Theoretical Computer Science
T-79.298 Postgraduate course in Digital Systems Science

Timo Latvala

Symbolic Model Checking and Partial Order Methods

Contents

1 Introduction

Model checking is a powerful technique for verifying the correctness of reactive systems. It can automatically prove that a model of a system has a desired property, which has been specified using a suitable temporal logic. Model checking also provides a counterexample to the given property if it does not hold, which is at least as important as its ability to automatically prove properties.

The applicability of model checking is, however, severely restricted by the state space explosion problem. For model checking to be truly useful, it must be possible to apply model checking to systems which are of relevant complexity. In other words, systems of the size which the industry currently produces.

The state space explosion problem can be explained by complexity theoretical arguments. Model checking is PSPACE-complete in the system description, when the system is described by a Petri net, a composition of parallel processes or other comparable formalisms. In many cases, notably for LTL and other more expressive logics, model checking is also PSPACE-complete in the length of the specification. Thus it is unlikely that we can find methods which will circumvent the state space explosion. However, methods for alleviating the state space explosion exist.

This work covers three different methods for alleviating the state space explosion in model checking. Two of the methods can be classified as symbolic: using binary decisions diagrams and bounded model checking. The third method, partial order reduction, is mostly applied in the domain of explicit state model checking.

We assume that the reader is familiar with propositional logic, and the basics of temporal logic and model checking.

2 Symbolic Model Checking

Symbolic model checking tries to alleviate the state space explosion problem by using efficient encodings of the state space. Instead of representing each state in the state space explicitly, the whole state space is encoded using an implicit representation. For model checking to be possible we also need a symbolic representation of the transition relation and the temporal operators.

One possible symbolic encoding is propositional formulae. There exist efficient techniques for manipulating boolean functions and propositional formulae. A propositional encoding can also be efficient: k variables can represent 2^k states. With a propositional encoding it straight forward to represent a shared variables program.

Consider a shared variables program P over the variables $V = \{v_1, v_2, \ldots, v_n\}$ with finite domains $\mathcal{D} = \{D_1, D_2, \ldots, D_n\}$. Any finite domain D_i can be encoded using $k_i = log_2|D_i|$ variables. Thus we can represent the state of the program with $m = n \sum_i k_i$ variables. The basic idea is that the set models of the propositional formula over $\mathcal{P} = \{v_1, \ldots, v_m\}$ represents a set of states of the program.

Example. Let $\mathcal{P} = \{v_1, v_2, v_3\}$. Then the formula $(v_1 \wedge v_2) \vee v_3$ represents the set $\{110, 001, 011, 101, 111\}$, where 0 stands for **false** and 1 for **true** and a string denotes the valuation for the variables in increasing index order.

Representing the transition relation of the program can be done by a propositional formula over $\mathcal{P} = \{v_1, \ldots, v_m, v'_1, \ldots, v'_m\}$. The primed variables represent possible values of the state variables reachable in one step.

Example. Consider the formula $R = (v_1 \leftrightarrow \neg v'_1) \wedge ((v_2 \rightarrow v_3) \leftrightarrow (v'_2 \wedge v'_3))$. Again the possible models represent the reachable states. From the state $v = v_1 \wedge \neg v_2 \wedge v_3$ the reachable states are characterised by $v' = \neg v'_1 \wedge v'_2 \wedge v'_3$.

Given a propositional formula, φ, representing a set of states and propositional formula, R, representing the transition relation, the states reachable in one step can be computed by existential quantification using substitution. Existential quantification is defined in the following way.

$$(\exists v \varphi) \leftrightarrow (\varphi\{v := \bot\} \vee \varphi\{v := \top\})$$

Let $\vec{v}$ denote all variables $v_1, \ldots, v_m$ and s be a propositional formula over $\vec{v}$ representing the current know set of states. The propositional expression representing the set of successors of s in terms of primed variables is

$$s' \equiv \exists \vec{v}(s \wedge R).$$

The formula is only over the variables $v'_1, \ldots, v'_m$ as the unprimed variables have been quantified away. The two sets s and s' can easily be combined into one set with variable substitution: $s \vee s'\{\vec{v'} := \vec{v}\}$.

Finding the shortest formula representing a given set is co-NP-hard. Therefore we need efficient methods for manipulating the formulae, which can become very large. The two methods presented next, approach the problem in different ways.

3 Binary Decision Diagrams

Using a normal form is one way of trying to keep the size of the formulae describing state space, transition relation and the temporal formulae small. A Binary Decision Diagram (BDD) is such a normal form.

We define the three-place connective $\mathbf{Ite}(\varphi, \psi_T, \psi_F)$ ('if-then-else') in the following way:

$$\mathbf{Ite}(\varphi, \psi_T, \psi_F) \stackrel{def}{=} (\varphi \wedge \psi_T) \vee (\varphi \wedge \psi_F)$$

Any propositional formula can be expressed using **Ite** and the constants $\top, \bot$. This can be easily be seen by the fact that $\varphi \rightarrow \psi \leftrightarrow \mathbf{Ite}(\varphi, \psi, \top)$. A boolean expression containing only **Ite**, $\top, \bot$ and variables, such that all tests are performed on variables is called the *If-then-else Normal Form* (INF). Consider the formula φ containing the variable v. The following equivalence holds.

$$\varphi \leftrightarrow \mathbf{Ite}(v, \varphi\{x := \top\}, \varphi\{x := \bot\})$$

This is called the Shannon expansion. Using the Shannon expansion the INF of any formula can be obtained. If the Shannon expansion is applied to a formula φ, we obtain two expressions $\varphi\{v_1 := \top\}$ and $\varphi\{v_1 := \bot\}$, which contain one variable less. By applying the expansion recursively to these two expressions, call them φ_1 and φ_0 we eventually get an expression which is in INF.

Example. Let $\varphi = (v_1 \leftrightarrow v_2) \wedge (v_3 \leftrightarrow v_4)$. We expand the variables in descending index order. Then the corresponding INF is:

$$\begin{aligned}
\varphi &= \mathbf{Ite}(v_4, \varphi_1, \varphi_0) \\
\varphi_1 &= \mathbf{Ite}(v_3, \varphi_{11}, \bot) \\
\varphi_0 &= \mathbf{Ite}(v_3, \bot, \varphi_{00}) \\
\varphi_{11} &= \mathbf{Ite}(v_2, \varphi_{111}, \varphi_{110}) \\
\varphi_{00} &= \mathbf{Ite}(v_2, \varphi_{001}, \varphi_{000}) \\
\varphi_{111} &= \mathbf{Ite}(v_1, \top, \bot) \\
\varphi_{110} &= \mathbf{Ite}(v_1, \bot, \top) \\
\varphi_{001} &= \mathbf{Ite}(v_1, \top, \bot) \\
\varphi_{000} &= \mathbf{Ite}(v_1, \bot, \top)
\end{aligned}$$

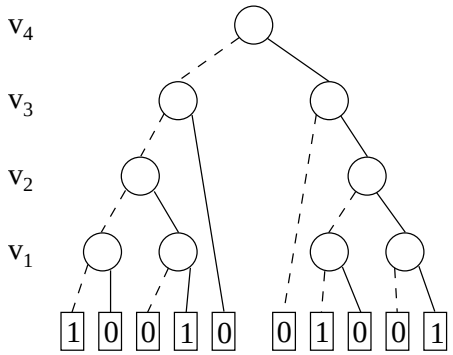

Figure 1: Decision diagram for $\varphi = (v_1 \leftrightarrow v_2) \wedge (v_3 \leftrightarrow v_4)$.

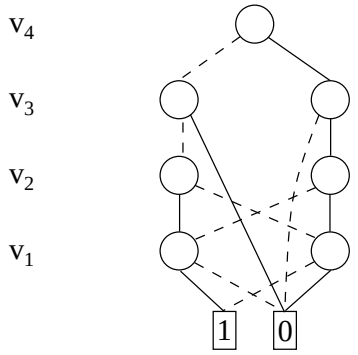

Figure 2: Binary Decision diagram for $\varphi = (v_1 \leftrightarrow v_2) \wedge (v_3 \leftrightarrow v_4)$.

The expression can be visualised as an expression tree called a *decision tree*. See Figure 1.

The decision tree contains a lot of redundancy. A BDD can be obtained from the decision tree by identifying all isomorphic subtrees and removing all redundant tests. Figure 2 shows the BDD obtained. The ordering of the variables follows the order in which the expression was expanded.

A BDD is a rooted, directed acyclic graph which has the following characteristics.

- There are one or two terminal nodes with zero outdegree labeled 0 and 1.
- Each variable node u is associated with a variable $var(u)$.
- Each variable node u has two outgoing edges $low(u)$ and $high(u)$

- All paths in the graph respect the given linear ordering $x_1 < x_2 < \cdots < x_n$.

We identify a BDD by its root node u. Each BDD node u with its children, can be treated as a separate BDD. A BDD φ^u defines a boolean function in the following way.

$$\begin{aligned} \varphi^0 &= 0 \\ \varphi^1 &= 1 \\ \varphi^u &= \mathbf{Ite}(var(u), \varphi^{high(u)}, \varphi^{low(u)}), u \text{ is a variable node.} \end{aligned}$$

A reduced BDD with a given variable ordering is canonical form for boolean functions $f : \mathbb{B}^n \to \mathbb{B}$. There is exactly one BDD representing each f.

Theorem (Canonicity). For any function $f : \mathbb{B}^n \to \mathbb{B}$ there is exactly one BDD u with variable ordering $v_1 < v_2 < \cdots < v_n$ such that $f^u = f(v_1, v_2, \ldots, v_n)$.

Proof: (sketch). The proof proceeds by induction on the number of arguments of f. For $n = 0$ the two possible boolean functions are true and false. Each of these have a unique BDD representation $\top$ and $\bot$. Since redundant tests are always removed, a BDD with a variable node must be non-constant. Let $f(v_1, \ldots, v_n, v_{n+1})$ be a function of $n+1$ arguments. Define $f_i(x_2, \ldots, x_{n+1}) = f(i, v_2, \ldots, v_{n+1}), i \in \mathbb{B}$. By the induction hypothesis both f_0 and f_1 have unique BDD representations u_0 and u_1 such that $f^{u_0} = f_0$ and $f^{u_1} = f_1$. By Shannon's expansion we have:

$$f(v_1, v_2, \ldots, v_{n+1}) = \mathbf{Ite}(v_1, f_1, f_2).$$

A simple case analysis ($u_0 = u_1$ and $u_0 \neq u_1$) shows that this resultant BDD is unique.

Canonicity has some implications. Checking if the formula represented by the BDD is satisfiable is a constant time operation. We just have to check that the BDD does not solely consist of the zero node. For sophisticated implementations, also comparing two BDDs for equality is a constant time operation.

3.1 Algorithms and Implementation

Constructing the corresponding BDD to formula by first making the decision diagram is very inefficient. This will always result in an exponential running

```
function PL2BDD (Formula φ) : (Nodeset, Bdd) =
    Nodeset table := {}; /*Table of BDD nodes*/
    Bdd max := 1;
    Bdd result := BDD(φ, 1);
    return (table, result)

function BDD(Formula φ, Bddvar i) : = Bdd
    if i > n then return eval(φ) /*φ; is constant*/
    else δ1 := BDD(φ{v_i := ⊥}, i + 1); δ2 := BDD(φ{v_i := ⊤}, i + 1);
        if δ1 = δ2 then return δ1;
        else if ∃δ : (δ, i, δ1, δ2) ∈ table then return δ;
        else max := max + 1; table := table ∪ {(max, i, δ1, δ2)};
            return max;
```

Figure 3: Creating a BDD from a formula φ

time w.r.t. the number of variables. The solution is to construct a BDD which is reduced on-the-fly.

To construct a reduced BDD, we must be able to identify isomorphic subtrees. For this purpose the set of BDD nodes is implemented as a hash table. Given that $\delta = \mathbf{Ite}(v, \delta_1, \delta_2)$ the hash table maps triples (v, δ_1, δ_2) to δ. v is the variable of the BDD, and δ_1 and δ_2 its children. Each BDD can be identified by its variable and two children.

A reduced BDD can now be created by recursively performing the Shannon expansion on the formula. If the two resulting branches are equal one of them is returned. Otherwise we check if the new BDD is in the table and return it if it is not. Figure 3 shows one possible way of converting a formula to a BDD. This algorithm performs much better in practice than the naive approach suggested above. The worst case performance is still exponential.

The size of the constructed BDD can greatly depend on the ordering of the variables. Consider a BDD for $(v_1 \leftrightarrow v_2) \wedge (v_3 \leftrightarrow v_4)$. As can be seen from Figure 2, when using the ordering $v_4 < v_3 < v_2 < v_1$ the BDD only has nine nodes while the ordering $v_4 < v_2 < v_3 < v_1$ depicted in Figure 4 results in a BDD with eleven nodes. For long formulas the difference is significant. A good ordering can result in a BDD which is linear w.r.t the number of variables while a bad ordering may result in an exponential BDD. Finding the optimal ordering is an NP-hard problem, and not feasible in practice. Heuristics for finding a good ordering can be used, but they will not

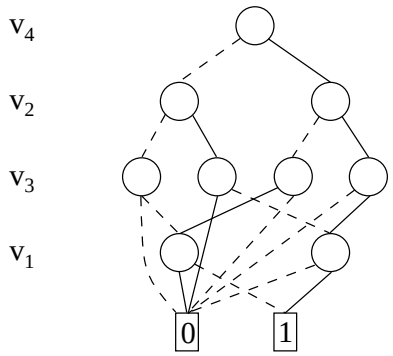

Figure 4: Binary Decision diagram for $\varphi = (v_1 \leftrightarrow v_2) \wedge (v_3 \leftrightarrow v_4)$ with another variable ordering.

always work. In general not all boolean expressions have small BDDs. There provably exists boolean expressions which always result in an exponential BDD, irrespectively of the variable ordering. Unfortunately these BDDs also appear in practice e.g. multiplier circuits.

In order for the BDD representation to be useful we must be able to apply boolean connectives on the two BDDs. In other words, we must be able to compute the BDD which is result of applying the connective to the propositional formulas the two original BDDs represent. If this can be done for implication, the other operands follow automatically. Figure 5 shows an algorithm to compute implication

The algorithm for computing the implication between two BDDs works in the following way. The first four lines of the algorithm deal with the cases where either φ or ψ is constant. The two first lines simply follow the definition of implication. The third line where $\psi = 0$ can be motivated by the equivalence.

$$(\mathbf{Ite}(v, \delta_1, \delta_2) \rightarrow) \;\leftrightarrow\; \mathbf{Ite}(v, (\delta_1 \rightarrow \bot, \delta_2 \rightarrow \bot)$$

The remaining lines are concerned with the case when both φ and ψ are in the table. By using the associativity of the **Ite**-operator the remaining cases recurse down the BDDs, advancing in the BDD which has the smaller variable in the variable ordering. The recursion must terminate as the terminal nodes of the BDDs are eventually reached.

Computing the implication is linear in the BDDs. Any boolean operation can be implemented in linear time. Bryant [3] has given a uniform scheme how to handle all 16 boolean operators. The function, BDDApply(op, Bdd φ, Bdd ψ), will not be presented here. Most BDD-packages use separate algorithms

```
function BDDImp (Bdd φ, Bdd ψ) : Bdd =
    if φ = 0 or ψ = 1 then return 1;
    else if φ = 1 return ψ;
    else if ψ = 0 and (ψ, i, φ1, φ2) ∈ table_φ
        then return new_node(i, BDDImp(φ1, 0), BDDImp(φ2, 0));
    else /*(φ, i, φ1, φ2) and (ψ, i, ψ1, ψ2)*/
        if (i = j) then return new_node(i, BDDImp(φ1, ψ1), BDDImp(φ2, ψ2));
        else if (i < j) then return new_node(i, BDDImp(φ1, ψ), BDDImp(φ2, ψ));
        else if (i > j) then return new_node(i, BDDImp(φ, ψ1), BDDImp(φ, ψ2));

function new_node(Bddvar i, Bdd δ1, Bdd δ2) : = Bdd
    if δ1 = δ2 then return δ1;
    else if ∃δ : (δ, i, δ1, δ2) ∈ table then return δ;
    else max := max + 1; table := table ∪ {(max, i, δ1, δ2)}; return max;
```

Figure 5: Computing the implication $\varphi \to \psi$ of two BDDs

```
function BDDExists (Bdd φ, Vars v⃗) : Bdd =
    if φ ∈ {0, 1} then return φ;
    else /*(φ, i, φ1, φ2) ∈ table*/
        δ1 = BDDExists(φ1, v⃗); δ2 = BDDExists(φ2, v⃗);
        if i ∈ v⃗ then return BDDApply(∨, δ1, δ2);
        else return new_node(i, δ1, δ2);
```

Figure 6: Computing the existential quantification $\exists \vec{v}\ \varphi$.

for each operator for increased efficiency. The special purpose algorithms can better make use of the symmetries and idempotencies present.

In the last section existential quantification was needed to compute one step reachability. In principal we already have everything need compute existential quantification as it was defined in the following way.

$$\exists v\ \varphi \leftrightarrow \varphi\{v := \top\} \vee \varphi\{v := \bot\}$$

Computing the existential quantification for a set of variables $\vec{v}$ could be done as a sequence of substitutions, but it would highly inefficient. The algorithm of Figure 6 shows how it can be done directly.

In contrast to performing a series of substitutions, the algorithm can quantify any of the variables on demand and therefore perform better. With existential quantification, all the pieces are now in place to facilitate symbolic model

checking.

4 Symbolic Model Checking for CTL

We now describe how symbolic model checking for CTL can be done. Essentially we describe algorithms for computing a BDD representation $\varphi^{\mathcal{F}}$ of the set states where the formula φ holds. Assume that the system is described by variables $\vec{v} = \{v_1, \ldots, v_n\}$. The transition relation R is over the variables $\{v_1, \ldots, v_n, v'_1, \ldots, v'_n\}$. For each $p \in \mathcal{P}$ a BDD is given which represents the set $\mathcal{I}(p)$. Computing the BDDs for the propositional case is easy. We simply use the algorithms described the in the previous section to combine BDDs. To evaluate a formula $\mathbf{E}(\psi_2 \mathbf{U} \psi_1)$ a similar iteration as in the explicit case must be done. We must compute the least fixpoint of the the set $\{w \mid \exists w'(w \prec w' \wedge (w' \in (\psi_1^{\mathcal{F}} \cup \psi_2^{\mathcal{F}} \cap E))\}$, where E is an intermediate result of the iteration. In BDD terms the we simply replace the set operations with boolean $\vee$ and $\wedge$, and $\prec$ is replaced the transition relation R. This can be described in the BDD terms the following way.

$$\begin{aligned} E_0(w') &= \emptyset \\ E_{i+1}(w) &= E_i(w) \vee \exists w'(R(w, w') \wedge (\psi_1^{\mathcal{F}}(w') \vee (\psi_2^{\mathcal{F}}(w') \wedge E_i(w')))) \end{aligned}$$

where $\psi_1^{\mathcal{F}}, \psi_2^{\mathcal{F}}$ and E are BDDs. As equality between BDDs can be detected in constant time, this fixpoint calculation is easy. To compute $\mathbf{A}(\psi_2 \mathbf{U} \psi_1)$ the least fixpoint of the set $\{w \mid \forall w'(w \prec w' \rightarrow (w' \in (\psi_1^{\mathcal{F}} \cup \psi_2^{\mathcal{F}} \cap E))\}$ must be computed.

5 Relational μ-calculus

The relational μ-calculus is rich logical language. It can be seen as a first order predicate logic with a recursion operator. The symbolic techniques presented previously can also be extended for model checking this expressive logic.

A collection of disjoint sets with a collection of relations over the sets is called a (typed) *structure*. The sets are referred to as *domains* and the elements of the sets as *objects*. A pair $\Sigma = (\mathcal{D}, \mathcal{R})$ is called a signature, where $\mathcal{D}$ is a finite set of *domain names* and $\mathcal{R}$ is is a set of *relation symbols*. Each relation

symbol has an associated type τ, which is a sequence of domain names. The names of the domains in the relation fix its type.

An *interpretation* $\mathcal{I}$ assigns a structure S to signature Σ. Formally, $\mathcal{I} : \Sigma \to S$ is a mapping which assigns a non-empty set to each domain name and relation of the appropriate arity and type to each relation symbol. For a relation R with type $\tau(R) = (D_1, \ldots, D_n)$ the interpretation is $\mathcal{I}(R) \subseteq \mathcal{I}(D_1) \times \cdots \times \mathcal{I}(D_n)$. When the interpretation of a predicate symbol R is a single domain, R is called a *constant*.

5.1 Syntax and Semantics

The syntax of the relational μ-calculus has two syntactic categories, *well-formed formulas* and *relation terms of type* τ. Let $\mathcal{V}$ be a set of variables with appropriate types from the signature Σ. Assume that the symbols $(,), \bot, \to, =, \exists, \mu, \lambda$ are not in the signature. A well-formed formula has the following syntax:

- $\bot$, $(\varphi \to \psi)$, where φ and ψ are well-formed formulas,
- $x_1 = x_2$, where x_1 and x_2 are individual variables of the same type,
- $\exists x\ \varphi$, where φ is a well-formed formula, and x is an individual variable, or
- $\rho\ x_1 \ldots x_n$, where ρ is a relation term of type $(D_1, \ldots, D_n)$, and x_i is an individual variable of type D_i.

The relation terms have their own syntax. Very complex relations can be formed using λ-*abstraction* or μ-*recursion*. A relation term of type $(D_1, \ldots, D_n)$ is

- a relation symbol R or a relation variable X of type $(D_1, \ldots, D_n)$,
- $\lambda x_1 \ldots x_n\ \varphi$, where φ is a well-formed formula and each x_i is an individual variable of type D_i, or
- $\mu X \rho$, where X is a relation variable of type $(D_1, \ldots, D_n)$, and ρ a relation term which is positive in X.

A relational term ρ is positive in X if every occurrence of X is under an even number of negation signs.

A *relational model* can now be defined for the calculus. A relation model $\mathcal{M} = (S, \mathcal{I}, \mathbf{v})$ for a signature Σ consists of a structure S, an interpretation $\mathcal{I}$ and variable valuation $\mathbf{v}$. The variable valuation assigns an object $\mathbf{v}(x) \in D$ to each variable x of type D. Each relation variable X of type $(D_1, \ldots, D_n)$ is assigned a relation $\mathbf{v}(X) \subseteq (D_1 \times \cdots \times D_n)$. The model also assigns a truth value $\varphi^{\mathcal{M}}$ to each formula φ. The semantics, also known as *denotation*, are as follows:

- $x^{\mathcal{M}} = \mathbf{v}(x)$, if x is an individual variable,
- $\bot^{\mathcal{M}} = \mathbf{false}$,
- $(\varphi \to \psi)^{\mathcal{M}} = \mathbf{true}$ iff $\varphi^{\mathcal{M}} = \mathbf{false}$ or $\psi^{\mathcal{M}} = \mathbf{true}$.
- $(x_1 = x_2)^{\mathcal{M}} = \mathbf{true}$ iff $x_1^{\mathcal{M}} = x_2^{\mathcal{M}}$,
- $(\exists x\ \varphi)^{\mathcal{M}} = \mathbf{true}$ iff $\varphi^{(S,\mathcal{I},\mathbf{v}')} = \mathbf{true}$ and $\mathbf{v}'$ differs from $\mathbf{v}$ at most in x.
- $(\rho x_1 \ldots x_n)^{\mathcal{M}} = \mathbf{true}$ iff $(x_1^{\mathcal{M}}, \ldots, x_n^{\mathcal{M}}) \in \rho^{\mathcal{M}}$,
- $R^{\mathcal{M}} = \mathcal{I}(R)$, if R is a relation symbol, i.e. the name is connected to the preselected interpretation,
- $X^{\mathcal{M}} = \mathbf{v}(X)$, if X is a relation variable,
- $(\lambda x_1 \ldots x_n \varphi)^{\mathcal{M}}) = \{(d_1, \ldots, d_n) \mid \varphi^{(S,\mathcal{I},\mathbf{v}')} = \mathbf{true}\}$ where $\mathbf{v}'$ differs form $\mathbf{v}$ only in the assignment of d_i to x_i, i.e. $(\lambda x_1 \ldots x_n \varphi)^{\mathcal{M}}$ is the relation consisting of all tuples of objects for which φ is **true**, and
- $(\mu X \rho)^{\mathcal{M}} = \cap\{Q \mid \rho^{\mathcal{F}}(Q) \subseteq Q\}$, where $\rho^{\mathcal{F}}(Q) = \rho^{(S,\mathcal{I},\mathbf{v}')}$, and $\mathbf{v}'$ differs from $\mathbf{v}$ only in $\mathbf{v}'(X) = Q$. $\mu X \rho$ is the least fixpoint of the functional $\rho^{\mathcal{F}}$.

The expressive power of the relational μ-calculus is between first-order logic and second order logic. The logic has existential quantification while the μ-recursion operator provides a restricted form of second order quantification. If λ-abstractions on relation variables were allowed, the result would be a second-order calculus. With the μ-recursion operator all recursive functions of arithmetic can be defined. This means that on infinite domains the relational μ-calculus has the expressive power of Turing machines. The addition-relation on natural numbers can be defined in the following way. Let

Z be the constant zero and S the successor relation. The addition-relation is defined by

$$\mu X(\lambda xyz(Zx \wedge y = z \vee \exists uv(Sux \wedge Svz \wedge Xuyv)$$

The formula essentially says that x, y and z are in the relation either if $x = 0$ and y equals z or there exists u and v such that $x = u + 1$, $z = v + 1$ and $u + y = v$.

5.2 Model Checking

For finite domains the model checking problem is polynomial in the size of the structure. Thus, given a relational frame $\mathcal{F} = (S, \mathcal{I})$ and a relational term ρ or a formula φ, model checking can be used to determine the denotation $\rho^{\mathcal{F}}$ or $\varphi^{\mathcal{F}}$.

The underlying implementation represents the different elements of the calculus in the following way. We assume for simplicity that all domains are binary. Non-binary domains are also possible by using an appropriate encoding. We modify our definition of BDDs somewhat to cope with relation names and symbols. BDDs are tuples $(\delta, i, \delta_1, \delta_2)$, where δ is the name of the node, i is a variable from the set $\{v_1, \ldots, v_n, x_1, \ldots, x_m\}$ and each δ_j is one of the constants 0 or 1, the name of a relation variable, or the name of another BDD node.

The interpretation $\mathcal{I}$ of a relation is BDD over the variables $v_1, \ldots, v_n$. Each v_i encodes one domain. A term or a formula with free individual variables $x_1, \ldots, x_m$ is represented as a BDD and BDD variables $x_1, \ldots, x_m$. Relation variables are represented simply by their names. As the variables can appear as successors in the BDDs, substitution is a simple matter of replacing the variable with a relation.

The algorithm which performs the model checking is called BDDForm and can it be seen in Figure 7. The algorithm recursively evaluates the given formula with a case analysis. Only the last case where a relational term is evaluated is a little complicated. The other cases are straightforward applications of the BDD algorithms presented earlier. For example, the equality test case builds a BDD which first tests if $x_1 = 1$ and then tests x_2 if it has the same value using three **Ite**-operators. The last case, where a relational term is evaluated, uses an auxiliary function BDDTerm. The function BDDTerm takes as arguments the relational term ρ and an interpretation $\mathcal{I}$. If ρ is a relation R or a relation variable X then the relation or the variable

```
function BDDForm (Formula φ, Interpretation I) : Bdd =
    /*Calculates the BDD of formula φ in the interpretation I*/
    case φ of
        x ∈ V : return Ite(x, 1, 0);
        (x_1 = x_2) : return Ite(x_1, Ite(x_2, 1, 0), Ite(x_2, 0, 1)) ;
        ⊥: return 0;
        (ψ_1 → ψ_2) : return BDDImp(BDDForm(ψ_1, I),
            BDDForm(ψ_2, I));
        ∃x φ : return BDDExists(x,BDDForm(φ, I));
        ρx_1 ... x_n : return BDDTerm(ρ, I){v_1 := x_1} ··· {v_n := x_n};

function BDDTerm(RelationalTerm ρ, Interpretation I) : = Bdd
    case ρ of
        R ∈ R: return I(R); /*pointer to BDD for R*/
        X ∈ V: return X; /*name of X*/
        λx_1 ... x_n φ : return BDDForm(φ, I){v_1 := x_1} ··· {v_n := x_n} ;
        μXρ : r :=BDDTerm(ρ, I); return BDDlfp(r, 0);

function BDDlfp(BDD r, BDD X^i) : BDD =
    X^{i+1} := r{X := X^i};
    if (X^{i+1} = X^i) then return X^i;
    else return BDDlfp(r, X^{i+1});
```

Figure 7: Algorithms for model checking the relational μ-calculus

name is returned. The third case, λ-abstraction uses BDDForm to return a BDD where the free individual variables have been replaced with $v_1, \ldots, v_n$. The last case, computing the least fixed point, can be done with standard techniques. First the a BDD for the functional ρ is computed. The least fixpoint is then computed by a series of approximations $X^0, X^1, \ldots, X^i, \ldots$, until $X^i = X^{i+1}$. Testing for equality of BDDs is a constant time operation for advanced implementations.

6 Bounded Model Checking

For some cases the BDD-based approach to model checking does not perform very well. There are systems for which an exponential BDD is required to represent the system w.r.t. the number of state variables.

An alternative approach to symbolic model checking is to encode the problem as an instance of propositional satisfiability and use state of the art satisfiability solvers to attack the problem. The encoding is possible for finite domains, as translating first-order logic to linear temporal logic is possible [4]. When the domains are finite, the existential quantifications can be replaced by a series of disjunctions. This approach avoids the use of a normal form and relies on the efficiency of the solver to deal with the complexity of the formulae. The main advantage is that the representation is some times exponentially more succinct than using BDDs. For this approach, time rather than memory is the main bottleneck.

The approach was introduced by Biere et. al[2]. They coined the term *bounded model checking* for a method of unrolling symbolically the system k steps, and then checking if it is possible to find a counterexample to a given LTL formula that is k steps long.

6.1 Example

Consider a two-bit register. We represent the contents of the register in the state i with the propositional variables w_1^i and w_2^i. The register has the following transition relation $R(w^i, w^{i+1}) = (w_1^i \vee w_2^i) \leftrightarrow w_1^{i+1} \wedge (w_1^i \oplus w_2^i) \leftrightarrow w_2^{i+1}$. Initially both registers are set, which is given by the initial predicate $I(w) = w_1 \wedge w_2$. We wish to verify that eventually the contents of the register will be empty, i.e. $\mathbf{F}(\neg w_1 \wedge \neg w_2)$. The negation of this formula is $\mathbf{G}(w_1 \vee w_2)$.

To consider counterexamples of length $k = 2$, we give a propositional formula which symbolically describes all possible paths of length two

$$M = I(w^0) \wedge R(w^0, w^1) \wedge R(w^1, w^2).$$

We add the requirement that the path must either contain a loop or w^2 must be terminal, so that it describes maximal paths required by the LTL semantics.

$$P = T(w^2) \vee R(w^2, w^2) \vee R(w^2, w^1) \vee R(w^2, w^0)$$

Here $T(w)$ is the terminal predicate. Finally we must formulate the requirement given by the formula $\mathbf{G}(w_1 \vee w_2)$. This is equivalent to that the register is not zero in each state:

$$F = (w_1^0 \vee w_2^0) \wedge (w_1^1 \vee w_2^1) \wedge (w_1^2 \vee w_2^2)$$

By combining these constraints for the model, path and formula we get the formula $M \wedge P \wedge F$ which is satisfiable if and only if the formula $\mathbf{F}(\neg w_1 \wedge \neg w_2)$ has a counterexample of length two. In this case the formula is satisfiable.

6.2 Translation to Propositional Logic

The procedure above can be generalised and done for any finite state Kripke structure and LTL formula. Let $\mathcal{M}$ be a Kripke structure, $I(w)$ the initial predicate and $T(w)$ the terminal predicate. The following formula $[[\mathcal{M}]]$ describes the legal maximal paths $w^0 \dots w^k$ of length k.

$$[[\mathcal{M}]] = I(w^0) \wedge \bigwedge_{i=1}^{k} R(w^{i-1}, w^i) \wedge \left(T(w^k) \vee \bigvee_{l=0}^{k} R(w^k, w^l) \right)$$

The path represented by $w^0 \dots w^k$ can represent infinite behaviour if it contains a loop.

In the translation of an LTL formula ψ to propositional logic the translation proceeds recursively on the structure of the formula. The propositional operators are trivial to translate as they are also a part of propositional logic. The translation of $\mathbf{U}^+$ merits some discussion.

The until operator $\varphi \mathbf{U}^+ \psi$ requires that ψ is true in some future state and that φ holds up until that state. For a path $w^0 \dots w^k$ there are two ways in which the until formula can be satisfied in a state w^i.

- $w^j \models \psi$ for some $i < j \leq k$ and $w^m \models \varphi$ for all $i < m < j$. This holds also when w^k is terminal.
- The path has loop from w^k to some w^l, i.e. $R(w^k, w^l)$, and $w^j \models \psi$ for $l \leq\ j \leq i$ and $w^m \models \varphi$ for all $i < m \leq k$ and $l \leq m < j$.

The two different situations are depicted Figure 8.

Now we define $[[\psi]]^i_k$ recursively on the structure of ψ. In this recursion k is fixed while i depends on the evaluation point. Let $k, i \in \mathbb{N}$ and $\vee^k_{j=l} \psi = \bot$ for $l > i$.

- $[[p]]^i_k = p(w^i)$, i.e. the proposition for p in w^i
- $[[\bot]]^i_k = \bot$

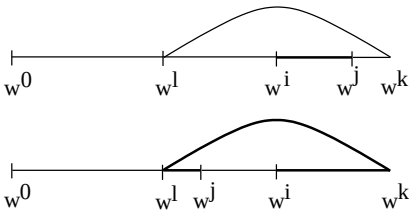

Figure 8: The two different ways of satisfying an 'until'-formula

- $[[(\varphi \rightarrow \psi)]]_k^i = ([[\varphi]]_k^i \rightarrow [[\psi]]_k^i)$
- $$[[(\varphi \mathbf{U}^+ \psi)]]_k^i = \bigvee_{j=i+1}^{k} \left([[\psi]]_k^j \wedge \bigwedge_{m=i+1}^{j-1} [[\varphi]]_k^m \right) \vee \bigvee_{l=0}^{k} \left(\bigwedge_{m=i+1}^{k} [[\varphi]]_k^m \wedge R(w^k, w^l) \wedge \bigvee_{j=l}^{i} \left([[\psi]]_k^j \wedge \bigwedge_{m=l}^{j-1} [[\varphi]]_k^m \right) \right)$$

The translation as it has been presented here is not very efficient. By introducing translations $\mathbf{G}^+, \mathbf{F}^+$, etc. it is possible to make a more efficient translation.

The presented results can summarised in the following theorem.

Theorem. There exists a maximal path of length k generated by $\mathcal{M}$ which initially validates ψ iff $([[\mathcal{M}]]_k \wedge [[\psi]]_k^0)$ is propositionally satisfiable.

Without knowing an upper bound for k, bounded model checking can only be used for falsification and not proving. For LTL, the upper bound for k is $|\mathcal{M}| \times 2^{|\psi|}$. It is likely that for many cases a better upper bound exists. However, finding a good upper bound for k is a hard problem.

7 Partial Order Methods

When a system consists of several parallel components, there usually is behaviour in the processes which is independent of the behaviour of the other processes. The interleaving semantics for concurrency does not, however, take this into account when the global behaviour is generated. The global behavior contains all interleavings Partial order methods formalise this notion of independence and utilise it for generating smaller state spaces. Only

the part of the state space which is needed to evaluate a given formula is generated.

Partial order methods are usually applied in the context of explicit model checking, although partial order methods for symbolic model checking also exist.

7.1 Elementary Petri Nets

Partial order methods work on the level of the system description. Here we use elementary Petri nets are our formalism.

Definition. An elementary Petri net is a tuple $N = \langle P, T, F, s_0 \rangle$, where

- P is a finite set of places,
- T is a finite set transitions s.t. $P \cap T = \emptyset$,
- $F \subseteq (P \times T) \cup (T \times P)$ is the flow relation, and
- $s_0 \subseteq P$ is the initial marking of the net.

By $\bullet x = \{y |\ (y, x) \in F\}$ we denote the preset of a place or a transition and $x\bullet = \{y |\ (x, y) \in F\}$ denotes the postset. A *marking* is a subset of P. A transition $t \in T$ is *enabled* if $\bullet t \subseteq m$ and $(t \bullet \cap m) \subseteq \bullet t$. If a transition t is enabled in a marking m it can be *fired* resulting in the marking $m' = (m \setminus \bullet t) \cup t\bullet$. The size of the state space bounded by $2^{|P|}$

7.2 Stuttering Invariance

Intuitively, most the reductions will result from ignoring different interleavings of independent events. Stuttering invariance formalises the equivalence between these different interleavings.

Given a set of atomic propositions $\{p_1, \ldots, p_n\} \subseteq \mathcal{P}$, two natural models $\mathcal{M}$ and $\mathcal{M}'$ are *strongly equivalent* w.r.t. $\{p_1, \ldots, p_n\}$, if they are of the same cardinality and for all $i \geq 0$ and all $p \in \{p_1, \ldots, p_n\}$ we have $w_i \in \mathcal{I}(p)$ iff $w'_i \in \mathcal{I}(p)$. A point w_{i+1} in $\mathcal{M}$ is *stuttering* w.r.t. $\{p_1, \ldots, p_n\}$, if for all $p \in \{p_1, \ldots, p_n\}$ $w_i \in \mathcal{I}(p)$ iff $w_{i+1} \in \mathcal{I}(p)$. In other words, the points w_i and w_{i+1} have the same valuation w.r.t $\{p_1, \ldots, p_n\}$. The *stutter-free kernel*

$\mathcal{M}^0$ of a model $\mathcal{M}$ is obtained by retaining all non-stuttering states of $\mathcal{M}$. Two states w, w' in $\mathcal{M}^0$, are consecutive $w \prec w'$ iff $w \prec w'$ in $\mathcal{M}$ or there are stuttering points $w_1, \ldots, w_k$ such that $w \prec w_1 \prec \cdots \prec w_k \prec w'$ in $\mathcal{M}$.

Two models $\mathcal{M}$ and $\mathcal{M}'$ are *stuttering equivalent* w.r.t. $\{p_1, \ldots, p_n\}$, if their stutter-free kernels are strongly equivalent. A formula φ is *stuttering invariant* if for all stuttering equivalent models $\mathcal{M}, \mathcal{M}'$, $\mathcal{M} \models \varphi$ăiff $\mathcal{M}' \models \varphi$.

To allow the partial order methods to perform reductions we must restrict the logic we are using, so that it cannot express properties which interfere with the reduction. For our reductions we need stuttering invariant formulas. How do we know if an **LTL** formula is stuttering invariant. Peled has presented a procedure for determining if an **LTL** formula is stuttering invariant. Another possibility is syntactically restrict **LTL**. Formulas containing the operator **X** are not in general stuttering equivalent. The simplest example is to consider the formula $\mathbf{X}p$ and the model $(\{w_0, w_1, w_2\}, \mathcal{I}, w_0)$, where $w_0 \prec w_1 \prec w_2$ and $\mathcal{I}(p) = \{w_0, w_1\}$. In this model $\mathbf{X}p$ holds, but not in the stuttering equivalent model $(\{w_0^0, w_1^0\}, \mathcal{I}^0, w_0^0)$. There are, however, formulas containing the operator **X** which are stuttering invariant.

Define **LTL** $-$ **X** to be the logic built from propositions $p \in \mathcal{P}$, boolean connectives $\rightarrow, \bot$ and the reflexive until $\mathbf{U}^*$ temporal operator.

Lemma. Any **LTL** $-$ **X** formula is stuttering invariant.

Proof: Let ψ be an **LTL** $-$ **X** formula, $\mathcal{M} = (U, \mathcal{I}, w_0)$ a model and $\mathcal{M}^0 = (U^0, \mathcal{I}^0, w_0^0)$ its stuttering free kernel w.r.t the atomic propositions in ψ. We show that

$$\mathcal{M} \models \psi \text{ iff } \mathcal{M}^0 \models \psi.$$

If this can be shown the theorem follows immediately.

The proof proceeds by induction on the structure of the formula. If ψ is a proposition then $w_0 \models \psi$ iff $w_0^0 \models \psi$, because $w_0 \in \mathcal{I}(p)$ iff $w_0^0 \in \mathcal{I}^0(p)$ by stuttering invariance. If the main operator of ψ is a boolean connective the equivalence follows trivially from the induction hypothesis. Finally the case where $\psi = \varphi_1 \mathbf{U}^* \varphi_2$. If $\mathcal{M} \models \psi$ it means that there exists a $w_1 \geq w_0$ such that $w_1 \models \varphi_2$ and $w_i \models \varphi_1$ for all $w_0 \leq w_i < w_1$. By the induction hypothesis there exists $w_1^0 \models \varphi_2$. There are now two cases: $w_1 = w_0$ and $w_1 \neq w_0$. If $w_1 = w_0$ we are done, because then we can infer directly from $w_1^0 \models \varphi_2$ that $\mathcal{M}^0 \models \psi$. If $w_1 \neq w_0$, we have by the induction hypothesis that there exists stuttering equivalent states for which $w_i^0 \models \varphi_1$ As stuttering equivalence respects the transition relation of the original model we know that

$w_0^0 \leq w_i^0 < w_1^0$. Thus, $\mathcal{M}^0 \models \psi$.

The converse can also be proven to hold.

Theorem. Any **LTL** formula which is stuttering invariant is expressible in **LTL** $-$ **X**.

Armed with these two theorems we can try to alleviate the state space explosion problem by generating smaller stuttering equivalent models instead of complete state spaces when model checking **LTL** $-$ **X** formulas.

7.3 A Partial Order Method for Elementary Nets

Next we investigate how to proceed, if we are given an elementary net N and a **LTL** $-$ **X** formula ψ. We are concerned with the question of efficiently generating a minimal stuttering equivalent state space w.r.t the atomic propositions in ψ. As we are working with elementary nets, the places of the net function as natural atomic propositions.

The basic idea is that when generating the state space we aim at firing only a subset of the enabled transitions, thus reducing the branching in the state space and possibly generating fewer states. A notion of *independence* between transitions is used as a basis for the reduction. Two transitions are considered independent if they do not enable nor disable each other and they *commute*, i.e. each firing sequence obtained by first firing the one of the transitions and then the other must be stuttering equivalent to any sequence where the transition are fired in reverse order.

This notion of independence is dynamic in nature and is too hard to check. A syntactic characterisation is needed. Given a net N and a marking m, a set of transitions $T_m \subseteq T$ is called *persistent* in m iff for all $t \in T_m$ and all firing sequences $t_0, t_1, \ldots, t_n, t$ such that $t_i \notin T_m$, there exists a stuttering equivalent firing sequence starting with t. Thus, by firing only transitions in T_m all possible sequences will have their stuttering equivalent representatives generated. For instance, the set of all enabled transitions is always a valid persistent set.

The method described below works by over approximating persistent sets at each marking and in this way generate reduced state spaces. One transition is selected and all interfering transitions are added, until a valid persistent set is reached.

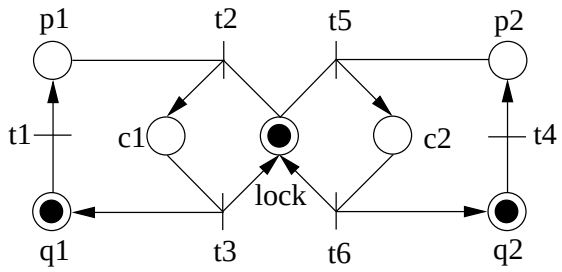

Figure 9: Elementary net for a simple mutex protocol

The net in Figure 9 will be used as a running example in the following. It is a simple mutual exclusion algorithm where two processes try to gain access to the critical section. A lock place guarantees that only one process has access at a time. We try to verify that $\psi = \mathbf{G}\neg(c_1 \wedge c_2)$.

We first consider the transitions which are necessary for a transition t, i.e. the transition which must occur before t can occur. For a transition to occur all its preplaces must be occupied. Thus any transition which puts tokens into a preplace of t is necessary. We say that a set $NEC(t,m)$ is *necessary* for t in m, if $NEC(t,m) = \{\bullet p |\ p \in (\bullet t \setminus m)\}$. In Figure 9, when m is the marking in the figure, $NEC(t_2,m) = \{t_1\}$. Let $NEC^*(t,m)$ be the transitive closure of $NEC(t,m)$ under necessity. If t is disabled in m, t cannot fire before some transitions in $NEC^*(t,m)$ fire. In the example $NEC^*(t_2,m) = \{t_1,t_2,t_3\}$.

If a transition can disable another transition we say that it is in conflict with the transition. The *conflict* of tăis defined by

$$C(t) = \{t' |\ \bullet t \cap \bullet t' \neq \emptyset\} \cup \{t\}.$$

In Figure 9, $C(t_2) = \{t_5\}$ while $C(t_1) = \emptyset$. For the propositions appearing in ψ, the order of the events may be relevant. Thus, transitions affecting these propositions must be inspected. A transition is called *visible* for ψ if $\bullet t \cup t\bullet$ appears in ψ In our example t_1, t_3, t_5, t_6 are visible. The *extended conflict* of t is $C(t)$ if t is not visible and $C(t)$ and all visible transitions otherwise.

For $t' \in C(t)$, any transition in $NEC(t',m)$ can enable t'. Thus, when we compute the dependent set of t it includes all transitions which are necessary for the transitions in conflict with t. $DEP(t,m)$ is any set such that for any t' in the extended conflict of t $NEC(t',m) \subseteq DEP(t,m)$. Our approximation must be closed under dependency. Therefore $READY(m)$ is any set fort which

$$DEP(t_f,m) \subseteq READY(m), \text{ if } t_f \in READY(m).$$

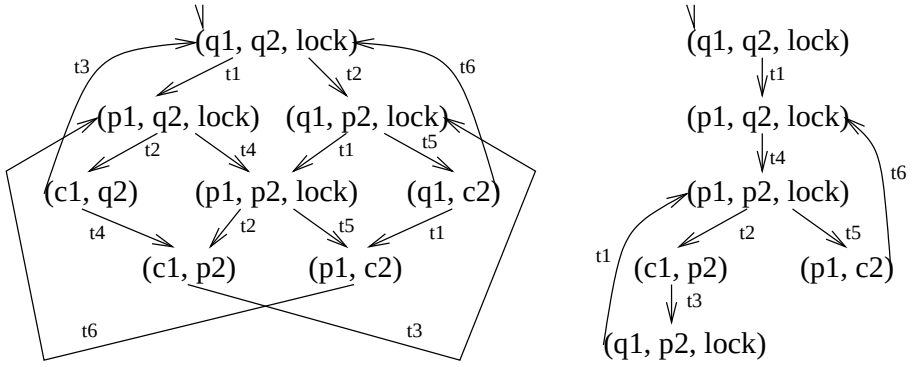

Figure 10: The complete state space (left) and the reduced state space (right).

The result is summarised in the following theorem.

Theorem. For any firing sequence ρ of the net there exists a firing sequence ρ' generated only by firing the enabled ready transitions such that ρ and ρ' are equivalent w.r.t. all **LTL-X** safety properties.

Consider again the net in Figure 9. In the marking shown in the figure, t_1 and t_4 are enabled. Both transitions have only themselves in their respective extended conflicts. Any of the transitions will be a valid ready set, because $NEC(t_1, m) = NEC(t_4, m) = \emptyset$. Figure 10 shows the reduced and the complete state space for the net.

A state space generated only by firing ready transitions can be exponentially smaller than the full state space. The complexity of the algorithm is cubic in the size of the net, but average examples can be completed in linear time. Extending the algorithm to also handle liveness properties correctly could be done by having the algorithm compute a different set, each time a state reached for the second time. Other logics such as $\mathbf{CTL}^* - \mathbf{X}$ and $\mathbf{ACTL} - \mathbf{X}$ also have their own versions of the algorithm.

References

[1] H.R. Andersen. *An Introduction to Binary Decision Diagrams*, available at `http://www.itu.dk/people/hra/bdd97-abstract.html`, 1998.

[2] A. Biere, A. Cimatti, E. Clarke and Y. Zhu. *Symbolic Model Checking without BDDs*. In W.R. Cleaveland, editor, Tools and Algorithms for the Construction and Analysis of Systems, pp 193–206, Springer, 1999.

[3] R.E. Bryant. Symbolic Manipulation with Ordered Binary Decision Diagrams, *ACM Computing Surveys* 24(3), 293–317, 1992.

[4] E.M. Clarke and B-H. Schlingloff. *Model Checking*. In A. Robinson and A. Voronkov, editors, Handbook of Automated Reasoning, pp 1637–1790, Elsevier, 2001