Budapest University of Technology and Economics
Faculty of Electrical Engineering and Informatics
Department of Measurement and Information Systems

Symbolic model checking and trace generation by guided search

TDK-thesis

Authors:

Dániel Élő
Adrián Soltész

Supervisors:

Vince Molnár
András Vörös

2015.

Contents

Chapter 1

Introduction

Model checking is a formal verification technique for verifying requirements on behavioral models of systems in a mathematically precise way. An advantage of model checking is that in case the requirements are violated, it can produce an execution trace (counterexample) to demonstrate the incorrect behavior of the system. When analyzing state reachability, or safety properties, it is especially important to generate traces leading to the states in the scope of the analysis. This information can be used for the correction of the errors or it can be used for model based test generation.

Traditional model checking algorithms systematically and often exhaustively explore the state space of the system, i.e., they build the graph representation of the state space. These algorithms are called *explicit model checkers*, because they explicitly enumerate the states and state transitions of the system. While explicit approaches are mature, they are inherently limited by the size of the state space that can fit into memory. Unfortunately, even relatively small systems can have huge state spaces, a phenomenon that is commonly referred to as *state-space explosion*.

One way of handling the state space explosion problem is *symbolic model checking*, which exploits the inner structure of states to encode the state space as a decision diagram. Symbolic encoding significantly extends the size of manageable state spaces, but does not allow to exploit the advantages of explicit techniques – this is why many of the efficient symbolic algorithms are incapable to give counterexample or only with a huge computational overhead.

One reason of state space explosion in concurrent systems is the interleaving semantics of model checking, i.e., total ordering of independent, asynchronous operations. Examination of each interleaving leads to a large amount of intermediate states that might be irrelevant for the requirements. The main idea of *partial order reduction* is to substitute the equivalent interleavings with a single representative trace to reduce the number of states to discover. The technique builds on explicit state traversal, so most of the methods used there remain applicable, i.e., we can compute traces to erroneous states efficiently.

Applying *directed (guided) model checking* can further improve the efficiency of explicit state model checking by heuristically classifying states and transitions of models based on which could give results sooner or produce shorter traces. Every heuristic builds on additional known properties of system model – the efficiency of the approach greatly depends on choosing these properties well.

This work aims to contribute in three different, but related fields.

1. We propose a workflow to combine the advantages of symbolic and explicit model checking algorithms, focusing on safety checking and counterexample generation.
2. In the field of symbolic model checking, we investigate state-of-the-art symbolic representations to assess their strengths and weaknesses and lay the foundations of potential future improvements, both in theory and in terms of implementation.
3. We propose a directed model checking algorithm building on a new kind of heuristic, combined with partial order reduction. We also prove the soundness and rationale of the heuristic.

Chapter 2

Background

In this chapter we introduce some basic concepts and methods used in the literature and refer to related work which server as the basis of our approach. Firstly, section 2.1 presents Petri nets, a widespread modeling language used in the field of formal methods. Secondly, section 2.2 introduces the Kripke structures commonly used to describe state spaces in the literature. Thirdly, section 2.3 presents the basic concepts of model checking and safety properties which is the basis of our approach. Fourthly, section 2.4 introduces the basics of symbolic model checking, a cardinal part in our work. Finally section 2.5 presents partial order reduction, a method to combat the state space explosion in explicit model checking.

2.1 Petri Nets

Our work relies on Petri nets as a modeling formalism: in this section we give a brief introduction. For a full-detailed description of Petri nets, refer to [17].

Petri nets are a popular formalism which are especially suitable for modeling concurrent, asynchronous and nondeterministic systems. – this is why we chose it for the first formalism to apply our solutions on.

Definition 1 (Petri net).
A Petri net is a 5-tuple $PTN = (P, T, A, W, M_0)$ where:

- P is a finite set of *places*;
- T is a finite set of *transitions*, such that $P \cap T = \emptyset$;
- $A \subseteq (P \times T) \cup (T \times P)$ is the set of *arcs*;
- $W : (P \times T) \cup (T \times P) \to \mathbb{N}_0$ is a *weight function* such that $W(x, y) > 0$ *iff* $(x, y) \in A$;
- $M_0 : P \to \mathbb{N}_0$ is the initial marking, i.e., the number of *tokens* on each *place*. ▪

A Petri net consists of places, transitions and arcs. A state of the Petri net is determined by the *marking function* ($M : P \to \mathbb{N}_0$) registering the number of tokens for every place.

Throughout this work, we will assume that a Petri net is *bounded*, which means that the number of tokens in every place is below a certain value in every reachable state. Furthermore, we assume that there are no parallel arcs (arcs that has the same start and end point) in the net, if there would be an equivalent net could be produced by merging the parallel arcs. For the sake of convenience, we introduce some common notations used to refer to the structure of the Petri nets.

- $\bullet t = \{p | p \in P \wedge (p,t) \in A\}$, i.e., the set of input places of t;
- $t\bullet = \{p | p \in P \wedge (t,p) \in A\}$, i.e., the set of output places of t;
- $\bullet p = \{t | t \in T \wedge (t,p) \in A\}$, i.e., the set of input transitions of p;
- $p\bullet = \{t | t \in T \wedge (p,t) \in A\}$, i.e., the set of output transitions of p;
- $W^+(t,p) = W(t,p)$ is the total amount of tokens transition t adds to place p when fired.
- $W^-(t,p) = W(p,t)$ is the total amount of tokens transition t removes from place p when fired.
- $W^* = W^+ - W^-$ represents the sum of removed and produced number of tokens in place p.

W^* can be used to calculate whether transition t increases or decreases the number of tokens on a place.

The behavior of a Petri net is defined by the following *firing rules*:

- A transition t is *enabled iff* $\forall p \in \bullet t : M(p) \geq W(p,t)$, i.e., all input places of t has at least as many tokens as the weight of the input arcs of t.
- An enabled transition may *fire* and change the marking of the Petri net. Every enabled transition can fire, but there is no ordering or precedence amongst them inducing a non-deterministic behavior.
- When a transition t fires, it removes $W(p,t)$ tokens from all of its input places $p \in \bullet t$ then puts $W(t,p)$ tokens to its output places $p \in t\bullet$.

An example of the firing mechanism in Petri nets is shown on figure 2.1.

We call τ a *firing sequence iff* it is a sequence of transitions that can be fired in the Petri net exactly in that order starting from the current state (commonly the initial state) of the Petri net. The sequence of states reached after each step in τ (including the initial state) is called a *path* and is denoted by ρ. We call a marking M reachable in the Petri net *iff* there is a path ρ that ends in M.

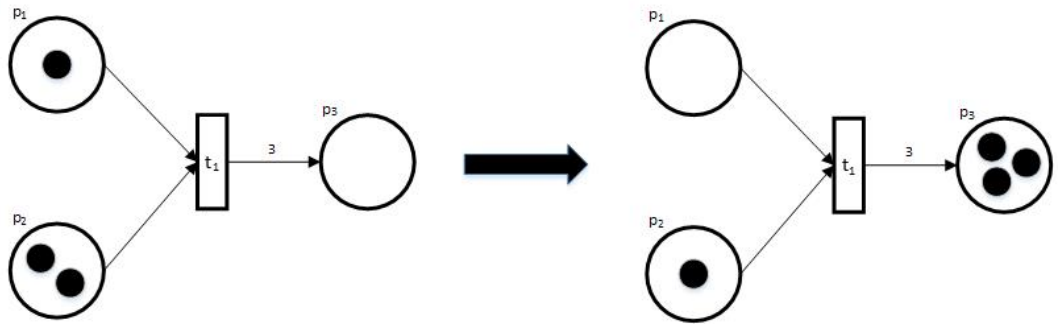

Figure 2.1. Firing, in a graphical representation of Petri net

2.2 Kripke Structures

The state space of high-level models can be described by so-called Kripke structures ([14]). Kripke structures are directed graphs, where the nodes are labeled. The states of the model are represented by the labeled nodes of the Kripke structure while transitions correspond to the arcs connecting the nodes. Labels provide a way to reason about states.

Definition 2 (Kripke structure). Given a set of atomic propositions AP a Kripke structure is a 4-tuple $K = (S, I, R, L)$, where:

- S is the finite set of states;
- $I \in S$ is the set of initial states;
- $R \in S \times S$ is the transition relation;
- $L : S \longrightarrow 2^{AP}$ is the labeling function that maps a state to a subset of atomic propositions; ▪

In a Kripke structure a path (trace) ρ is a sequence of states, corresponding to a directed path in the graph, i.e., states in ρ follow each othr according to R.

2.2.1 Connection Between Petri Nets and Kripke Structures

As stated in section 2.2, Kripke structures can be used to represent the state-space of high level models.

In case of Petri nets a state of the Kripke structure describes the marking of a Petri net. The atomic propositions constituting the labels on the Kripke structure are relations interpreted over the marking of the Petri net. The initial state of the Kripke structure corresponds to the initial marking of the Petri net. Transitions of the Kripke structure can be regarded as instances of the transitions of the Petri net – they correspond to a single firing of an enabled transition. This way, the state-space of the Petri net is described by a (not necessarily connected) Kripke structure. Exploring the state space off the Petri net essentially means the traversal of the Kripke structure.

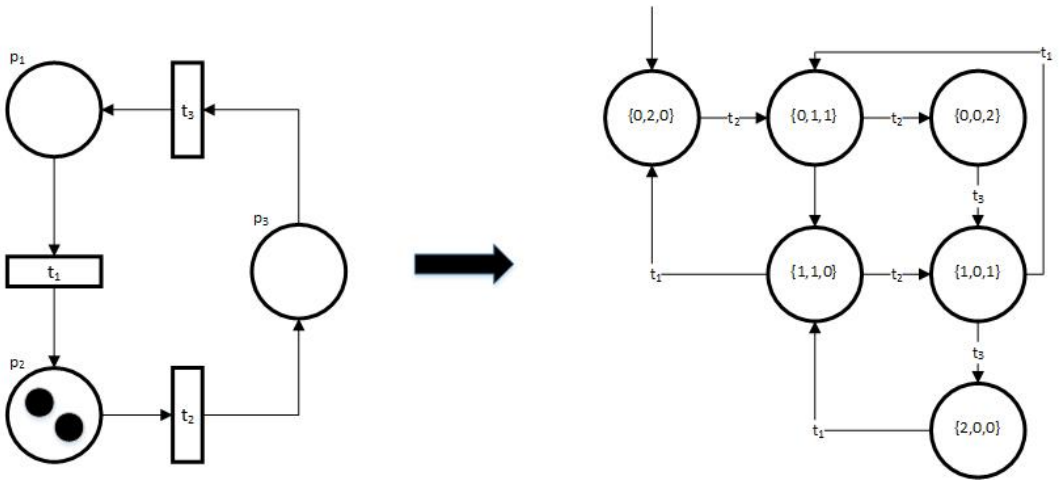

Figure 2.2. Kripke structure describing state space of Petri net

It is important to note, that (as shown in figure 2.2) the state space of a Petri net model can be much larger and much more complex than the structure itself, that is why discovering the full state space can be hard. In sections 2.4, 2.5 we will show methods to handle this problem.

2.3 Model Checking

Model checking [4] is an automatic formal verification technique for exhaustively computing and analyzing the state space to see if it satisfies a given requirement.

The input of the model checking procedure is the model of the system (in our work a Petri net model) and some formal specifications (this work considers safety properties). The states or traces of the model are examined and if the model violates the requirements it a counterexample is given to demonstrate the error. (Commonly a trace to an erroneous state which violated said specifications).

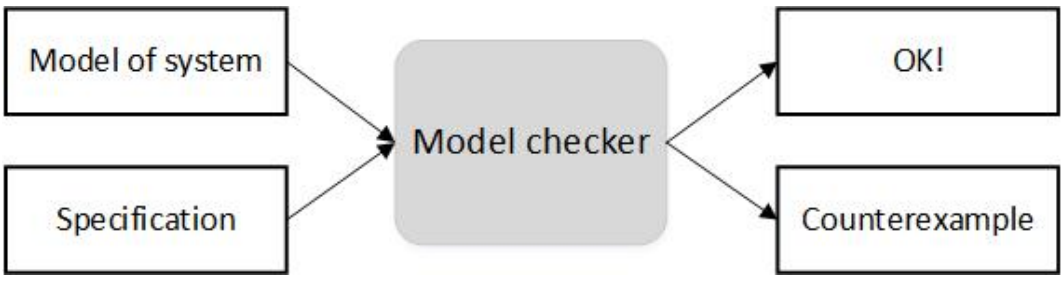

Figure 2.3. The general workflow of model checking

Due to the fact that model checking computes the state-space of high-level models, the aforementioned state space explosion arises. Sections 2.4, 2.5 will introduce two ways of dealing with state-space explosion.

One of the main contributions of this work is the combination of these techniques to be efficient even for large state-spaces preserving the ability to generate counterexamples.

Similar works are present in the literature [1][10] but we use a significantly different approach.

2.3.1 Reachability

A common use of model checking is checking safety properties, i.e., that some unwanted event or situation cannot occur in the system. This problem can be reduced to reachability of these states.

"Bad" states (or goal states in general) are usually described by state predicates referring to state variables of the system (e.g., a given place in a Petri net must not have more than 2 tokens). Checking reachability is then performed by evaluating the predicates on reachable states to see ifa described state can be found.

2.3.1.1 State Properties

In this section we will introduce state predicates to describe a set of states. A state is in the described set if it satisfies the state predicates. We describe the state predicates

The atomic state propositions are based on the marking of a Petri net in the following form:

$\phi ::= p_i < k \mid p_i > k \mid p_i \leq k \mid p_i \geq k \mid p_i = k \mid p_i \neq k \mid \top \mid \bot$

This rule generates the set of atomic propositions AP=$\{\ell_1, \ell_2, ...\ell_n\}$ used as labels in the Kripke structures, where $p_i \in P$ and k is a constant integer.

A state predicate formula is described in terms of atomic propositions as follows:

$\Phi ::= \phi_i \mid \neg\Phi \mid \Phi \vee \Phi \mid \Phi \wedge \Phi$

to denote a concrete formula generated by the grammar we will use φ.

Example:

$\varphi \equiv p_1 \neq 1 \wedge (p_4 < 5 \vee p_{30} \geq 3)$ means that "on place p_1 cannot have exactly 1 token and either p_4 has less than 5 tokens or p_{30} has at least 3 tokens".

A number of functions and notations are introduced below to describe certain aspects of atomic propositions. Given a state predicate φ, an atomic proposition ℓ and a marking M.

- $AP(\varphi) = \{\ell_1 \dots \ell_n\}$ *denote the set of atomic propositions in the current formula* φ.
- $sub(\ell) \in P(\ell \in AP(\varphi) \longrightarrow p \in P)$ *is the subject of* ℓ*, i.e., the place* ℓ *refers to.*
- $k(\ell) : \ell \in AP(\varphi) \longrightarrow \mathbb{N}$ *the constant in* ℓ.
- $op(\ell) : \ell \in AP(\varphi) \longrightarrow \{=, \neq, <, \leq, >, \geq\}$ the operator of ℓ.

- $Less(\ell, M) = \begin{cases} true: & M(sub(\ell)) = k(\ell) \wedge op(\ell) \in \{\neq\} or \\ & M(sub(\ell)) < k(\ell) \wedge op(\ell) \in \{=, >, \geq\} \\ false: & otherwise. \end{cases}$
- $More(\ell, M) = \begin{cases} true: & M(sub(\ell)) = k(\ell) \wedge op(\ell) \in \{\neq\} or \\ & M(sub(\ell)) > k(\ell) \wedge op(\ell) \in \{=, <, \leq\} \\ false: & otherwise. \end{cases}$

$Less$ ($More$) means that in M there is currently less (more) token on $sub(\ell)$ than any state satisfying ℓ. These functions are used as guiding heuristics in our guided search algorithm presented in Chapter 5

2.3.2 Directed Model Checking

Directed model checking [10] is an efficient method to combat the state-space explosion problem in case a goal state is reachable. Explicit directed model checkers [8] replace the standard depth-first search strategy with heuristic search in order to guide the exploration of the state-space towards the goal states. If it is enough to find a single goal state, the algorithm can terminate as soon as the first solution is found. In this case, guiding the search towards a goal state can significantly reduce the number of states that has to be traversed before termination.

The heuristic search algorithm is usually A^*, while the most common heuristics are based on some notion of distance between markings (e.g., Hamming distance)[10]. It is important to note that such a heuristic is only useful when a goal state is actually reachable.

An important use case for directed model checking is in the field of model-based test generation for software and hardware. With this approach, it is possible to compute test cases to check if the implementation conforms to the specification model. This can be regarded as an automated way of specification-based testing, reducing the cost of test engineering and raising the quality of the product.

2.3.3 Symbolic Model Checking

Complex systems often have a huge state space with large state vectors, so efficient encoding of states are necessary. In order to tackle the state space explosion problem, symbolic algorithms introduce special encodings of the state space. These approaches handle huge sets of states together and encode them in a compact symbolic representation.

The general idea behind symbolic algorithms is to operate on large sets of states instead of single states [2]. As stated above, these methods encode state spaces in a compact form, directly manipulating the symbolic representation. Common representations include binary functions and decision diagrams, this work focuses on the latter approach.

2.4 Decision Diagrams

Decision diagrams are more compact forms of decision trees, where the nodes with the same meaning were contracted to achieve more efficient, less redundant storage without loss of information. In this section, we introduce two of the most widely used decision diagrams (*binary decision diagram and multivalued decision diagram*) and a specialized hierarchic decision diagram, called *set decision diagram.*

2.4.1 Binary Decision Diagrams

Binary decision diagrams (or *BDDs*) were introduced by Randal Bryant in 1986, to efficiently represent binary functions. [3] In the following we introduce BDDs in more detail as they are the most basic type of decision diagrams, encoding a set of binary vectors.

Definition 3 (Binary decision diagram).
A *binary decision diagram* is a directed acyclic graph, with a node set (V) consisting of two types of nodes: terminal and non-terminal nodes. Every nonterminal node $v \in V$ has two outgoing edges to two children nodes, we denote them as $v[0] = low(v) \in V$ and $v[1] = high(v) \in V$. Nodes are associated with levels: $level(v) \in \mathbb{Z}^+$. For every non-terminal node, $level(low(v)) < level(v)$ and $level(high(v)) < level(v)$ must hold. There are also exactly two terminal nodes, $\mathbf{0} \in V$ and $\mathbf{1} \in V$, called *terminal zero* and *terminal one* respectively. The terminal nodes also have fixed level numbers: $level(\mathbf{0}) = level(\mathbf{1}) = 0$. The terminal nodes encode binary values: $value(\mathbf{0}) = 0$, $value(\mathbf{1}) = 1$. Every BDD has a *root node* which is on the highest level (top level). ▪

The graphical representation of BDDs consists of circles for the non-temrinal nodes, squares for the terminal nodes and arrows for the directed edges. The squares are labeled with the value of the corresponding terminal node. There are two type of arrows: dotted arrows for the $low(v)$ edges and solid arrows for the $high(v)$ edges.

The semantics of BDDs come from the level numbers: each level corresponds to a variable. The value of the encoded function can be computed by traversing the graph starting from the root, and following the edges according to the value of the variable of the current level.

In Figure 2.4 presents a BDD encoding a binary functions that is true for the following Boolean tuples: *(0, 0, 0)*, *(0, 1, 0)*, *(0, 1, 1)*, *(1, 0, 1)*, *(1, 1, 1)*.

2.4.2 Multivalued Decision Diagrams

Multivalued decision diagrams (or *MDDs*) are extensions of binary decision diagrams. In BDDs, nodes have two outgoing edges: low(v) and high(v). An MDD encodes an integer function $f(x_1, ..., x_n) \longrightarrow \{0, 1\}$, so a node in an MDD corresponds to an variable x_i with a finite domain D_i. This way, a node of an MDD has $|D_i|$ outgoing edges. For the sake of simplicity, we use integer domains without loss of generality: $D_i = \{0, 1, 2, ..., |D_i| - 1\}$.

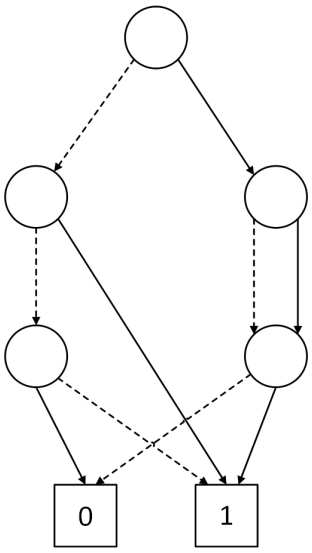

Figure 2.4. Graphical representation of a BDD

Just like BDDs, MDDs have two terminal nodes: *terminal zero* and *terminal one*. Non-terminal nodes, as previously stated, are different, with node v having $|D_i|$ outgoing edges pointing to nodes $v[0], v[1], ..., v[|D_{level(v)}| - 1]$.

When there are no isomorphic sub-diagram in an MDD, we call it a *canonical MDD*. Formally, if $v = w \in V, level(v) = level(w)$ and $\forall i \in D_{level(v)} : v[i] = w[i]$ in a canonical MDD, then $v = w$.

If a canonical MDD has no edges skipping levels, we call it a *quasi-reduced MDD*. Formally, this means that $\forall i \in D_{level(v)} : level(v[i]) = level(v) - 1$ holds for every non-terminal node.

When representing the MDD graphically, we still use circles and squares as in BDDs, but the edges in MDDs are always solid lines with an i integer label, which corresponds to the edge leading to $v[i]$. If an edge is not shown on a figure, then it either points to the terminal zero or to a *zero node*, which is a node where every outgoing *path* leads to the terminal zero.

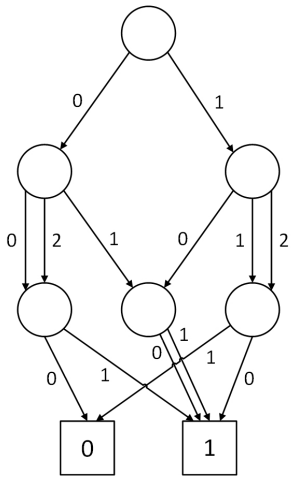

Figure 2.5. Graphical representation of a quasi-reduced MDD

Figure 2.5, we can see a MDD encoding a function which is true for the following *(x_1, x_2, x_3)* values: *(0, 0, 1)*, *(0, 1, 0)*, *(0, 1, 1)*, *(0, 2, 1)*, *(1, 0, 0)*, *(1, 0, 1)*, *(1, 1, 0)*, *(1, 2, 0)*.

2.4.2.1 Operations on Multivalued Decision Diagrams

MDDs are often used to encode a set of integer vectors as the function corresponding to the MDD returns **1** for exactly the vectors included in the set. Based on this, it is possible to define set operations on MDD nodes. The result of MDD operations on two MDD nodes encodes the same set as the corresponding set operations would produce from encoded sets of the original nodes. Operations are defined strictly to nodes on the same level.

The *union* of nodes v and w is

$$v \cup w = \begin{cases} v = 1 \vee w = 1 & \text{if } level(v) = level(w) = 0 \\ x & \text{otherwise, where } \forall i \in D_{level(v)} : x[i] = v[i] \cup w[i]. \end{cases}$$

The *intersection* of nodes v and w:

$$v \cap w = \begin{cases} v = 1 \wedge w = 1 & \text{if } level(v) = level(w) = 0 \\ x & \text{otherwise, where } \forall i \in D_{level(v)} : x[i] = v[i] \cap w[i] \forall i \in D_{level(v)}. \end{cases}$$

The *relative complement* of node w in v:

$$v \setminus w = \begin{cases} v = 1 \wedge w = 0 & \text{if } level(v) = level(w) = 0 \\ x & \text{otherwise, where } \forall i \in D_{level(v)} : x[i] = v[i] \setminus w[i]. \end{cases}$$

The union, the intersection and the relative complement of terminal nodes w and v is similar to Boolean logic.

Due to the recursive definition of the operations, they can be efficiently realized with recursive functions. Using a cache also improves the performance, since the same nodes can be reached along multiple paths.

2.4.3 Set Decision Diagrams

Set decision diagrams (SDDs) were introduced by [6], aiming to represent hierarchy in the data structure. The main idea behind SDDs is that its edges encode sets of values instead of a single value. To achieve this, the outgoing edges of a node are labeled by another decision diagram node instead of an integer. The definition in [6] builds on a special decision diagram type called data decision diagram (DDD). However, the ideas of SDDs follow more naturally if they are introduced as hierarchic extensions of MDDs.

Definition 4 (Set decision diagram).
A *set decision diagram* is a set of integer tuples represented by a directed acyclic graph, with a node set (V) consisting of two types of nodes: terminal and non-terminal nodes. Every nonterminal node $v \in V$ has at least one outgoing edge to a child node. Nodes are associated with levels: $level(v) \in \mathbb{Z}^+$. For every non-terminal node, $level(v[i]) < level(v)$ for every children of v. There are also exactly two terminal nodes, $\mathbf{0} \in V$ and $\mathbf{1} \in V$, called *terminal zero* and *terminal one* respectively. The terminal nodes also have fixed level numbers: $level(\mathbf{0}) = level(\mathbf{1}) = 0$. The terminal nodes encode binary values: $value(\mathbf{0}) = 0$, $value(\mathbf{1}) = 1$. The edges of the SDD encode sets of integer tuples. Edges are denoted by $x \xrightarrow{a_i} y$, where $x, y \in V$, and a_i is the root node of an MDD or SDD representation of the set of integers. ▪

A path from the root node of an SDD to the terminal one encodes the Cartesian product of the sets encoded by the labels of traversed edges. For the sake of convenience, the notion of a node is often used to refer to the set it encodes, if not ambiguous.

In order to efficiently use SDDs they have to be unambiguous.

Definition 5 (Canonical set decision diagram).
An SDD is *canonical iff*:

- $\forall v \xrightarrow{a_i} w \Longrightarrow a_i \neq \emptyset \wedge w \neq 0$;
- $\forall v \xrightarrow{a_i} w \wedge v \xrightarrow{a_j} z \Longrightarrow a_i \cap a_j = \emptyset \wedge w \neq z$. ▪

Definition 5 can be fulfilled by applying the following reduction rules. For a visual explanation, see Figure 2.6:

- A canonical representation of $v \xrightarrow{a_i} w$ and $v \xrightarrow{a_j} w$ is $v \xrightarrow{a_i \cup a_j} w$.
- A canonical representation of $v \xrightarrow{a_i} w$ and $v \xrightarrow{a_j} z$, where $w \cup z \neq \emptyset$ and $w \cap z \neq \emptyset$, is $v \xrightarrow{a_i \setminus a_j} w$, $v \xrightarrow{a_j \setminus a_i} z$ and $v \xrightarrow{a_i \cap a_j} w \cup z$.

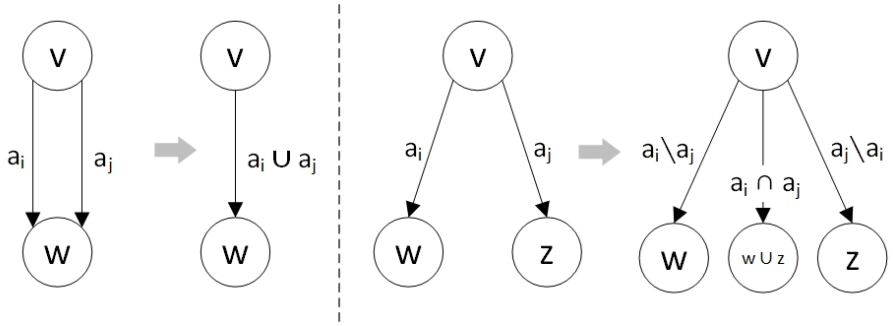

Figure 2.6. SDD reduction rules visualized

The advantage of MDDs is that they can exploit the similarities of the encoded tuples to achieve a compact representation. In addition to this, SDDs add the capability of

exploiting the inner structure of the encoded tuples, i.e., symmetries inside a tuple can be efficiently represented hierarchically. When used in symbolic model checking, this feature aligns with the compositional structure of high level models.

The graphical representation of an SDD is problematic due to the hierarchic structure. The convention used in this work is to represent labeled nodes by a dashed arrow pointing from the referencing edge to the referenced decision diagram's node. Figure 2.7 shows the graphical notations and also illustrates the source of compactness in SDDs.

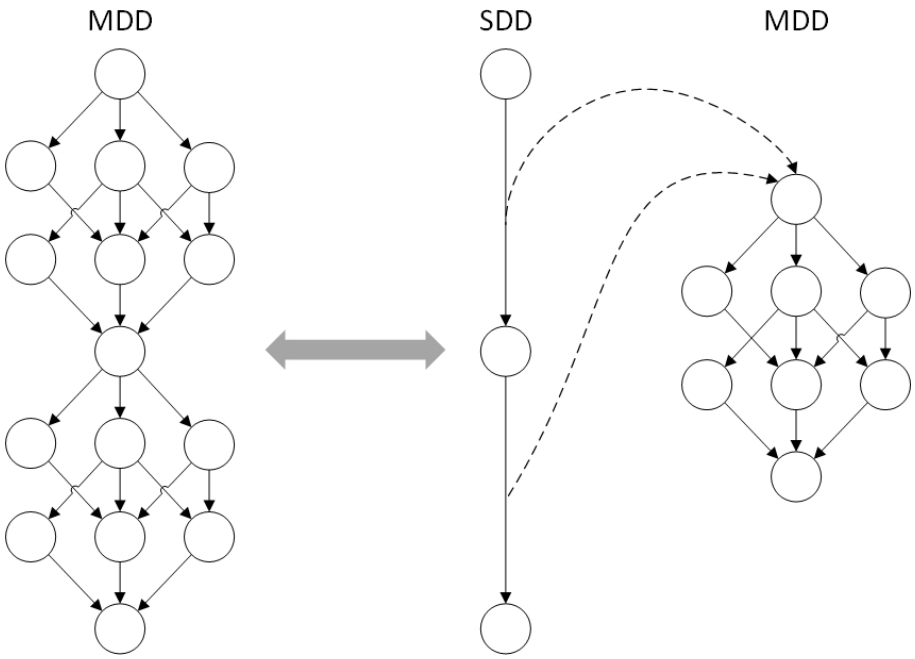

Figure 2.7. An MDD and an SDD hierarchy encoding the same set

2.5 Partial Order Reduction

In this section we show another way of handling the state space explosion problem.

For an illustration, see figure 2.8. In order to find a trace from P to Q, we must use transitions a, b, c, but the order of these transitions does not affect the reachability of Q. Partial order reduction would choose a representative ordering (e.g. a, c, b) omitting 4 intermediate states from the state space without affecting the reachability.

Partial order reduction uses that commonly the reason behind state-space explosion in concurrent systems is the interleaving semantics, i.e., the total ordering of actions in independent asynchronous processes. This interleaving leads to a huge number of possible behaviors and intermediate states that can be irrelevant with regard to our requirements. Partial order reduction chooses some representative orderings from these traces and discovers only them, resulting in a reduced state-space.

Partial order reduction[11][12][4] is an explicit technique, so it stores the states of the system explicitly. However the reduced state-space often allows to handle huge concurrent models. An important requirement for the reduction to preserve interesting properties. In this case of reachability properties it means that if a *goal state* [1] is reachable in the full state space it must be also reachable in the reduced state space. It is important to note that the reduction has to be performed without exploring the full state-space, because the main reason of the reduction is to avoid storing all the states in the memory. Most of the reduction approaches use some structural property of the model, because they can be checked statically without computing the full state space.

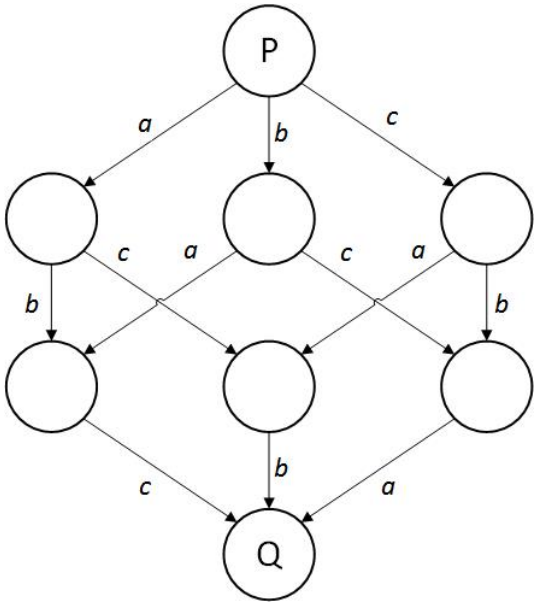

Figure 2.8. An example of indepently reorderable transitions

The idea of partial order reduction has been implemented in a number of approaches [9][16][1][11][15][12]. In this work, we use the *stubborn sets* method, which can be considered as the state-of-the-art algorithm in this field.

2.5.1 Stubborn Sets

The name of the stubborn sets method comes from the strategy of calculating sets of transitions whose firing can not disable transitions outside the stubborn set, and transitions outside the stubborn sets cannot disable transitions from the stubborn set either. These sets are stubborn in the sense that they stay enabled as long as only transitions of other sets are fired. Note that these stubborn sets are correspond to the independent concurrent processes, which we said partial order reduction uses to reduce the state-space.

The rationale behind stubborn sets is that in a given state it is enough to exhaustively fire a single set, because every other set will stay enabled and can be processed in a later state.

[1] Note that states that are not characterized by the predicate can become unreachable

For convenience reasons, we define the following notations:

- $s \xrightarrow{t_1 \ldots t_n} s'$ means that if we fire transition sequence $t_1 \ldots t_n$ from s we reach s'.
- $s \xrightarrow{t_1 \ldots t_n}$ means that the transition sequence $t_1 \ldots t_n$ is enabled from s.

Definition 6 (Stubborn sets). $STUB(s) \subseteq T$ is a stubborn set in state s *iff* the following conditions hold [16]:

D0: $STUB(s)$ is empty *iff* s is a deadlock state.

D1: $\forall t \in T$ and $\forall t_1 ... t_n$ sequence such that each $t_i \in STUB(s)$ and for which $s \xrightarrow{t, t_1 \ldots t_n} s'$ it holds that $s \xrightarrow{t_1 \ldots t_n, t} s'$, i.e., if a s' state is reachable from s via firing any transition from the net, and then all the $STUB(s)$ members, stays reachable if we fire the members of $STUB(s)$ first and then t.

D2: $\forall t \in STUB(s)$ and $\forall t_1 ... t_n$ sequence such that each $t_i \notin STUB(s)$ and for which $s \xrightarrow{t_1 \ldots t_n}$ if $s \xrightarrow{t}$ then $s \xrightarrow{t_1 \ldots t_n, t}$, i.e., if a transitions from $STUB(s)$ is enabled from state s no transition sequence outside the $STUB(s)$ can disable it. ▪

The condition *D0* ensures that deadlocks are preserved, i.e., we find a deadlock state in the reduced state space *iff* that state is a deadlock in the full state space. Conditions *D1*, *D2* ensures the stubborn behavior of the $STUB(s)$ members.

There can be many stubborn sets for a given state of a model, the key for the reduction is to choose which sets to fire before the others. An ideal solution is to use the one which takes the exploration closer to its goal. A more general direction is to choose the smallest set in order to produce a smaller reduced state-space. Note, however that the basic stubborn set methods only preserves deadlocks, so there is no guarantee that a set contains the transitions to reach a a desired part of the state-space. In order to preserve additional properties other constraints have to be fulfilled by the chosen sets. Since the union of stubborn sets is also a stubborn set by definition, so it is possible to construct a stubborn set that contains all the "essential" transitions. Chapter 5 will present a way of acquiring such transitions.

Note that *D1* and *D2* are very hard to check based on the definition, because it would require exploring the intermediate states, the exact thing tat the approach aims to avoid. Most of the implementations use constructions that inherently satisfy *D1* and *D2*, but are easier to compute statically. One such way is to compute the independent transitions based on the structure of the high-level model (see more in Section 5.2), which is a cheap over-approximation.

Chapter 3

Overview of the Approach

This work focuses on properties that can be reduced to reachability checking (e.g., safety properties). The presented algorithms are capable of determining if a given set of states is reachable from the initial state. In case of safety properties, a reachable unsafe state implies that the original safety property is false, and a counter-example (trace) is generated to demonstrate how to realize the violating behavior. The absence of reachable bad states indicate that our model is correct with regard to the property. In this section we introduce a hybrid model checking approach combining a symbolic and an explicit model checking algorithm (presented in chapters 4 and 5) to implement a scalable solution for reachability analysis and efficient trace generation.

3.1 A Hybrid Model Checking Procedure

In this section we describe our approach in terms of the general model checking workflow.

Symbolic and explicit algorithms have different trade-offs that often complement each other. For example, symbolic model checking algorithms are efficient in handling huge state spaces, however their decision diagram based variants are not efficient in trace generation. Explicit model checking algorithms can easily produce traces if the state space fits into memory. However, state space explosion often prevents their application. It is therefore desirable to find a solution that keeps the good from both worlds. Our proposed solution combines the advantages of both approaches: a symbolic state space traversal algorithm is used to check the properties and an explicit search algorithm is applied for trace generation, if needed.

To exploit the strengths of the two different approaches, both algorithms restricted to perform only those tasks in which they are better in. The symbolic algorithm can easily decide if a requirement holds, but additional operations are costly. Explicit algorithms can be specialized in computing traces, but then they easily fail if a goal state is not reachable. One of the main ideas of this work is to minimize the load of each algorithm by

dividing the tasks and providing information that is easy to compute. A similar approach is presented in [9].

The two tools work together, but separately. First a symbolic model checking is performed, then – if there are reachable goal states – the explicit algorithm is run with additional parameters computed in the first step, such as the exact specification of the reached goal state and its depth to it to aid trace generation. Figure 3.1 summarizes this strategy.

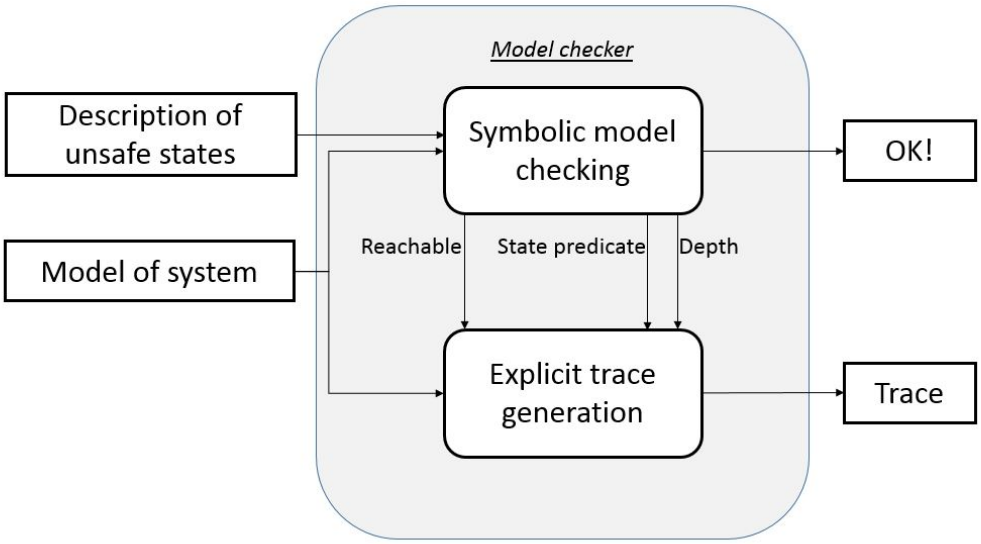

Figure 3.1. The general workflow in our work

3.2 Contributions in Symbolic Model Checking

Chapter 4 aims to investigate the possibilities of set decision diagrams in symbolic model checking. Since [6] and [13] use SDDs in a special context, a number of operations have to be defined to use this type of diagram in traditional symbolic algorithms. A special representation of transitions is also necessary in order to align with the hierarchical structure of SDDs.

In this work, we define the basic set operations for hierarchical decision diagram nodes to employ SDDs in two basic state space traversal algorithms. Our main goal is to evaluate the potential of SDDs compared to MDDs in these settings, so an implementation of a prototype is also presented. In Chapter 6, conclusions are derived from empirical comparisons of the different implementations, demonstrating the strengths and weaknesses of both representations and highlighting interesting directions for future work.

3.3 Contributions in Explicit Trace Generation

Chapter 5 presents a new approach of explicit trace generation. In the field of directed model checking it is common to aid the directed trace generation with a symbolic model

checking beforehand[10], but the majority of algorithms extracts exact markings from the symbolic search and bases its heuristic search on distances between markings by token count differences[10][8]. Our contribution to the field of directed model checking is that we calculate our reachability heuristic based on the behavior of transitions in the Petri net. This way we can produce very short traces for huge models efficiently if a goal state is reachable, but the method needs the aid of the symbolic model checker for the above discussed reasons.

Chapter 4

Symbolic Method

In this chapter, we present different approaches for symbolic state space representation and generation. Symbolic algorithms are essential when explicit algorithms cannot deal with the size of the state space anymore, especially when the whole state space has to be generated, such as the case when proving reachability. While these methods have the ability to decide if a given state is reachable, it is not trivial – and usually inefficient – to generate a demonstrating trace.

First, Section 4.1 shows how to use multivalued decision diagrams and set decision diagrams to encode the state space of a system, as well as a compact way to represent transitions of a Petri net (or in general, vector addition systems). Our implementation of set decision diagrams are described in Section 4.2. Finally, Section 4.3 discusses basic state space exploration algorithms in terms of the different decision diagram representations.

4.1 Symbolic Representations

When dealing with symbolic model checking, an important question is how to encode the states and transitions of the system. In this section, two types of decision diagram based encodings are given for states, and a lightweight representation is proposed specifically for Petri net transitions.

4.1.1 Representation of the State Space

The most trivial way to represent a state space of a model is to store its state graph, explicitly enumerating the different states and transitions.

Example. Figure 2.2 illustrates a Petri net and its state space. A marking of this net can be stored as an integer vector (or tuple) of length three: the first integer referring to the token count of *p1*, the second to *p2* and the third to *p3*. In this case, the initial state can be represented as *[0, 2, 0]*. The state space can be stored as a set of the vectors encoding

the possible states. This Petri net has six distinct states, so this approach gives the set $\{[0,2,0],[0,1,1],[0,0,2],[1,1,0],[1,0,1],[2,0,0]\}$ as the state space.

The example above clearly shows that the explicit storage is very straight-forward, but also very naive. Usually. there are multiple states in which the token count of a place is the same (e.g., in the example above, *p1* has zero token in three different states), which causes a redundancy. The strength of the decision diagram encoding is that it exploits the redundancy in the state space to achieve a compact storage.

4.1.1.1 Encoding States with Multivalued Decision Diagrams

MDDs are commonly used to represent sets of states, e.g., in [18]. As mentioned in Section 2.4.2, an MDD encodes an integer function $f(x_1, ..., x_n) \longrightarrow \{0, 1\}$. Suppose x_i refers to the token count of the i-th place, and $f(x_1, ..., x_n)$ is 1 *iff* $[x_1, ..., x_n]$ is part of the encoded set. The levels of the MDD then correspond to places in the Petri net. MDDs have set-like operations as seen in Section 2.4.2.1, thus it is possible to directly manipulate this representation.

Example. The Petri-net on Figure 2.2 has three places, so the MDD encoding the state space will have three levels. Let the first level represent *p1*, the second *p2* and the third *p3*. The MDD representation of the state space can be seen on Figure 4.1, with paths from the root node to the terminal **1** denotes the encoded vectors.

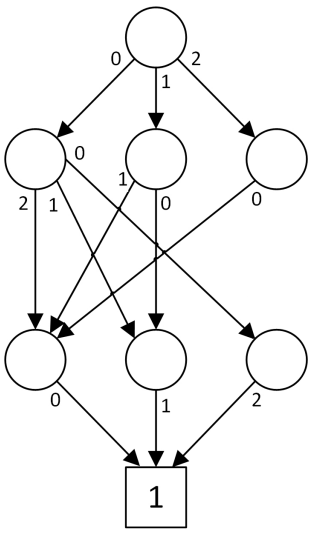

Figure 4.1. An MDD representation of the state space of the Petri net on Figure 2.2

Figure 4.1 above illustrates the way an MDD provides a less redundant way to encode the state space. However, this method is still not redundancy-free.

4.1.1.2 Ecoding States with Set Decision Diagrams

There are models which are structurally symmetrical, and this causes redundancy in the MDD representation of their state spac. Suppose a model has similar, repeated sub-models, or *components*. If these components can be arranged into a hierarchy, then we call it a *hierarchical model*.

Example. An example for hierarchical models could be the dining philosophers model. This model has multiple philosophers around a table, their meals in front of them, and a fork between every neighboring philosopher. A fork can be used by only one philosopher, and a philosopher needs both forks next to him to eat. For more information of the model, see Appendix A.

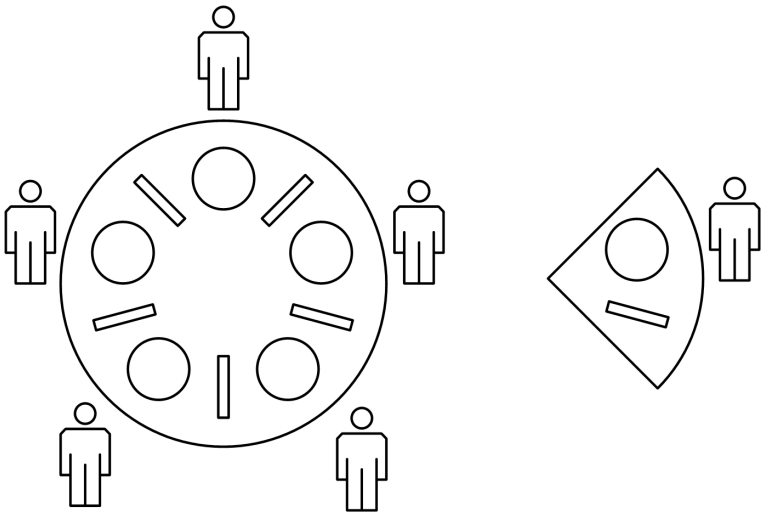

Figure 4.2. The dining philosophers problem for five philosophers.

As Figure 4.2 shows, the problem is composed of similar, repeated components. In this case, a component consists of a philosopher and a fork. These components have similar behavior and also similar states, so the MDD has repeating patterns (see Figure 4.3).

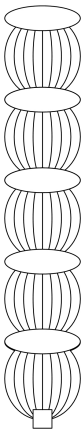

Figure 4.3. Approximate shape of an MDD encoding the state space of the dining philosophers problem.

To make the state space representation even more compact, set decision diagrams can be used (see Section 2.4.3). Using SDDs, the symmetric parts can be encoded only once, and their recurring appearance can be referenced by the use of edge labels.

Example. The aforementioned dining philosophers model can be arranged into three hierarchy levels: the topmost level, with five philosopher-fork pairs, the middle level, encoding the combination of a philosopher and a fork, and the lowest level, which encodes the states of philosophers and forks in MDDs. A route to the terminal **1** in the first hierarchy level references the second hierarchy level five times by its labels. A route in the second hierarchy level has two reference to the lowest hierarchy level (containing MDDs), with one label pointing to the MDD node encoding the states of a philosopher and another one encoding a fork.

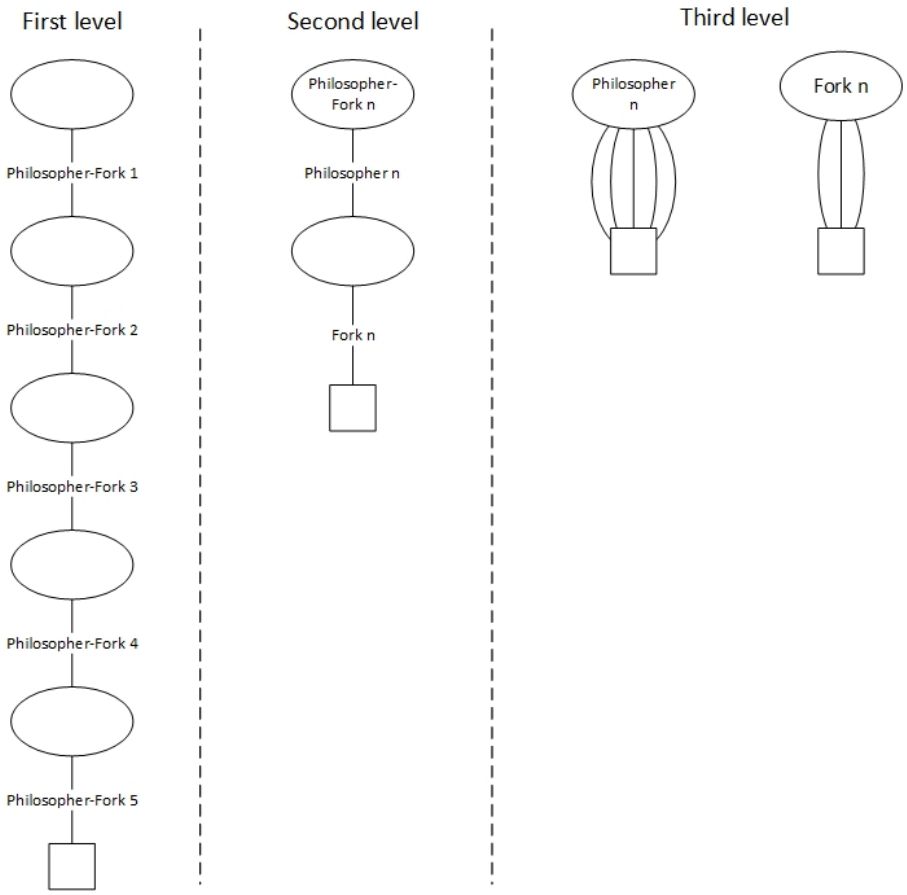

Figure 4.4. Schematich figure of a hierarchical encoding of the state space of the dining philosophers problem.

Figure 4.4 shows a schematich figure of the resulting SDD hierarchy. Instead of storing the broad, branching parts of the state space multiple times, now they are encoded only once on the MDD level, and referenced multiple times by labels on the upper levels. Using this method, the redundancy of the state space caused by the hierarchic and repeating structure of the model is significantly reduced. For an even more detailed example on using SDD hierarchy levels, see [13].

4.1.2 Encoding of Petri Net Transitions

Symbolic algorithms generate new reachable states from the already explored part of the state space. To find these states, the algorithm has to know the transitions and their exact effect. Thus we have another problem beside the representation of the state space – the representation of transitions and their firing rules.

In case of Petri nets, the weight functions W^- and W^+ (see Section 2.1) are a sufficient representation when working with MDDs, because levels of the diagram are associated with places. However, if we want to represent the state space with SDDs, level numbers do not correspond to places anymore, because the same diagram can encode the states of different components. In other words, the meaning of a level is now context dependent.

To solve this, we developed a way to represent the transitions hierarchically. The hierarchy of a model is basically a tree structure, so we encoded transitions in trees with the exact same structure as the model. In this construct, the missing context is reintroduced by processing the proper subtree of the transition encoding.

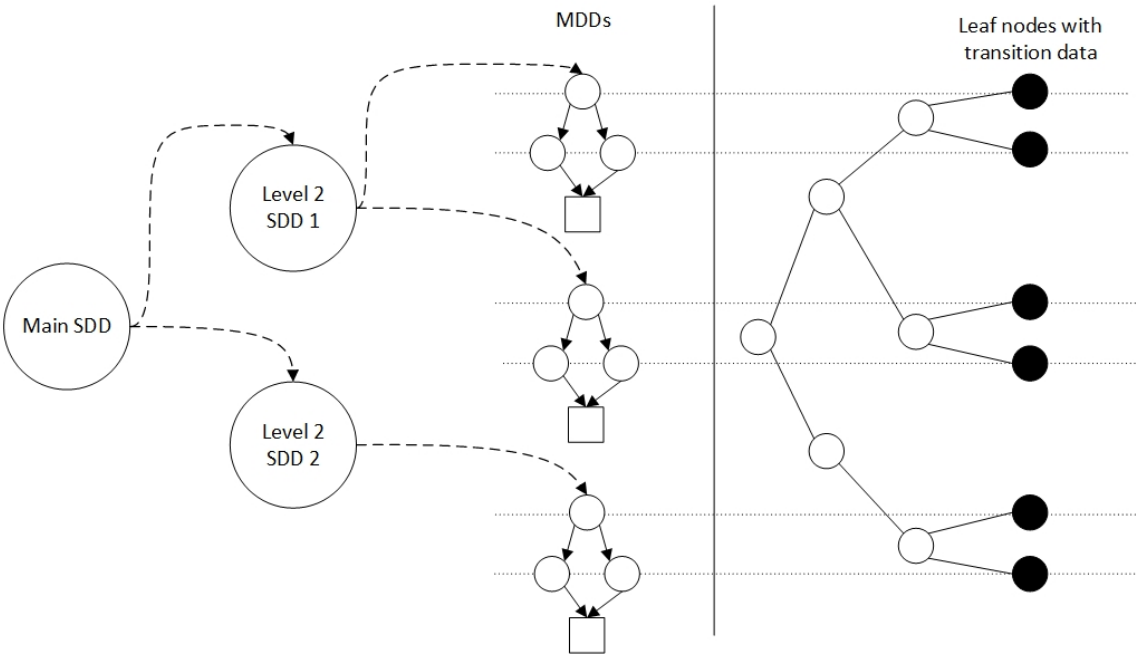

Figure 4.5. A hierarchical structure of decision diagrams, and the tree structure of the transitions

Figure 4.5 depicts a decision diagram hierarchy along with a tree describing a transition. Leaf nodes correspond to levels of MDDs and they encode the effects of the transition in two integer values: the number of tokens removed and added to the corresponding place. Every non-leaf node corresponds to a decision diagram, with the ith child node giving context to the labels of the ith level of the SDD. This relation is formalized in the following definition:

Definition 7 (Hierarchic transition tree).
A *transition tree* describing a transition in a hierarchic Petri net is a directed tree, with

a node set V consisting of internal nodes and leafs (L), an edge set E, and a function $T : L \longrightarrow (\mathbb{Z}, \mathbb{Z})$ such that $T(w) : (W(p,t), W(t,p))$ for $\forall w \in L$. Every component in the hierarchy is associated with an internal node $v \in V \setminus L$. Every place $p \in P$ in Petri net is associated with a corresponding leaf node $w \in L$. There is an edge $(x, y) \in E$ in the tree *iff* the component or place corresponding to y is the part of the subcomponent of the component corresponding to x. ▪

4.2 Implementing Set Operations on Set Decision Diagram

This section introduces algorithms to implement set operations on set decision diagrams. These operations were not defined in [6] and [13] where SDDs were introduced, but they defined the so-called *homomorphisms*. A homomorphism is an abstract operation that maps canonical SDD nodes to canonical SDD nodes. They can be used to define various operations, including set operations, integer arithmetic or even variable assignments. In this sense, the following operations can be regarded as our custom homomorphisms.

The following algorithms will be doubly recursive: for every edge in the resulting diagram the label and the child node has to be computed by a recursive calls to another set operation. The recursions computing the child nodes traverse the current diagram and will be terminated on the terminal level. The recursions computing the labels to the new children traverse the hierarchy, and will eventually result in a simple MDD operation (introduced in Section 2.4.2.1). Furthermore, reduction rules must be enforced during the computation.

When defining the following algorithms, one must consider the path passing through the current decision diagram node(s). In every case we will consider how to compute the child and label nodes of the resulting edges such that the resulting node is canonical.

4.2.1 Intersection

The intersection of two SDDs is an SDD encoding the set of vectors encoded by both of the operands. Our first consideration is that every path that is present in the intersection has to pass trough some label in both the first and the second operand. Therefore, to compute the labels of resulting edges, we have to intersect labels of every edge of the first operand with labels of every edge of the second operand. The resulting intersection are the candidates to be labels on the edges of the result node. Furthermore if a path got through the labels, it still has to reach the terminal one in both diagrams, so the intersection of the children nodes also has to be computed for every edge candidate. If the intersection is non-empty, we add an edge to the result node labeled with the intersection of the labels leading to the intersection of their corresponding children. This construction inherently satisfies both of the reduction rules.

Algorithm 1: Intersection of two SDD nodes

Input: SDD nodes v and w on the same level in two canonical SDD graph
Output: SDD node z encoding the intersection of the inputs

```
1  if v = 0 ∨ w = 0 then
2  |  return 0;
3  end
4  if v = w then
5  |  return v;
6  end
7  z ← new node on the same SDD level as operands;
8  foreach outgoing edge e of v do
9  |  foreach outgoing edge f of w do
10 |  |  c ← child(e) ∩ child(f);
11 |  |  l ← label(e) ∩ label(f);
12 |  |  if n ≠ 0 and l ≠ 0 then
13 |  |  |  create edge z --l--> c ;
14 |  |  end
15 |  end
16 end
17 if z does not have any edges then
18 |  z ← 0;
19 end
20 return z;
```

4.2.2 Union

The union of two SDDs is an SDD encoding the set of vectors encoded by any of the operands. Our first consideration is that every path that is present in both of the operands has to path trough some label in both the first and the second operand. But unlike in the case of intersection, the corresponding children has to be merged. Furthermore, in this case, paths only present in one of the operands also have to be included in the result. Therefore, the parts of labels not present in the intersections has to be added as a new edge with their original child node.

While the definition of intersection implied that the reduction rules are satisfied, it is not the case here. To see this, consider the following. Assume that the label of an edge had an intersection with the label of another edge, whose child was the same. In this case the algorithm above would create two edges that lead to the same child. Therefore, the last step of the algorithm has to apply the first reduction rule (defined in Section 2.4.3) by merging the labels of the edges leading to the same child node. The other rule is still guaranteed by the construction.

4.2.3 Subtraction

The subtraction of an SDDs from another SDD encoding the set of vectors encoded only by the latter. In our implementation, we subtracted the right operand from the left operand.

Algorithm 2: Union of two SDD nodes

Input: SDD nodes v and w on the same level in two canonical SDD graph
Output: SDD node z encoding the union of the inputs

if $v = \mathbf{1}$ *or* $w = \mathbf{1}$ **then**
  return 1;
end
if $v = w$ **then**
  return v;
end
$z \leftarrow$ new node on the same SDD level as operands;
foreach *outgoing edge* e *of* v **do**
  foreach *outgoing edge* f *of* w **do**
    $c \leftarrow \text{child}(e) \cup \text{child}(f)$;
    $l \leftarrow \text{label}(e) \cap \text{label}(f)$;
    if $l \neq \mathbf{0}$ **then**
      create edge $z \xrightarrow{l} c$;
      replace e with $v \xrightarrow{label(e) \setminus l} \text{child}(e)$;
      replace f with $v \xrightarrow{label(f) \setminus l} \text{child}(f)$;
    end
  end
end
add every remaining edges from v and w to z;
while $\exists a_i, a_j, x : z \xrightarrow{a_i} x \wedge z \xrightarrow{a_j} x$ **do**
  remove edges $z \xrightarrow{a_i} x$ and $z \xrightarrow{a_j} x$;
  create edge $z \xrightarrow{a_i \cup a_j} x$;
end
return z;

Our first consideration is that every path that is present in both of the operands has to path trough some label in both the first and the second operand. But unlike in the case of intersection, the corresponding children has subtracted, with subtracting the children of the right operands from the children of the left operand, thus eliminating the paths found in both of the operands. Furthermore, in this case, paths only present in the left operand also have to be included in the result. Therefore, the parts of labels from the left operand not present in the intersections has to be added as a new edge with their original child node.

While the definition of intersection implied that the reduction rules are satisfied, it is not the case here. To see this, consider the following. Assume that the label of an edge had an intersection with the label of another edge, whose children nodes are disjunct. In this case the algorithm above would create two edges that lead to the same child. Therefore, the last step of the algorithm has to apply the first reduction rule (defined in Section 2.4.3) by merging the labels of the edges leading to the same child node. The other rule is still guaranteed by the construction.

Algorithm 3: Subtraction of an SDD node from another SDD node

Input: SDD nodes v and w on the same level in two canonical SDD graph
Output: SDD node z which is the subtraction of w from v

```
1  if v = 0 or w = 0 then
2  |  return v;
3  end
4  if v = w then
5  |  return 0;
6  end
7  z ← new node on the same SDD level as operands;
8  foreach outgoing edge e of v do
9  |  foreach outgoing edge f of w do
10 |  |  c ← child(e) \ child(f);
11 |  |  l ← label(e) ∩ label(f);
12 |  |  if l ≠ 0 then
13 |  |  |  create edge z --l--> c;
14 |  |  |  replace e with v --label(e) \ l--> child(e);
15 |  |  end
16 |  end
17 end
18 add every remaining edges from v to z;
19 while ∃a_i, a_j, x : z --a_i--> x ∧ z --a_j--> x do
20 |  remove edges z --a_i--> x and z --a_j--> x;
21 |  create edge z --a_i ∪ a_j--> x;
22 end
23 return z;
```

4.3 Exploring the State Space

In this section, the necessary algorithms for exploring the state space are presented: the algorithm responsible for firing the transitions in the hierarchical model, and common path finding algorithms.

4.3.1 Firing a Transition

To traverse through the state space, an algorithm has to be defined to generate the set of states reached from an already discovered set of states after firing an enabled transition. Algorithms 4 and 5 are designed to use the transition trees for firing transitions by calling doubly recursion in accordance with the structure presented in Section 4.1.2. The first algorithm is called on the root node of the SDD on the top hierarchy level, and when it travels down to the lowest SDD hierarchy levels, then eventually it calls the MDD variant of the algorithm, when the labels are referring to MDDs. This MDD variant algorithm will determine if a transition is enabled, and if it is, then calculates the new token count on the place of the Petri net corresponding the MDD level within the current context.

The other recursion traverses downwards in the levels of the SDD to calculate the child nodes, and terminating on the terminal level.

Algorithm 4: Fire transition algorithm for SDD nodes

Input: SDD node v, tree node t from the transition tree of a transition
Output: SDD node w encoding the sub-states resulted from firing the transition corresponding to the tree of t on the states encoded by v

```
1  if level(v) = 0 then
2  |  return v;
3  end
4  w ← new node on the same MDD level as the operands;
5  foreach outgoing edge e of v do
6  |  n ← fireTransition(node(e), t);
7  |  l ← fireTransition(label(e), t[level(v)]);
8  |  if l ≠ 0 ∧ n ≠ 0 then
9  |  |  create edge w --l--> n;
10 |  end
11 end
12 if w don't have any edges then
13 |  return 0;
14 end
15 while ∃a_i, a_j, x : w --a_i--> x ∧ w --a_j--> x do
16 |  remove edges w --a_i--> x and w --a_j--> x;
17 |  create edge w --a_i ∪ a_j--> x;
18 end
19 return w;
```

4.3.2 Traversing Algorithms

The two methods of traversing the state space presented here are the widely known *breadth first search* (BFS), and a similar algorithm called *chaining loop*.

BFS starts at the initial state, and explores the states reachable by firing only one transition with every iteration. That means BFS finds every reachable state with the minimum required firings needed to reach that state from the initial state. With every iteration, BFS constructs the result for the firing of every enabled transition on the currently explored state space, and then it sums the results and this original set.

Chaining loop fires every transition after each other as well, but this algorithm makes the union of the result set of states with the original state space straightaway. This means that on average, chaining loop will find most of the reachable states sooner then the BFS, but it can only give a loose upper estimation on firing needed to reach a given state.

Algorithm 5: Fire transition algorithm for MDD nodes

Input: MDD node v, tree node t from the transition tree of a transition
Output: MDD node w which encodes the sub-states resulted from firing the transition corresponding to the tree of t on the states encoded by v

```
1  if level(v) = 0 then
2  |  return v;
3  end
4  w ← new node on the same MDD level as the operands;
5  i = 0;
6  foreach outgoing edge e of v do
7  |  if n[i] ≠ 0 ∧ i − t[level(v)].From >= 0 then
8  |  |  x ← fireTransition(node[i], t);
9  |  |  if x ≠ 0 then
10 |  |  |  w[i − t[level(v)].From + t[level(v)].To] ← x;
11 |  |  end
12 |  end
13 end
14 if w don't have any edges then
15 |  w ← 0;
16 end
17 return w;
```

Algorithm 6: Breadth first search

Input: a set s containing the initial state only
Output: the encoded state space

```
1  while new states are found do
2  |  d ← empty set;
3  |  foreach every transition t do
4  |  |  d ← d ∪ fireTransition(s, t);
5  |  end
6  |  s ← s ∪ d;
7  end
8  return s;
```

Algorithm 7: Chaining loop

Input: a set s containing the initial state only
Output: the encoded state space

```
1  while new states are found do
2  |  foreach every transition t do
3  |  |  s ← s ∪ fireTransition(s, t);
4  |  end
5  end
6  return s;
```

Chapter 5

Guided Partial Order Reduction

This chapter introduces our new explicit model checking algorithm building on concepts of directed search and partial order reduction. The proposed method focuses on efficiently generating traces to unsafe states even in large and complex systems. To handle cases where the algorithm has to prove unreachability, the efficiency of partial order reduction (see Section 2.5) is employed to complement the guiding heuristics. Nevertheless, our approach tends to sacrifice performance in the unreachable case in favor of efficient trace generation. This is in accordance with the strategy of Chapter 3 – proving unreachability is assumed to be the task of a symbolic model checker. Due to the techniques our approach builds on, we call it *Guided Partial Order Reduction (GPOR)*.

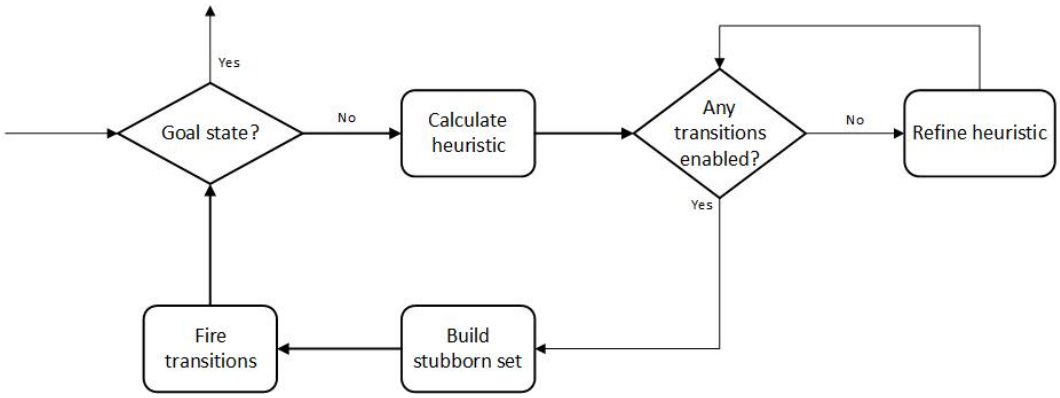

Figure 5.1. General workflow of guided partial order reduction

5.1 Guided Search in Petri Nets

Our contribution in the field of explicit model checking is a new heuristic to perform directed model checking on Petri nets. The introduced new algorithm incorporates this heuristic into partial order reduction to build stubborn sets using the information about which transitions should be fired in order to "get closer" the goal state. This section introduces the heuristic and our notion of "closer" to introduce the theoretical foundations of the algorithm.

For the rest of this section, we assume that the reachability criteria φ is given as the conjunction of positive atomic propositions, i.e., $\varphi = \ell_1 \wedge \cdots \wedge \ell_n$. Section 5.2 will discuss methods to reduce other forms of φ to this form.

5.1.1 Closed-Quarters Navigation with UP Sets

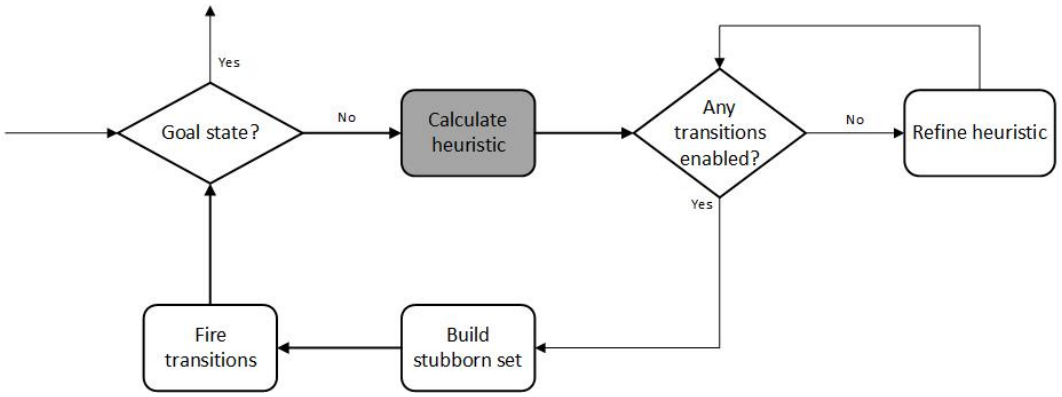

Figure 5.2. Place of UP sets in our workflow

Our heuristic is based on the information extracted from the structure of Petri nets: the algorithm investigates if transitions can modify the number of tokens on "interesting" places – e.g., places that appear in the requirement. Using this heuristic the algorithm can prioritize transitions and fire the more promising ones first. As the Petri net reachability problem for bounded Petri nets is computationally hard, we can not ensure to find the solution efficiently all the time. As a first approach, we define a set of "good" transitions called the UP set. The idea of our algorithm is based on [15], however we propose significant improvements.

Definition 8 (Positive UP set). Let $UP_i^+ = \{t \mid t \in \bullet sub(\ell_i) \wedge W^*(t, sub(\ell_i)) > 0)\}$, i.e., all the transitions that adds more tokens on $sub(\ell_i)$ than it removes. ▪

Definition 9 (Negative UP set). Let $UP_i^- = \{t \mid t \in sub(\ell_i) \bullet \wedge W^*(t, sub(\ell_i)) < 0)\}$, i.e., all the transitions that adds more tokens on $sub(\ell_i)$ than it removes. ▪

Such positive and negative UP sets can be statically computed for every atomic propositions of φ. In order to use the sets as guiding heuristics during state-space exploration the proper sets have to be chosen for every unsatisfied atomic proposition.

Observation: If the operator of ℓ_i is $\neq$, both sets are useful. In this case we do not know whether we want to increase or decrease the token count, we only know that the present marking is not satisfying the atomic proposition. The choice between a positive or negative UP set is determined by the values of the functions *Less* and *More* (introduced in Section 2.3.1.1). Observation of the definitions of the functions reveals that in case of operator "$\neq$", both UP sets can help to satisfy the atomic proposition. This is reflected in the following definition of UP sets.

Definition 10 (UP set). Let s be the marking of a Petri net and $\varphi = \ell_1 \wedge \cdots \wedge \ell_n$ a reachability criteria. Then $UP(s) = (\bigcup_{\{\forall \ell_i \in \varphi \mid Less(\ell_i,s)\}} UP_i^+) \cup (\bigcup_{\{\forall \ell_i \in \varphi \mid More(\ell_i,s)\}} UP_i^-)$ is the UP set corresponding to s. ▪

An UP(s) set is the union of the $UP_i^{\pm}$ with regard to all the atomic propositions based on the state s. The role of the UP set is to characterize "good transitions" based on the following theorem.

Theorem 1 (Theorem of UP sets). Every ρ path starting in s and leading to a goal state g according to reachability criteria φ contains at least one firing from the transitions in $UP(s)$. ▪

Proof 1. Indirect proof. Assume that a current state is not a goal state and there is a path ρ starting in s leading to a goal state g according to reachability criteria φ without firing a transition in UP(s).

If $\exists \ell_i \in AP(\varphi) : Less(\ell_i, s)$ then a transition along path ρ has to raise the tokencount of $sub(\ell_i)$. However transitions that can raise the token count on $sub(\ell_i)$ is by definition in UP_i^+, which is a subset of UP(s) according to Definition 10, so this assumption leads to a contradiction.

If $\exists \ell_i \in AP(\varphi) : More(\ell_i, s)$ then a transition along path ρ has to decrease the tokencount of $sub(\ell_i)$. However transitions that can decrease the token count on $sub(\ell_i)$ is by definition in UP_i^-, which is a subset of UP(s) according to Definition 10, so this assumption leads to a contradiction.

Otherwise, the definition of $Less$ and $More$ implies that every atomic propositions of φ is satisfied which in term implies that s is a goal state. This is again a contradiction. □

Note that both UP_i^+ and UP_i^- can be computed in a preprocess step, because they do not depend on the current state, only on structural properties off the model and an atomic proposition. This way the calculation of UP(s) is a single decision based on the current state and an union of the chosen sets. This can significantly reduce the overhead of UP set computation during state space exploration.

5.1.2 Road Signs in the Net: UP Layers

UP sets can be used to guide the search towards goal states, unless none of the chosen transitions are enabled. To mitigate this situation, additional "road signs" has to be defined in the model to guide the search up until a point where one of the transitions in $UP(s)$ become enabled. The basic idea of the following heuristic is that finding a state in which a transition of $UP(s)$ is enabled is similar to the original problem of reaching a goal state.

To implement this idea, layered structure is built for every $UP_i^{\pm}$, with every layer including transitions that can enable transitions of the previous layer.

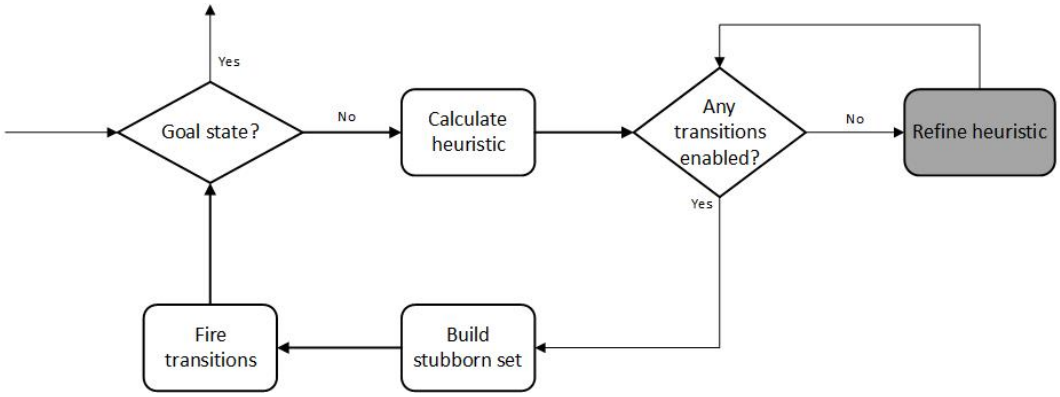

Figure 5.3. The place of UP layers in our workflow

Definition 11 (UP^+ layer). For every atomic proposition $\ell_i \in AP(\varphi)$, we define the UP^+ as follows:

$$UP^+_{n,i} = \begin{cases} n = 0: & UP^+_i \\ \forall n > 0: & \{t | t \in \bullet\bullet(UP^+_{n-1,i}) \wedge p \in \bullet(UP^+_{n-1,i}) \wedge W^*(t,p) > 0\} \setminus \bigcup_{j=0}^{j<n} UP^+_{n,j} \end{cases}$$

Definition 12 (UP^- layer). For every atomic proposition $\ell_i \in AP(\varphi)$, we define the UP^- as follows:

$$UP^-_{n,i} = \begin{cases} n = 0: & UP^-_i \\ \forall n > 0: & \{t | t \in \bullet\bullet(UP^-_{n-1,i}) \wedge p \in \bullet(UP^-_{n-1,i}) \wedge W^*(t,p) > 0\} \setminus \bigcup_{j=0}^{j<n} UP^-_{n,j} \end{cases}$$

The topmost layer ($UP^\pm_{0,i}$) is the UP set defined in the previous section, then the role of lower layers are to enable transitions in the higher layers. In addition, transitions can only belong to the highest possible layer, meaning that every transition belongs to at most one UP layer. Note that $UP^\pm$ layers are still statically computable.

Choosing between $UP^\pm$ layers again depends on the current state defined in the following function.

Definition 13 (UP layers). Let s be the marking of a Petri net and $\varphi = \ell_1 \wedge \cdots \wedge \ell_n$ a reachability criteria. Then $UP_n(s) = (\bigcup_{\{\forall i \in I | Less(\ell_i, s)\}} UP^+_{n,i}) \cup (\bigcup_{\{\forall i \in I | More(\ell_i, s)\}} UP^-_{n,i})$ where s is the current state of the Petri net. ▪

The definition is similar to Definition 10, $UP_n(s)$ is built based on the current state. Every layer helps enabling the layer "on top of it", while the topmost layer helps reaching a goal state. The representation of these sets in the actual Petri net is a set of transitions that influence the reachability of a goal state surrounded by similar sets that influence the enabling of these transitions.

The following definition characterizes the UP layer that is a local best choice according to our heuristics.

Definition 14 (Highest available UP layer). The highest available UP layer is the highest layer that contains an enabled transition in state s, and is denoted by $UP^*(s)$. ▪

To show that it is indeed a good choice to fire the transitions of $UP_n(s)$ we the following lemma and theorem.

Lemma 1. Assuming that there is no transition enabled in $UP_n(s)$, there is no path ρ starting in s that contains a firing from $UP_n(s)$ but not from $UP_{n+1}(s)$. ▪

Proof 2. Similarly to the proof of Theorem 1, it is easy to see that a transition in $UP_n(s)$ cannot become enabled unless tokens are placed onto its input places, but transitions raising the token count on these places are in $UP_{n+1}(s)$. ▪

Theorem 2 (Theorem of UP layers). Every ρ path starting in s and leading to a goal state g according to reachability criteria φ contains at least one firing from the transitions in $UP^*(s)$. ▪

Proof 3. Inductive proof. If $UP^*(s) = UP_0(s)$, then Theorem 1 provides a proof. If $UP^*(s) = UP_n(s)$ and $n > 0$, then Definition 14 implies that none of the transitions in $UP_{n-1}(s)$ is enabled. Based on Lemma 1, transitions in $UP_{<n}(s)$ can only become enabled if at least one transition is fired from $UP_n(s)$.

Theorem 2 implies that it is inevitable to fire a transition from $UP^*(s)$ if we want to reach a goal state. This serves as the basis of our guided model checking algorithm, presented in Section 5.2.

In order to be able to compute $UP^*(s)$ it is important to prove that the number of non-empty UP sets is finite.

Lemma 2 (UP layer calculation is finite). There is a finite integer n such that $UP_n(s) = \emptyset$. ▪

Proof 4. If we do not include any transition in $UP_n(s)$ then the statement holds. If we include in every layer at least one transition, sooner or later (in a finite Petri net) the remaining set of transitions we can choose from will become $\emptyset$, because we do not include a transition more than once (see Definitions 11,12,13). ▪

If we find an empty UP layer $UP_n(s) = \emptyset$, we can terminate the calculation of layers, because an empty set of transitions has an empty set of input places, which, in turn, has an empty set of input transitions (i.e., $UP_{n+1}(s) = \emptyset$)

It is also worth noticing that if $UP^*(s) = \emptyset$ there is no path leading to a goal state (this is a consequence of Theorem 2).

5.2 Introducing the new algorithm

In this section we will introduce our guided partial order reduction algorithm that uses the above described heuristic. The algorithm can generate traces to reachable states very efficiently even in huge models.

As we mentioned in Section 2.5, there are many ways to create stubborn sets, and one way is to heuristically guess which transitions are "helpful" to reach a goal state. In this work we use UP layers as a guiding heuristic. We will construct our stubborn sets in a way that they contain all enabled UP layer transitions. From the theoretical proofs above, we know that transitions of the UP layers will eventually guide the search to a goal state. Partial order reduction is necessary to make sure that the search is complete.

5.2.1 Preprocess Steps

As we described above, the $UP^{\pm}$ sets and layers are computable before the actual model checking procedure, saving computation time during the actual checking. If we would have to compute the UP layers in every state from scratch, our algorithm would be very slow and redundant, because (even with partial order reduction) the state-space is huge.

5.2.1.1 UP Layer Cache

As mentioned in Section 5.1, UP layers can be precomputed statically to boost the performance of the algorithm. However this tabel can grow unnecessarily large, so it is often an overhead to compute all of the layers in advance.

For this reason our algorithm uses a lazy computation strategy that only computes the first layer initially and delays the computation of lower layers until necessary, but uses a cache to save redundant computations.

5.2.1.2 Negation Normal Form

For our algorithm to work it is necessary to convert the reachability criteria to negation normal form. Such a form can be reached using the DeMorgan rules on the expression-tree recursively:

- $\neg(A \wedge B) = \neg A \vee \neg B$
- $\neg(A \vee B) = \neg A \wedge \neg B$

Example:

$\neg(\neg(A \wedge B) \vee C) \wedge D = \neg A \vee \neg B \vee C \vee \neg D$

With these rules we can push the negations into the leaf-nodes, and absorb them by negating the operators in ℓ. The operator negation rules are the following:

- $=$ in negation is $\neq$ and $\neq$ in negation is $=$
- $<$ in negation is $>=$ and $>=$ in negation is $<$
- $>$ in negation is $<=$ and $<=$ in negation is $>$

This step is necessary because negations must be built into the UP sets.

5.2.2 Exploration of the Reduced State-Space

In this section we will show the actual state-space exploration process, then we introduce a method that produces stubborn sets without checking the conditions described in Section 2.5 but inherently satisfying them.

5.2.2.1 Computing Stubborn Sets

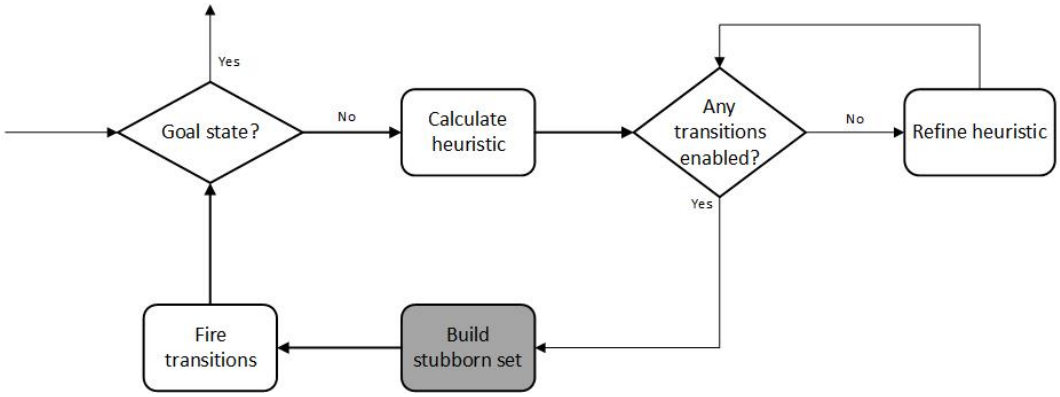

Figure 5.4. Place of partial order reduction in our workflow

We compute the stubborn sets in a way that inherently satisfies properties in Definition 6. The essence of the method is to group the transitions in a Petri net to disjunct sets based on their dependencies. Two transitions belong to the same set if one of them can disable the other, i.e., transitions that have common input places. This is a stricter rule than the ones described in Definition 6, but it is a relatively cheap over-approximation and can be derived from the Petri net structure only.

Example:

On figure 5.5 there are three groups: $Group1 = \{t_1, t_2, t_5\}$, $Group2 = \{t_3\}$, $Group3 = \{t_4\}$. The basis of the grouping is the common input place, because in ordinary Petri nets, the only way to disable an enabled transition is to remove tokens from one (or more) of its input places. A group is defined as the transitive closure of this dependecy relation. If we find a transition that disables another, they belong to the same group together with all the other transitions that can disable either. Every time we include a transition in a group, we must check again for the transitions it has common input places with, and add them to the group. For example, in figure 5.5, t_1 cannot disable t_5 but both can disable t_2

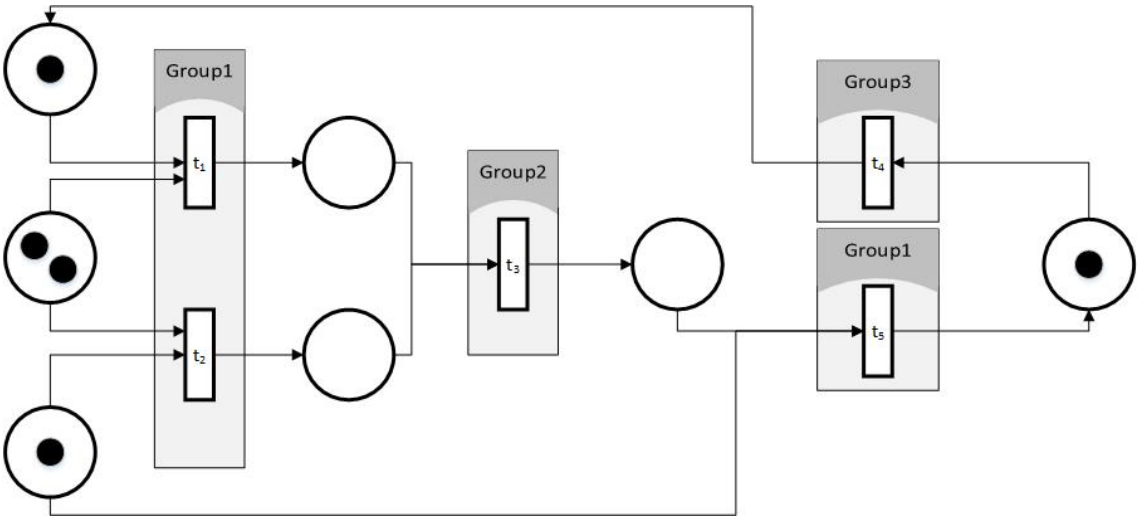

Figure 5.5. Grouping of transitions based on ability to disable each other

so they belong to the same group (Group1). We call these groups of transitions *dependecy groups*.

To satisfy Definition 6 we must make sure that if we include a transition in a stubborn set, we include the whole dependency group it belongs to. To use the defined heuristics we start from the best UP layer, $UP^*(s)$, then add transitions of related dependency groups.

- **D0:** This requirement is only violated *iff* $UP^*(s)$ is empty, but according to Theorem 2 this also means that goal states are not reachable from this state.
- **D1 and D2:** STUB members cannot disable transitions outside the STUB set and a non-STUB members cannot disable a STUB members, due to the definition of dependency groups thus inherently satisfy **D1** and **D2**.

5.2.2.2 Discovering the Reduced State-Space

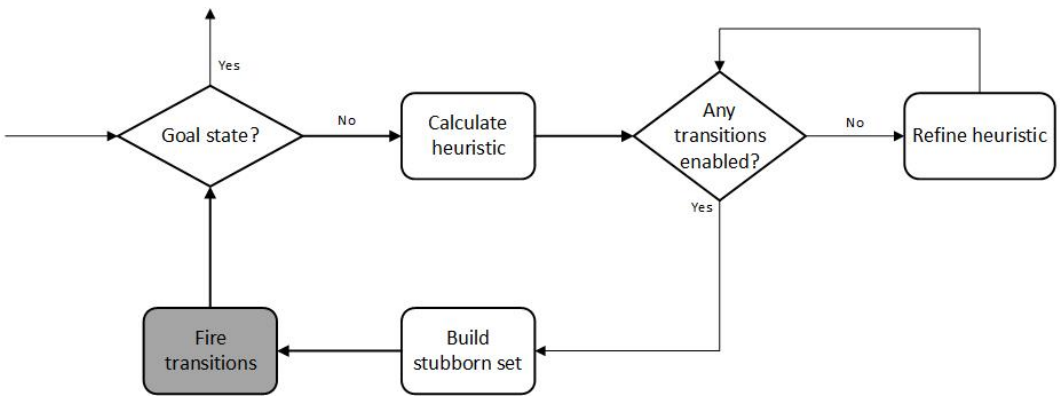

Figure 5.6. Place of state-space discovery in our workflow

The core of the state-space exploration algorithm is depth first search. We start the search from the initial state of the Petri net. The essence of our approach is that instead of discovering every neighbor the current state. we compute a stubborn set and only fires the transitions in it. As mentioned before, the core of each stubborn set is an UP layer so heuristic search is achieved by firing those transitions first.

Algorithm 8 presents the core of our explicit guided partial order reduction model checker. In accordance with our plans, we sacrifice the unreachable cases in favor of generating shorter traces faster. This is caused by the fact that if we backtrack to a state, we does not pop it from the stack immediately (like we would in a traditional DFS), instead, we try the stubborn set built around the next UP layer. This is a key step in the algorithm, because there is no guarantee that the ”best“ stubborn set preserves reachability, it is possible that the worse best stubborn set contains the key transition to reach a goal state. Heuristics are only best guesses to direct the state-space traversal, in accordance with our workflow shown in 5.1.

5.2.3 Strengths and Weaknesses of UP Sets

The heuristics we described are very efficient in guiding the DFS state-space discovery. Combined with partial order reduction we can efficiently check models for safety properties, but there is a slight complication when it comes to complex specifications. If the reachability criteria describe two disjunct sets of unsafe states that mutually exclude each other, we face a problem illustrated on Figure 5.7.

If the heuristic tries to direct the search towards multiple directions, although it gives right answer, only a much longer longer, suboptimal traces is produced. this problem can be solved by transforming the reachability criteria (φ) to disjunctive normal form (DNF). By this, way we can decompose the problem into easier sub-problems. These problems can be solved sequentially or in parallel, yielding shorter counterexamples. In case of sequential execution, unreachable sub-problems can cause a memory and runtime overhead, this is why it is advantageous to extract the exact description of a reachable goal state from a symbolic model checking run.

Algorithm 8: Limited depth first search to produce reduced state-space

Input: the Petri net structure, the reachability formula (φ), and *maxdepth* the depth limit to the search

Output: trace to a goal state if reachable, an empty set if not

```
1  var Stack and ExampleStack are stacks storing states;
2  var DiscoveredStates is a set to store the discovered states;
3  var depth=0 is an integer;
4  Set the Petri net to the initial state; Put the initial state on Stack;
5  while Stack is not empty do
6  |  var s = Stack.pop();
7  |  try to insert s to DiscoveredStates;
8  |  if DiscoveredStates already had s and we fired all of its UP layers then
9  |  |  if s is the the top element in ExampleStack then
10 |  |  |  ExampleStack.pop();
11 |  |  end
12 |  |  Stack.pop();
13 |  |  continue;
14 |  end
15 |  else
16 |  |  ExampleStack.push(s);
17 |  |  ++depth;
18 |  end
19 |  if s satisfies φ then
20 |  |  return ExampleStack;
21 |  end
22 |  if depth < maxdepth then
23 |  |  var STUB = Get the next stubborn set for s;
   |  |  ;            // if we were ot in this state before, get the first
   |  |  ;                              // by different stubborn sets we mean
   |  |  ;                               // which UP layer we build it around
24 |  |  foreach t in STUB do
25 |  |  |  if t is not in DiscoveredStates then
26 |  |  |  |  var ss = the state result of firing t from current state;
27 |  |  |  |  Stack.push(ss);
28 |  |  |  end
29 |  |  end
30 |  end
31 |  if s is the the top element in Stack and we saw all of its stubborn sets then
32 |  |  Stack.pop();
33 |  |  ExampleStack.pop();
34 |  end
35 end
36 retrurn ExampleStack;
```

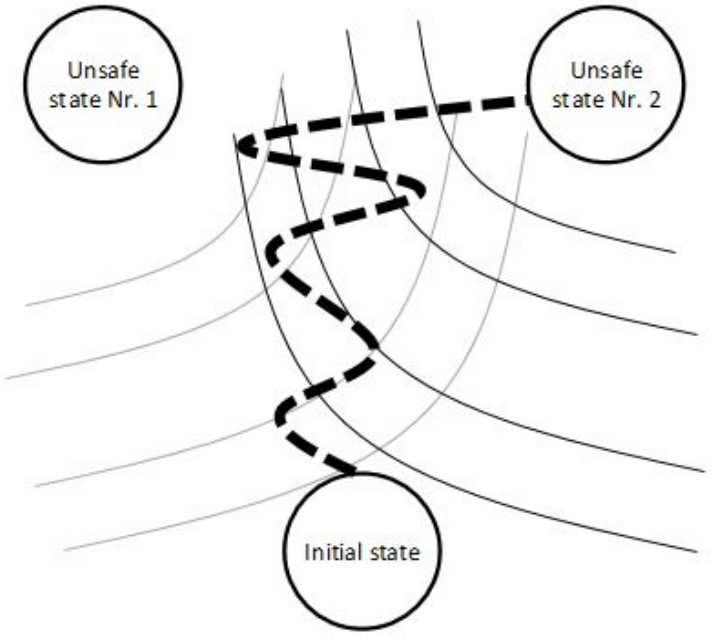

Figure 5.7. Example for two disjunct set of unsafe states and the interfering upsets

Chapter 6

Evaluation

In this chapter we present the performance results of our algorithms and data structures in terms of independent capabilities and the efficiency of combining the two approaches. Our measurements shows an interesting comparison between the symbolic and explicit techniques and presents the efficiency of the combination of the best of both worlds.

6.1 State Space Exploration Tools

We compared the capabilities of SDDs and MDDs combined with common state space exploration algorithms. In these measurements, our goal was to traverse through the whole state space, so these measurements composed exclusively from symbolic methods, and we did not use any expression to evaluate: the algorithms were finished when they found all reachable states. We tried out four different combinations for the exploration: we tried out MDDs and SDDs for data structures and breadth first search and chaining loop for the traversing algorithm. We wanted to inspect the differences between these combinations to find possibilities for future works.

6.1.1 Prototype Implementations

In order to evaluate our algorithms, we implemented prototype tools. The algorithms of Chapters 4 and 5 were implemented separately to express their loose coupling and make it possible to combine them with other solutions.

6.1.1.1 Prototypes for Symbolic Model Checking

There are only a handful of papers available on SDDs, all of them without any words on the concrete implementation [6] [13]. To the best of our knowledge, SDDs were implemented only once, in a fully different environment[1] [5], so we had to start the development of the data structure from scratches. We decided to use the C# Language and the .NET

[1] `http://ddd.lip6.fr/`

framework, to integrate our algorithms into the successor project of the PetriDotNet[2] model checking framework. We implemented the SDD as a component of the tool, so we paid attention to align the interface of the SDD to match the other data structures in the program.

6.1.1.2 Prototypes for Explicit Trace Generation Algorithms

The explicit trace generation algorithms were implemented in C++11 for performance reasons, because explicit model checkers especially need to exploit the full potential of computation resources. The prototype is a stand-alone tool with a command-line interface to be able to integrate with other tools more easily.

6.2 The Hybrid Model Checker

Our hybrid model checker follows the workflow presented on Figure 3.1. At first a symbolic model checking is performed and then, if needed the explicit guided algorithm is used for trace generation. For our hybrid model checker we used the most efficient (based on measurement results) combination of the symbolic methods presented in chapter 4.

6.3 Measurements

In this section we present the actual performance results of our tools, detailed measurement results can be found at Tables 6.1, 6.2, 6.3.

6.3.1 Process of Measuring

We performed the measurements on an system: Intel i7-3610-QM CPU @ 2.30 GHz, 6 GB DDR3-1333 MHz RAM, Microsoft Windows 8.1 operating system with .NET platform 4.6. We limited the memory usage of all configurations to a maximum of 4 GB and enforced a time limit of 10 minutes. If a test-case violated these limits we terminated it and displayed the reason of termination in the results.

At first we measured the various combinations of symbolic methods in order to decide which is the most efficient to serve as basis of our hybrid model checker. As shown in th Table 6.1 this configuration was the usage of multivalued decision diagrams (MDDs) with the chaining loop algorithm.

Then we performed the model checking with the guided partial order reduction algorithm alone without the aid of any additional information in order to be able to measure the efficiency boost of our approach.

[2] `http://petridotnet.inf.mit.bme.hu/en`

Finally we performed the measurements with our hybrid model checker and evaluated the results. In Section 6.3.3 we present interesting evaluations which can further aid the work in the field of safety property model checking.

6.3.2 Results

In this section we present the actual results of our work, the notations in the tables are the following.

- The *Model* column contains the name of the model we ran test-case on.
- The *Crit.* column contains the code of reachability criteria we asked from our model checkers. The description of codes can be found in appendix A
- The $|S|$ column shows the size of the state-space of the model, if it is unknown in the literature it contains the "?" symbol.
- The *RT* columns shows the total runtime of the process.
- The *PMU* columns shows the peak memory usage of the process.
- The *DS/TL* columns shows for the explicit algorithm the size of the discovered (reduced) state-space (DS) in comparison to the length of trace generated (TL)
- > 4 *GB* means that the process terminated due to too much memory consumption.
- > 10 *m* means that the process terminated due to reaching the time limit.
- if in a cell the symbol "–" showed it means that the cell cannot contain valuable data due to termination because of other reasons or that the data is not representative.

Model	Crit.	|S|	SDD				MDD			
			BFS		Chaining Loop		BFS		Chaining Loop	
			RT	PMU	RT	PMU	RT	PMU	RT	PMU
Dekker 10	3	6 144	> 10 m	–	> 10 m	–	1.86 s	62 MB	1.88 s	63 MB
Dekker 20	3	278 528	> 10 m	–	> 10 m	–	–	> 4 GB	–	> 4 GB
Dekker 50	3	?	> 10 m	–	> 10 m	–	–	> 4 GB	–	> 4 GB
Peterson 2	3	20 754	> 10 m	–	> 10 m	–	4.24 s	124 MB	4.28 s	123 MB
Peterson 3	3	3.408×10^6	> 10 m	–	> 10 m	–	–	> 4 GB	–	> 4 GB
Peterson 4	3	6.299×10^8	>10 m	–	> 10 m	–	–	> 4 GB	–	> 4 GB
FMS 10	3	2.501×10^9	64.57 s	15 MB	65.42 s	11 MB	5.12 s	114 MB	0.51 s	16 MB
FMS 100	3	2.703×10^{21}	> 10 m	–	> 10 m	–	–	> 4 GB	–	> 4 GB
FMS 500	3	?	> 10 m	–	> 10 m	–	–	> 4 GB	–	> 4 GB
DPhil 10	3	59 049	20.03 s	2 MB	8.04 s	1 MB	0.24 s	13 MB	0.07 s	5 MB
DPhil 100	3	5.146×10^{47}	> 10 m	–	> 10 m	–	–	> 4 GB	20.8 s	401 MB
DPhil 500	3	3.64×10^{238}	> 10 m	–	> 10 m	–	–	> 4 GB	–	> 4 GB
Kanban 10	3	1.006×10^9	52.07 s	14 MB	29.03 s	7 MB	3.81 s	72 MB	0.8 s	19 MB
Kanban 100	3	8.054×10^{11}	> 10 m	–	> 10 m	–	–	> 4 GB	–	> 4 GB
Kanban 1000	3	?	> 10 m	–	> 10 m	–	–	> 4 GB	–	> 4 GB
TokenRing 5	3	166	69.8 s	7 MB	58.42 s	2 MB	0.18 s	7 MB	0.16 s	7 MB
TokenRing 10	3	58 905	> 10 m	–	> 10 m	–	14.44 s	324 MB	14.29 s	324 MB
TokenRing 20	3	2.477×10^{10}	> 10 m	–	> 10 m	–	–	> 4 GB	–	> 4 GB

Table 6.1. Results of the full state space exploration

Model	Crit.	R?	|S|	GPOR Alone			Symb in Hybrid		GPOR in Hybrid			Total Hybrid	
				PMU	RT	DS/TL	PMU	RT	PMU	RT	DS/TL	PMU	RT
Dekker 10	1	✓	6 144	2 MB	< 1 ms	3 / 3	1 MB	0.15 s	2 MB	< 1 ms	3 / 3	2 MB	0.15 s
Dekker 20	1	✓	278 528	2 MB	0.01 s	3 / 3	3 MB	0.14 s	2 MB	0.01 s	3 / 3	3 MB	0.15 s
Dekker 50	1	✓	?	13 MB	0.5 s	3 / 3	50 MB	0.34 s	13 MB	0.46 s	3 / 3	50 MB	0.8 s
Dekker 10	2	✓	6 144	6 MB	10.13 s	6144/3	1 MB	0.17 s	2 MB	< 1 ms	3 / 3	2 MB	0.17 s
Dekker 20	2	✓	278 528	> 4 GB	–	–/–	3 MB	0.17 s	2 MB	0.1 s	3 / 3	3 MB	0.27 s
Dekker 50	2	✓	?	> 4 GB	–	–/–	50 MB	0.35 s	13 MB	0.5 s	3 / 3	50 MB	0.85 s
Dekker 10	3	×	6 144	6 MB	10.13 s	6144/–	82 MB	3.1 s	–	–	–	82 MB	3.1 s
Dekker 20	3	×	278 528	> 4 GB	–	–/–	> 4 GB	–	–	–	–	> 4 GB	–
Dekker 50	3	×	?	> 4 GB	–	–/–	> 4 GB	–	–	–	–	> 4 GB	–
Peterson 2	1	✓	20 754	11 MB	1.56 s	8994/43	80 MB	3.04 s	11 MB	1.55 s	8994 / 43	80 MB	4.59 s
Peterson 3	1	✓	3.408×10^6	2.08 GB	527.60 s	949 561 / 232	> 4 GB	–	–	–	–/–	> 4 GB	–
Peterson 4	1	✓	6.299×10^8	–	> 10 m	/	> 4 GB	–	–	–	–/–	> 4 GB	–
Peterson 2	2	✓	20 754	12 MB	5.49 s	20 754 / 54	158 MB	6.2 s	11 MB	1.56 s	8994 / 43	158 MB	7.76 s
Peterson 3	2	✓	3.408×10^6	–	> 10 m	–/–	> 4 GB	–	–	–	–/–	> 4 GB	–
Peterson 4	2	✓	6.299×10^8	–	> 10 m	–/–	> 4 GB	–	–	–	–/–	> 4 GB	–
Peterson 2	3	×	20 754	12 MB	3.81 s	20 754/–	80 MB	3.09 s		–	–	80 MB	3.09 s
Peterson 3	2	×	3.408×10^6	–	> 10 m	–/–	> 4 GB	–	–	–	–	> 4 GB	–
Peterson 4	3	×	6.299×10^8	–	> 10 m	/–	> 4 GB	–	–	–	–	> 4 GB	–
FMS 10	1	✓	2.501×10^9	2 MB	< 1 ms	11/11	8 MB	0.31 s	2 MB	< 1 ms	11 / 11	8 MB	0.31 s
FMS 100	1	✓	2.703×10^{21}	2 MB	< 1 ms	101/101	> 4 GB	–	MB	s	/	> 4 GB	–
FMS 500	1	✓	?	2 MB	0.01 s	501/501	> 4 GB	–	MB	s	/	> 4 GB	–
FMS 10	2	✓	2.501×10^9	> 4 GB	–	–/–	8 MB	0.31 s	2 MB	< 1 ms	11 / 11	8 MB	0.31 s
FMS 100	2	✓	2.703×10^{21}	> 4 GB	–	–/–	> 4 GB	–	MB	s	/	> 4 GB	–
FMS 500	2	✓	?	> 4 GB	–	–/–	> 4 GB	–	MB	s	/	> 4 GB	–
FMS 10	3	×	2.501×10^9	> 4 GB	–	–/–	17 MB	0.61 s	–	–	–	17 MB	0.61 s
FMS 100	3	×	2.703×10^{21}	> 4 GB	–	–/–	> 4 GB	–	–	–	–	> 4 GB	–
FMS 500	3	×	?	> 4 GB	–	–/–	> 4 GB	–	–	–	–	> 4 GB	–
PRISE	1	✓	$\sim 10^8$	5 MB	0.05 s	29 / 29	52 MB	1.52 s	5 MB	0.05 s	29 / 29	52 MB	2.02 s
PRISE	2	✓	$\sim 10^8$	–	> 10 m	–/–	52 MB	1.46 s	5 MB	0.05 s	29 / 29	52 MB	1.96 s
PRISE	3	×	$\sim 10^8$	–	> 10 m	/–	> 4 GB	–	–	–	–	> 4 GB	–

Table 6.2. Results of hybrid model checking

Model	Crit.	R?	|S|	GPOR Alone			Symb in Hybrid		GPOR in Hybrid			Total Hybrid	
				PMU	RT	DS/TL	PMU	RT	PMU	RT	DS/TL	PMU	RT
Kanban 10	1	✓	1.006×10^9	2 MB	< 1 ms	70 / 70	7 MB	0.3 s	2 MB	< 1 ms	70 / 70	7 MB	0.3 s
Kanban 100	1	✓	8.054×10^{11}	2 MB	0.01 s	700 / 700	> 4 GB	–	–	–	–/–	> 4 GB	–
Kanban 1000	1	✓	?	7 MB	0.05 s	7 000 / 7 000	> 4 GB	–	–	–	–/–	> 4 GB	–
Kanban 10	2	✓	1.006×10^9	> 4 GB	–	–/–	7 MB	0.31 s	2 MB	< 1 ms	70 / 70	7 MB	0.31 s
Kanban 100	2	✓	8.054×10^{11}	> 4 GB	–	–/–	> 4 GB	–	–	–	–/–	> 4 GB	–
Kanban 1000	2	✓	?	> 4 GB	–	–/–	> 4 GB	–	–	–	–/–	> 4 GB	–
Kanban 10	3	×	1.006×10^9	> 4 GB	–	–/–	18 MB	1.05 s	–	–	–	18 MB	1.05 s
Kanban 100	3	×	8.054×10^{11}	> 4 GB	–	–/–	> 4 GB	–	–	–	–	> 4 GB	s
Kanban 1000	3	×	?	> 4 GB	–	–/–	> 4 GB	–	–	–	–	> 4 GB	–
DPhil 10	1	✓	59 049	2 MB	< 1 ms	3 / 3	1 MB	0.15 s	2 MB	< 1 ms	3 / 3	2 MB	0.15 s
DPhil 100	1	✓	5.146×10^{47}	5 MB	0.01 s	3 / 3	114 MB	13.1 s	5 MB	0.01 s	3 / 3	114 MB	13.1 s
DPhil 500	1	✓	3.64×10^{238}	74 MB	0.3 s	3 / 3	> 4 GB	–	–	–	–/–	> 4 GB	–
DPhil 10	2	✓	59 049	70 MB	28.37 s	59 049 / 3	1 MB	0.15 s	2 MB	< 1 ms	3 / 3	2 MB	0.15 s
DPhil 100	2	✓	5.146×10^{47}	> 4 GB	–	–/–	114 MB	13.1 s	5 MB	0.01 s	3 / 3	114 MB	13.1 s
DPhil 500	2	✓	3.64×10^{238}	> 4 GB	–	–/–	> 4 GB	–	–	–	–/–	> 4 GB	–
DPhil 10	3	×	59 049	70 MB	28.36 s	59 049 / –	5 MB	0.21 s	–	–	–	5 MB	0.21 s
DPhil 100	3	×	5.146×10^{47}	> 4 GB	–	–/–	536 MB	26.48 s	–	–	–	536 MB	26.48 s
DPhil 500	3	×	3.64×10^{238}	> 4 GB	–	–/–	> 4 GB	–	–	–	–	> 4 GB	–
TokenRing 5	1	✓	166	2 MB	0.01 s	48 / 9	6 MB	0.2 s	2 MB	0.01 s	48 / 9	6 MB	0.2 s
TokenRing 10	1	✓	58 905	6 MB	1.00 s	1 558 / 9	71 MB	2.21 s	6 MB	1.01 s	1558 / 9	71 MB	3.32 s
TokenRing 20	1	✓	2.477×10^{10}	> 4 GB	–	–/–	> 4 GB	–	–	–	–/–	> 4 GB	–
TokenRing 5	2	✓	166	2 MB	0.05 s	166 / 9	6 MB	0.2 s	2 MB	0.01 s	48 / 9	6 MB	0.22 s
TokenRing 10	2	✓	58 905	36 MB	72.5 s	58 905 / 9	446 MB	22.84 s	6 MB	1.01 s	1558 / 9	446 MB	23.85 s
TokenRing 20	2	✓	2.477×10^{10}	> 4 GB	–	–/–	> 4 GB	–	–	–	–/–	> 4 GB	–
TokenRing 5	3	×	166	2 MB	0.05 s	166 /–	12 MB	0.35 s	–	–	–	12 MB	0.35 s
TokenRing 10	3	×	58 905	36 MB	72.3 s	58 905 /–	75 MB	2.27 s	–	–	–	75 MB	2.27 s
TokenRing 20	3	×	2.477×10^{10}	> 4 GB	–	–/–	> 4 GB	–	–	–	–	> 4 GB	–
IBMB2S565S3960	1	✓	1.551×10^{16}	4 MB	0.012 s	226 / 226	> 4 GB	–	–	–	–/–	> 4 GB	–
IBMB2S565S3960	2	✓	1.551×10^{16}	> 4 GB	–	–/–	> 4 GB	–	–	–	–/–	> 4 GB	–
IBMB2S565S3960	3	×	1.551×10^{16}	> 4 GB	–	–/–	> 4 GB	–	–	–	–	> 4 GB	–

Table 6.3. Results of hybrid model checking

6.3.3 Conclusion of the Measured Configurations

From the test-cases above it is clear that unfortunately the extreme-hard problems that are unsolved today remain unresolved, however we show some significant results to guide our future work in the field of model checking.

6.3.3.1 Evaluation of Symbolic approaches

Table 6.1 contains the measurement datas for the two different decision diagrams combined with the two different state space traversal algorithms. An interesting result is that the automated termination of the program is always caused by the runtime in case of SDDs, and by the memory usage in case of MDDs. Unfortunately, SDD were terminated many times, but it is clear from looking at the peak memory usages that it is indeed a way more compact data structure than the MDD. In [13] it was demonstrated that the SDDs (and most data structures as seen in [7]) are far more efficient with saturation. Implementing saturation for hierarchical structures are beyond the scope of this thesis, but it can be an interesting path for our future work. Because the faster runtimes of the MDD, we chose that to be the base structure behind the symbolic component of our hybrid model checker..

Judged from the measurements, the chaining loop proved to be the more efficient algorithm: whereas at models like TokenRing, Dekker or Peterson, there are little to no differences between the breadth first search and the chaining loop, but at DPhil, FMS and Kanban, the chaining loop have beaten the BFS sometimes with a decimal. That is why we chose the chaining loop to accompany the MDD in the hybrid tool.

6.3.3.2 Evaluation of Explicit Trace Generation

From Table 6.2 and 6.3 it is clear that our explicit approach (as expected) cannot handle big state-spaces and in the majority of unreachable criteria it terminates because of the memory limit. However it is important to note, that there are huge models (e.g., FMS-500 or Kanban-1000) that have so huge state-spaces that we do not even know its size, but if a goal state is reachable the trace generator very efficiently generates a short trace with minimal resource usage. Of course this exceptional results fade, if we combine our tool with the symbolic method because then our algorithm does not even get called due to the symbolic algorithm's failure, but these cases are as said, exceptional.

It is very important to notice that if the reachability criteria is hard, our trace generator tends to fail, but if a marking is reachable and a symbolic algorithm is able to find it, that boosts the trace generator greatly producing traces with the hybrid tools even when the GPOR alone failed.

Furthermore even if the GPOR alone did not fail for a harder reachable goal state, when we aid its search with additional information from the symbolic checking it produces shorter

traces more efficiently and in the majority of cases the combined runtime and memory usage of the hybrid algorithm is far below the usage of the GPOR.

It is also an important result, that there are test-cases (like Peterson) where the time limit aborted the GPOR. that means that the computation of the UP sets and layers is still very expensive, and one of our future goal, that we can optimize it further.

6.3.3.3 Evaluation of Our Hybrid approach

Based on the results shown in Table 6.2 and 6.3 our hybrid approach, however there is still much room for development, it is a success. Even though it fails for big models, these models are mostly so huge that even state-of-the-art model checkers cannot handle them.

For the majority of test-cases when the symbolic model checking is able to produce results the trace generator is so efficient it barely impacts the performance, thus we can rapidly check relatively big state-spaces and generate short traces for them efficiently. The performance of the tool is highly dependent on the symbolic model checker, with our future symbolic model checker plans further described in section 7.3 the usability of this tool could be boosted greatly.

It is also very important to note that our symbolic model checker was able to run for an industrial model (PRISE), not just for academic ones, further adding to its real-life usability.

Chapter 7

Summary

In this chapter we summarize our results and contributions, and present some of our future plans of further developing our algorithms.

7.1 Conclusion

The main conclusion of our work is that although we produced great results, there are still unanswered questions in the field of reachability checking, and there is room for further development. It is clear that our hybrid approach is a promising start for a full-fledged symbolic model checker that can also efficiently generate traces.

Nowadays more and more applications become so complex regular test-engineering starting to become harder and harder, and if we can aid the process of software and hardware testing with automatic test-generator tools that is also capable of producing a chain of events to reproduce an erroneous behavior we can develop quality products with less effort, in shorter time.

7.2 Contributions

In this section we will summarize our contributions to the different fields our work is related to.

7.2.1 Contributions to Explicit Trace Generation and Directed Model Checking

Our greatest contribution to the field of directed model checking is the new heuristic to guide the state-space traversal of the model checker. If a goal state is reachable it produces very short traces, furthermore it is able to utilize additional informations given by other model checkers to boost its own efficiency. However the heuristic of UP sets are considerably efficient, there are models where its usefulness is limited. In particular for

cyclic models it tends to generate suboptimal traces, but the cyclic property of a Petri net is a structural property so this downside of the method can be improved in the future.

7.2.2 Contributions to Symbolic Model Checking

The main contribution to the field of symbolic model checking is the implementation of set decision diagrams, for use it in model checking of hierarchical models. Its goal is to reduce the redundancy in the representation of hierarchical models even more then the common decision diagrams, and to represent state spaces with the least amount of physical memory. Our SDD implementation still needs a lot of improvements because of its speed, but there are ways of improving its performance worth considering, detailed in Section 7.3.

7.3 Future Work

In future work, we want to develop the set decision diagrams further, because their memory usage is more than promising. Implementing saturation to hierarchical structures is a part of our future plans with SDDs, because it can greatly improve their now logging speed as seen in [13]. Designing homomorphisms also seems to be a viable solution to increase the effectiveness of SDDs, because these are the original operations defined on the structure.

In the future we plan to further develop the UP set heuristic to produce even shorter traces for models now its suboptimal (like TokenRing in Table 6.3). One way of further development could be that we leave the partial order reduction approach and optimize for directed model checking. That way we would rely greatly on some other symbolic model checking tool to ensure us that the state we are searching for is reachable, but based on our measurement results it only boosts the efficiency of our algorithm. An other way of further development could be that, if we are ensured that a goal state is reachable we can neglect the model checking approach and use the UP sets and layers for only trace generation. This approach can be combined with the technique called cone of influence to reduce the size of the model and in the far future we can try to exploit the best properties of partial order reduction *within* the UP layers.

Our plan for the near future is to integrate our hybrid approach to the successor of the PetriDotNet model checking tool that is currently being developed at our department.

Acknowledgments

We would like to thank our supervisors, Vince Molnár and András Vörös for consulting us with their nonstop enthusiasm.

We also would like to thank IncQueryLabs Ltd. for their support during our summer internship.

List of Figures

Bibliography

[1] R. Alur, R.K. Brayton, T.A. Henzinger, S. Qadeer, and S.K. Rajamani. Partial-order reduction in symbolic state space exploration. In Orna Grumberg, editor, *Computer Aided Verification*, volume 1254 of *Lecture Notes in Computer Science*, pages 340–351. Springer Berlin Heidelberg, 1997.

[2] C. Baier and J.-P. Katoen. *Principles of Model Checking*. MIT Press, 2008.

[3] Randal E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Transactions on Computers*, 35:677–691, August 1986.

[4] Edmund M. Clarke, Jr., Orna Grumberg, and Doron A. Peled. *Model Checking*. MIT Press, Cambridge, MA, USA, 1999.

[5] Jean-Michel Couvreur, Denis Poitrenaud, and Yann Thierry-Mieg. libddd library, 2012.

[6] Jean-Michel Couvreur and Yann Thierry-Mieg. Hierarchical decision diagrams to exploit model structure. In Farn Wang, editor, *FORTE*, volume 3731 of *Lecture Notes in Computer Science*, pages 443–457. Springer, 2005.

[7] Dániel Darvas, András Vörös, and Tamás Bartha. Improving saturation-based bounded model checking. *Acta Cybernetica*, 21(???):???–???, 2014. Accepted, in press.

[8] Stefan Edelkamp and Shahid Jabbar. Action planning for directed model checking of petri nets. *Electron. Notes Theor. Comput. Sci.*, 149(2):3–18, February 2006.

[9] Stefan Edelkamp, Stefan Leue, and Alberto Lluch-Lafuente. Partial-order reduction and trail improvement in directed model checking. *STTT*, 6(4):277–301, 2004.

[10] Ansgar Fehnker, Stefan Edelkamp, Victor Schuppan, Dragan Bosnaki, Anton Wijs, and Husain Aljazzar. Survey on directed model checking. In Doron Peled, editor, *MoChart 2008*. Sptinger, apr 2009.

[11] Cormac Flanagan and Patrice Godefroid. Dynamic partial-order reduction for model checking software. *SIGPLAN Not.*, 40(1):110–121, January 2005.

[12] Patrice Godefroid. *Partial-Order Methods for the Verification of Concurrent Systems: An Approach to the State-Explosion Problem*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1996.

[13] Alexandre Hamez, Yann Thierry-Mieg, and Fabrice Kordon. Hierarchical set decision diagrams and automatic saturation. In KeesM. van Hee and Rüdiger Valk, editors, *Applications and Theory of Petri Nets*, volume 5062 of *Lecture Notes in Computer Science*, pages 211–230. Springer Berlin Heidelberg, 2008.

[14] Saul A. Kripke. Semantical considerations on modal logic. *Acta Philosophica Fennica*, 16(1963):83–94, 1963.

[15] Lars Michael Kristensen, K. Schmidt, and Antti Valmari. Question-guided stubborn set methods for state properties. *Formal Methods in System Design*, 29(3):215–251, 2006.

[16] Stefan Leue and Alberto Lluch Lafuente. Partial-order reduction for general state exploring algorithms. In *Model Checking Software. LNCS*, pages 271–287. Springer, 2006.

[17] Tadao Murata. Petri nets: Properties, analysis and applications. *Proceedings of the IEEE*, 77(4):541–580, April 1989.

[18] András Vörös, Dániel Darvas, Attila Jámbor, and Tamás Bartha. Advanced saturation-based model checking of well-formed coloured Petri nets. *Periodica Polytechnica, Electrical Engineering and Computer Science*, 58(1):3–13, 2014.

Appendices

Appendix A

Description of Tested Models and Criteria

In this appendix we will enumerate describe the basic properties of these models with regard to the reachability criteria we tested on them.We used the models of the Model Checking Contest (MCC) from 2013 and a behavioral model of the nuclear reactor in Paks (PRISE model).

In general, all reachability criteria we tested are in the form of the following: First we ask a reachable marking, then ask the same but we describe an other unreachable marking too to harden the task of the explicit trace generation, and then we ask an unreachable marking.

Dekker-n

This model is a variant of Dekker's algorithm for mutual exclusion, it is parametrized by the number of processes it realizes the algorithm on. The source of the model is the model checking contest 2013. in this work we examined the Dekker-10, Dekker-20, Dekker-50 models where 10, 20, 50 process' mutual exclusion were simulated.

We tested the following criteria on Dekker-n:

No.	**Question**	**Reachable?**
1	$p3/1 = 1$	✓
2	$p3/1 = 1 \wedge p3/2 = 1 \vee p3/1 = 1$	✓
3	$p3/1 = 1 \wedge p3/2 = 1$	×

FMS-n

This Petri net is extracted a benchmark used for SMART. It models a flexible manufacturing system. it is parametrized with the starting token-count on each process. In this work we simulated FMS-10, FMS-100, FMS-500

We tested the following criteria on FMS-n:

No.	Question	Reachable?
1	$P1 = 1$	✓
2	$P2 > 1000 \vee P1 = 0$	✓
3	$P2 > 1000$	×

Peterson-n

This is a model of the Peterson's algorithm for the mutual exclusion problem, in its generalized version for N processes. This algorithm is based on shared memory communication and uses a loop with N-1 iterations, each iteration is in charge of stopping one of the competing processes. In this work we simulated Peterson-2, Peterson-3, Peterson-4.

We tested the following criteria on Peterson-n:

No.	Question	Reachable?
1	$AskForSection\_0\_1 = 1 \wedge CS_2 = 1$	✓
2	$AskForSection\_0\_1 = 1 \wedge CS_2 = 1 \vee CS_1 = 1 \wedge CS2 = 1$	✓
3	$CS_1 = 1 \wedge CS2 = 1$	×

Kanban-n

This Petri net is extracted a benchmark used for SMART. It models a Kanban system. In this work we simulated Kanban-10, Kanban-100, Kanban-1000.

We tested the following criteria on Kanban-n:

No.	Question	Reachable?
1	$P1 = 1$	✓ .
2	$P1 = 1 \vee CS_1 = 1Pm1 > 1000$	✓
3	$Pm1 > 1000$	×

DPhil-n

This is the famous model that illustrates an inappropriate use of shared resources generating deadlocks. N philosophers share a table with N plates and sticks. They are thinking

and, when they need to eat, they go to the table, grab one stick from one side of their plate, then the second from the other side, then eat, and then go back thinking. we simulated Dphil-10, Dphil-100, Dphil-500.

We tested the following criteria on DPhil-n:

No.	Question	Reachable?
1	$Eat_4 = 1$	✓
2	$Eat_4 = 1 \vee Eat_5 = 1 \wedge Eat_6 = 1$	✓
3	$Eat_5 = 1 \wedge Eat_6 = 1$	×

TokenRing-n

A very complex model from the MCC 2013, parametrized with the number of processes. We simulated TokenRing-5, TokenRing-10, TokenRing-20

We tested the following criteria on TokenRing-n:

No.	Question	Reachable?
1	$State\_4\_1 = 1 \wedge State\_3\_0 = 1$	✓
2	$State\_4\_1 = 1 \wedge State\_3\_0 = 1 \vee State\_3\_1 = 1 \wedge State\_3\_0 = 1$	✓
3	$State\_3\_1 = 1 \wedge State\_3\_0 = 1$	×

IBM

A very complex industrial model from the MCC 2013.

We tested the following criteria on IBM:

No.	Question	Reachable?
1	$output = 7$	✓
2	$output = 7 \vee output > 8$	✓
3	$output > 8$	×

PRISE

This is a huge model of the nuclear reactor of Paks, modeled by András Vörös at our department.

We tested the following criteria on PRISE:

No.	Question	Reachable?
1	$output\_1false = 1$	✓
2	$output\_1false = 2 \vee output\_1false = 1$	✓
3	$output\_1false = 2$	×