MŰEGYETEM 1782

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

Guided Model Checking of Petri Nets

BACHELOR'S THESIS

Author
Dániel Élő

Supervisors
Vince Molnár
András Vörös

December 11, 2015

Contents

HALLGATÓI NYILATKOZAT

Kivonat

A modellellenőrzés egy formális verifikációs technika, melynek feladata a rendszerrel szemben támasztott követelmények matematikailag precíz ellenőrzése a rendszer viselkedésmodelljén. A modellellenőrzés előnye, hogy a követelmények megsértése esetén képes végrehajtási szekvenciával (ellenpéldával) demonstrálni a rendszer helytelen működését. Állapotelérhetőség vizsgálatakor kiemelten fontos az elérhető állapotokba vezető szekvenciák generálása, mivel sok alkalmazási területen (például modellalapú tesztelés) a fő kimenet maga az ellenpélda.

Az első modellellenőrzők a problémát a rendszer lehetséges állapotainak szisztematikus és gyakran kimerítő átvizsgálásával, vagyis az állapottér gráfjának előállításával oldották meg. Ezeket az algoritmusokat explicit modellellenőrzőknek nevezzük, mivel az állapotokat és állapotátmeneteket explicit módon tartják nyilván. Ez a megoldás azonban nagyon hamar korlátokba ütközik, mivel az állapottér már viszonylag kis modellek esetén is kezelhetetlenül nagyméretű lehet: ezt a jelenséget a szakirodalom állapottér-robbanásnak nevezi.

Konkurens rendszerekben az állapottér-robbanásnak gyakori oka az egymástól független aszinkron műveletek teljes sorrendezése, amely rengeteg, a követelmény szempontjából érdektelen köztes állapothoz vezet. A részleges rendezéses redukció az ekvivalens sorrendezéseket egyetlen reprezentatív szekvenciával helyettesítve csökkenti a bejárt állapotok számát. A módszer az explicit modellellenőrzésre épít, így az ott használt technikák többnyire alkalmazhatóak maradnak.

Az állapottér-robbanás kezelésének egy másik módja az irányított modellellenőrzés. Ez a módszer csak akkor hatékony, ha előre tudjuk, hogy a keresett állapot elérhető (ez többféleképp biztosítható, például egy szimbolikus ellenőrzéssel). A módszer kihasználja, hogy ha a célállapot elérhető akkor elegendő egy oda vezető utat felfedezni a teljes állapotér helyett, ezáltal csökkentve a memóriahasználatot. Ezt úgy valósítjuk meg, hogy hagyományos keresési algoritmusok helyett vezérelt keresést alkalmazunk és egy heurisztikával irányítjuk az állapottér-bejárást a célállapot felé, tehát nem végzünk hagyományos kimerítő keresést.

A dolgozatban bemutatunk egy irányított modellellenőrzőt, ami egy új heurisztikával irányítja az állapottér bejárást. Ezt a módszert kombináltuk a részleges rendezés redukcióval, hogy az algoritmus nem elérhető célállapotok esetén is hatékony maradjon. A dolgozat végén mérésekkel demonstráljuk az általunk készített eszköz képességeit és kiértékeljük megközelítésünk előnyeit és hátrányait.

Abstract

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, it is especially important to generate traces to reachable states, because a lot of applications (e.g., model-based testing) use them as the primary output of the model checker.

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 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 are irrelevant with regard to 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 the explicit approach, so most of the methods used there remain applicable.

Another way of handling the state-space explosion is directed model checking. The method performs well only if we know that the sought state is reachable (this can be ensured via many ways, e.g., a symbolic checking beforehand). The algorithm exploits the fact that we only have to discover a trace to the sought state and do not have to discover the whole state-space thus shrinking memory consumption. The main idea is using a heuristic function to direct the search towards the sought state, and do not perform a traditional exhaustive search.

In this work, we present a directed model checker which uses a new heuristic to guide the state-space traversal, combined with Partial Order Reduction to boost our efficiency for cases where a sought state is not reachable. The work also includes measurement results to demonstrate and evaluate our approach.

Chapter 1

Introduction

Model checking [4, 2] 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, it is especially important to generate traces to reachable states, because a lot of applications (e.g., model-based testing) use them as the primary output of the model checker.

The first model checker algorithms exhaustively enumerated every reachable state of the model, and inspected these states whether they satisfy a criteria or not. They are called *explicit model checkers*. This algorithm reached its limitations very soon, because even small models can have huge state-spaces. This phenomenon is called *state-space explosion*. In order to perform traditional explicit model checking we have to store all of the reachable states, leading to enormous memory consumption thus this method often fails for larger problems. Although model checking is a hard problem, there are algorithms that combat the state-space explosion and enable us to check models with larger state-spaces.

One way to handle the state-space explosion is *symbolic model checking* [4]. Some of these techniques usually use *decision diagrams* [3, 5] to compress the state-space, and store sets of states instead of individual states. Directly manipulating the decision diagram representations is fast and memory efficient, but the implicit encoding of states makes it impractical to handle states explicitly – affecting the ability to produce traces. In this work we decided not to complement these methods, because we focus on reachability problems, and there the main output is the trace itself.

There are explicit methods [4] to handle the state-space explosion. These methods discover a smaller, reduced state-space ins such a way that the interesting properties of the system are preserved, but excluding states that are indifferent with regard to the current model checking problem. The main issue with this approach that we do not know in advance which states are important for the current model checking problem. We cannot discover all the states, then exclude the unnecessary ones. That way we still have to store the full state space, the exact thing we want to avoid. These methods usually rely on some structural property of the model or some additional information about the model and use different, more complex graph-search algorithms that can decide a problem without traversing the whole graph. This work focuses on such techniques, mainly due to their capability of efficient trace generation.

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 are irrelevant with regard to the requirements. An explicit technique, called *partial order reduction* [11] combats the state-space explosion by substituting the equivalent interleavings with a single representative trace to reduce the number of states to discover. We will give a detailed overview of this approach in Section 2.4.

Another explicit technique handling the state-space explosion problem is *directed model checking* [9]. This method exploits the fact that if the sought state is reachable, we do not have to discover the full state-space, it is enough to traverse the trace leading to it. This method uses heuristics to guide the state-space traversal, commonly with strategies similar to A* search. However if the sought state is not reachable, there is no reduction, the full state-space has to be explored. In these cases, the approach is slower than traditional explicit techniques due to the cost of calculating the heuristics. That is why this algorithm is highly dependent on a previous verdict of reachability, potentially decided by a symbolic model checker beforehand. In this work, we assume that such a verdict is present, and we propose a new heuristic to guide the state-space traversal in Chapter 4.

The structure of the thesis is the following. Firstly, we give a brief overview of the field and the theoretical background of our work in Chapter 2. Secondly we give an overview of our explicit model checking workflow in Chapter 3. Thirdly, we introduce our guided partial order reduction algorithm in detail in Chapter 4. Fourthly we describe the implementation of our model checking tool in Chapter 5. Fifthly we present our measurement results and evaluating our tool's capabilities. Finally we draw conclusions from our work and the measurement results and propose a line of future work to further enhance our model checker in Chapter 7.

Chapter 2

Background

In this chapter we introduce some basic concepts and methods used in the literature and refer to related work which serves 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. Finally section 2.4 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 [16].

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.

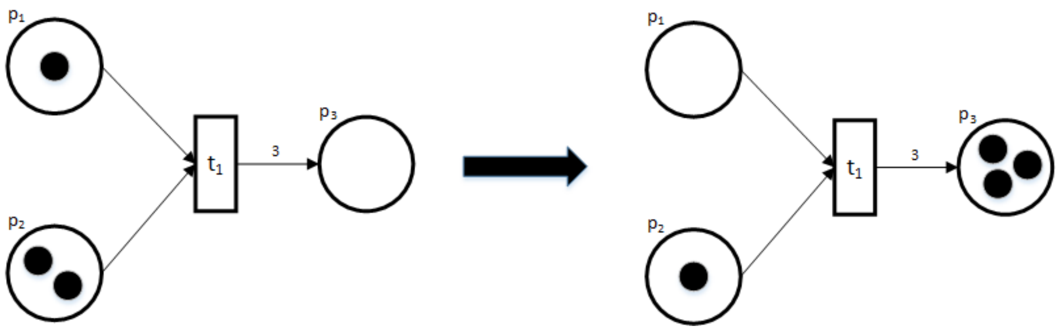

Figure 2.1. Firing, in a graphical representation of Petri net

2.2 Kripke Structures

The state-space of high-level models can be described by so-called Kripke structures [13]. 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.

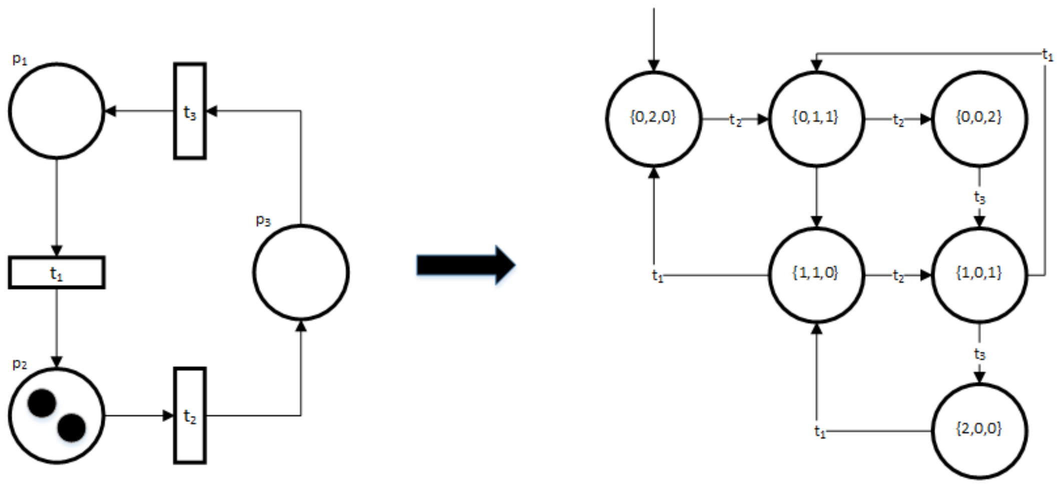

Figure 2.2. Kripke structure describing state-space of Petri net

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 2.3, 2.4 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).

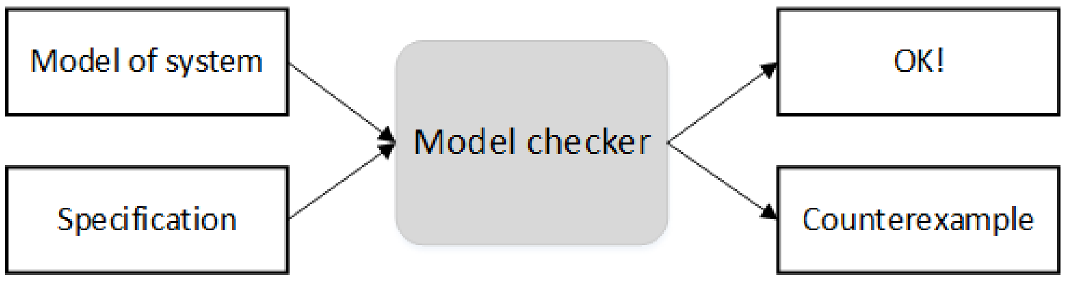

Figure 2.3. The general workflow of model checking

Due to the fact that model checking computes the state-space of high-level models, the aforementioned state-space explosion arises. Sections 2.3.2, 2.4 will introduce two ways of dealing with state-space explosion.

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.

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.

- $AP(\varphi) = \{\ell_1 \ldots \ell_n\}$ *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.
- $Less(\ell, M) = \begin{cases} true: & M(sub(\ell)) = k(\ell) \wedge op(\ell) \in \{\neq\} or \\ & M(sub(\ell)) < k(\ell) \wedge op(\ell) \in \{=, >, \geq\} \\ false: & otherwise. \end{cases}$

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

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 4

2.3.2 Directed Model Checking

Directed model checking [9] is an efficient method to combat the state-space explosion problem in case a goal state is reachable. Explicit directed model checkers [6] 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) [9]. It is important to note that such a heuristic is only useful when a goal state is actually reachable. That is why this method is usually combined with other algorithms (e.g., a symbolic model checking beforehand to decide reachability) to ensure the efficiency in cases, where no goal state is reachable [9].

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.

2.4 Partial Order Reduction

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

For an illustration, see figure 2.4. 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[10, 11, 4, 15] 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.

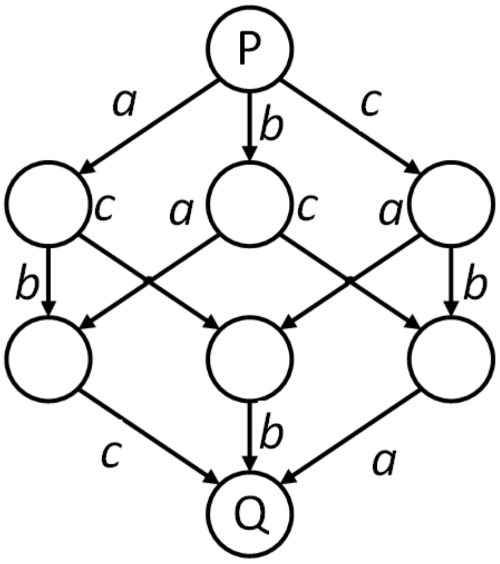

Figure 2.4. An example of indepently reorderable transitions

The idea of partial order reduction has been implemented in a number of approaches [7, 15, 1, 10, 14, 11]. In this work, we use the *stubborn sets* method, which can be considered as the state-of-the-art algorithm in this field.

2.4.1 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.

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

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.

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 3 (Stubborn sets). $STUB(s) \subseteq T$ is a stubborn set in state s *iff* the following conditions hold [15]:

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 4 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 4.2), which is a cheap over-approximation.

Chapter 3

Overview

In this chapter we give a brief overview of our workflow. Firstly we show a blackbox figure of our tool on Figure 3.1, then we will introduce each component briefly and state the reasons we applied each technique.

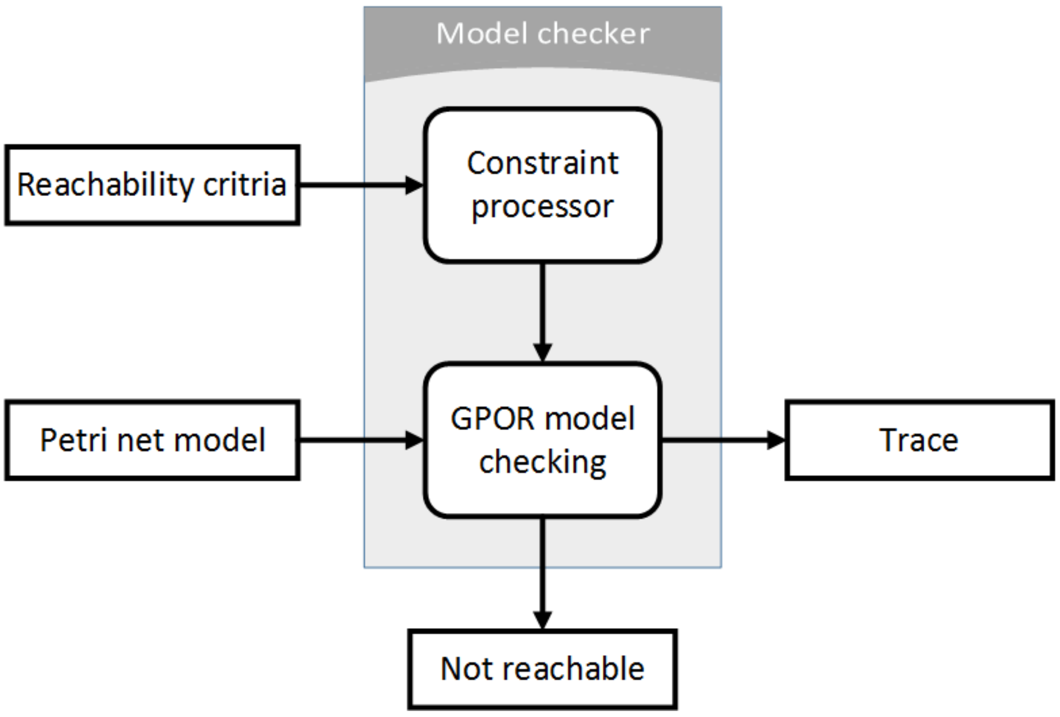

Figure 3.1. Our explicit model checking process

3.1 Constraint Processing

For the reasons detailed in Section 4.2.3 our algorithm needs to operate with a certain form of the reachability criteria – in fact we need the formula to be in disjunctive normal form and in negation normal form at the same time. This can by various transformation as described in Section 4.2. Also it is important that we process the user input in a way that our tool understands. For example $A\&B$ means the same as A *and* B or $A\&\&B$, however they are different inputs. For this preprocess task we introduced the syntax in Section 2.3.1 and we will employ a stand-alone constraint processor application detailed in Chapter 5.

3.2 The Guided Model Checker

This component performs the actual model checking given every input preprocessed. It consists of two major methods. The algorithm is basically a directed model checker with the aid of partial order reduction. The detailed description of the algorithms is shown in Chapter 4.

3.2.1 Directed Search

The heart of our solution is a directed model checking algorithm utilizing a new heuristic to guide the state-space traversal. For the reasons detailed in Chapter 1 this method only performs well in cases where a goal state is reachable, so it heavily depends on other algorithms. To see detailed description of this algorithm see Chapter 4.

3.2.2 Partial Order Reduction

Partial order reduction is a state-of-the-art explicit technique to combat the state-space explosion. We combine this approach with our directed model checker to raise its efficiency in cases it would otherwise fail. For a detailed description of this combination see Chapter 4.

Even though we expected much from this combination, evaluation of measurements revealed that we utilized partial oder reduction in a less effective way. We show a proposal for a better approach of using partial order reduction with our heuristic search in Chapter 7

3.3 Contributions

This research is still in a premature state and we are still in the experimenting phase. Our new heuristic for directed model checking is one of the main contributions of this work, but to employ it efficiently in real-life scenarios we still need to develop more algorithms that aids it in model checking. One of these methods were to run a symbolic model checker beforehand and only employ it in cases a goal state is reachable [8]. Another approach was to combine it with partial order reduction, but our strategy was not justified by our measurements. Instead, we plan to investigate another solution to combine partial order reduction and our heuristic. We present these ideas in Chapter 7.

Chapter 4

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 2.4) is employed to complement the guiding heuristics. Nevertheless, our approach tends to sacrifice performance in the unreachable case in favor of efficient trace generation. Due to the techniques our approach builds on, we call it *Guided Partial Order Reduction (GPOR)*.

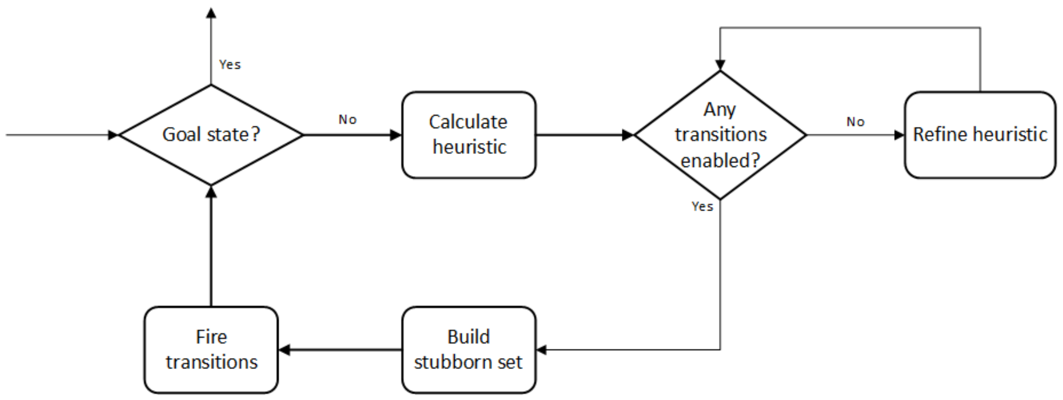

Figure 4.1. General workflow of guided partial order reduction

4.1 Guided Search in Petri Nets

Our work is based upon 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 4.2 will discuss methods to reduce other forms of φ to this form.

4.1.1 Closed-Quarters Navigation with UP Sets

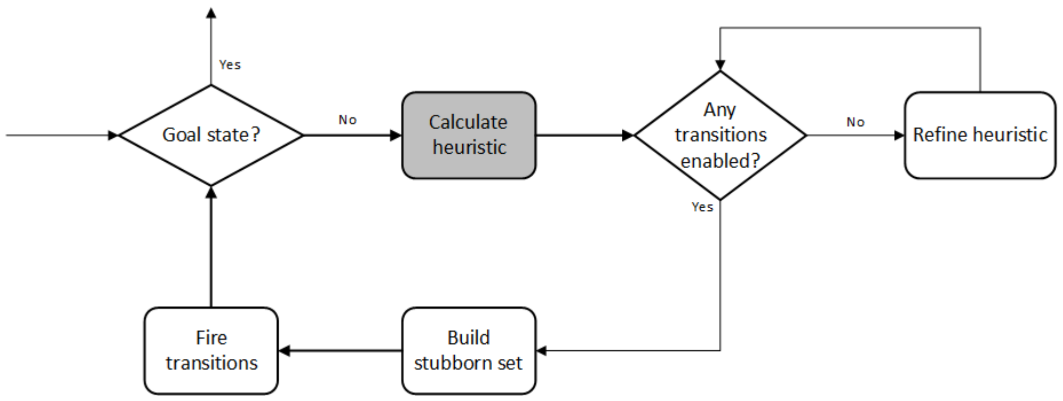

Figure 4.2. Place of UP sets in our workflow

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 [14], however we propose significant improvements.

Definition 4 (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 5 (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). 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 6 (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 6, 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 6, 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.

4.1.2 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.

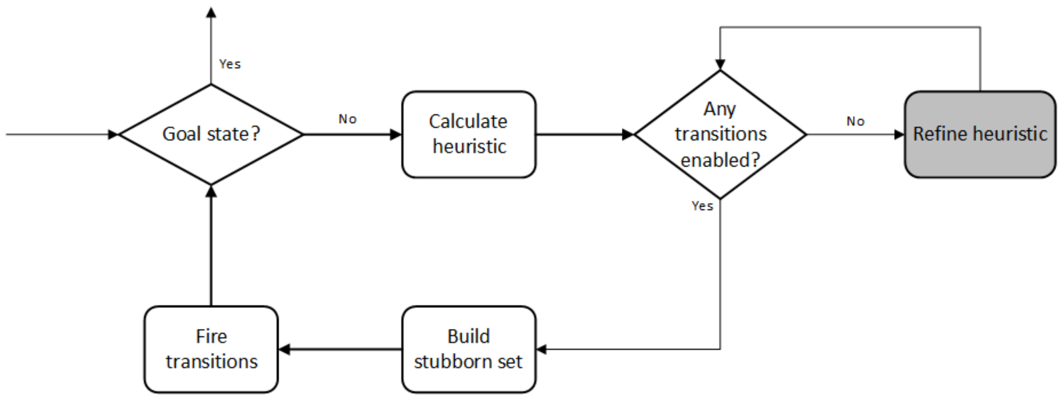

Figure 4.3. The place of UP layers in our workflow

Definition 7 (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 8 (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 9 (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 6, $UP_n(s)$ 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 10 (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 10 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 4.2.

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 7,8,9). ▪

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).

4.2 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 2.4, 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.

4.2.1 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.

UP Layer Cache

As mentioned in Section 4.1, 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.

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.

4.2.2 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 2.4 but inherently satisfying them.

Computing Stubborn Sets

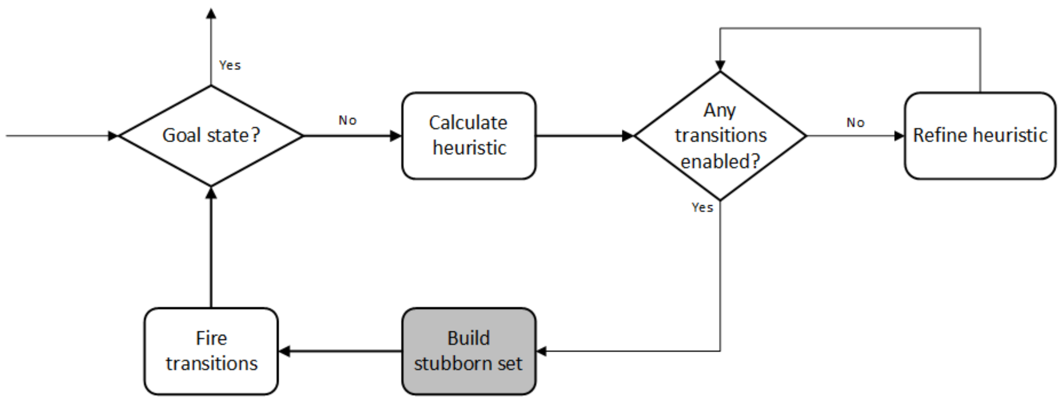

Figure 4.4. Place of partial order reduction in our workflow

We compute the stubborn sets in a way that inherently satisfies properties in Definition 3. 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 3, but it is a relatively cheap over-approximation and can be derived from the Petri net structure only.

Example:

On figure 4.5 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 4.5, t_1 cannot disable t_5 but both can disable t_2

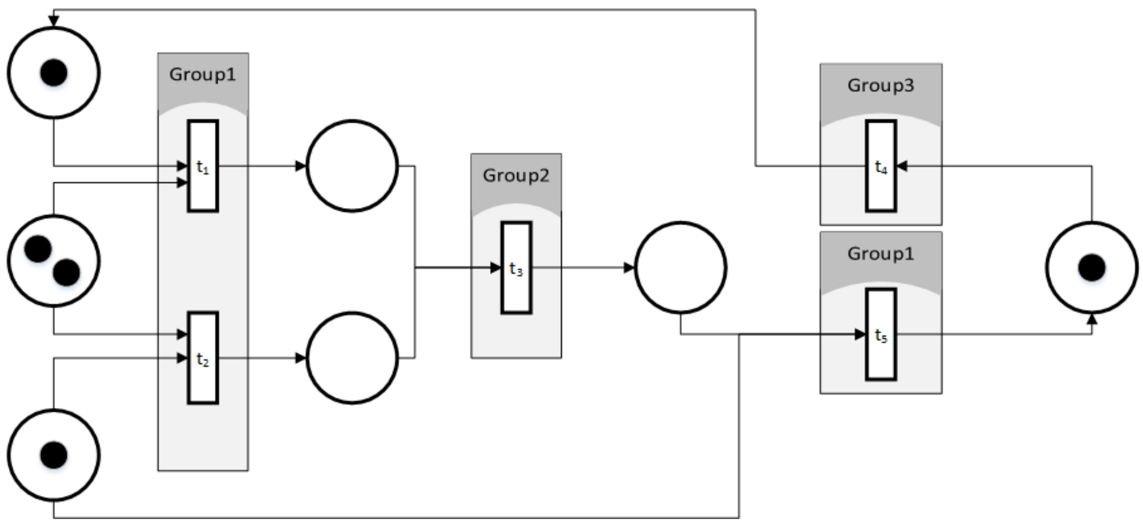

Figure 4.5. Grouping of transitions based on ability to disable each other

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

To satisfy Definition 3 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**.

Discovering the Reduced State-Space

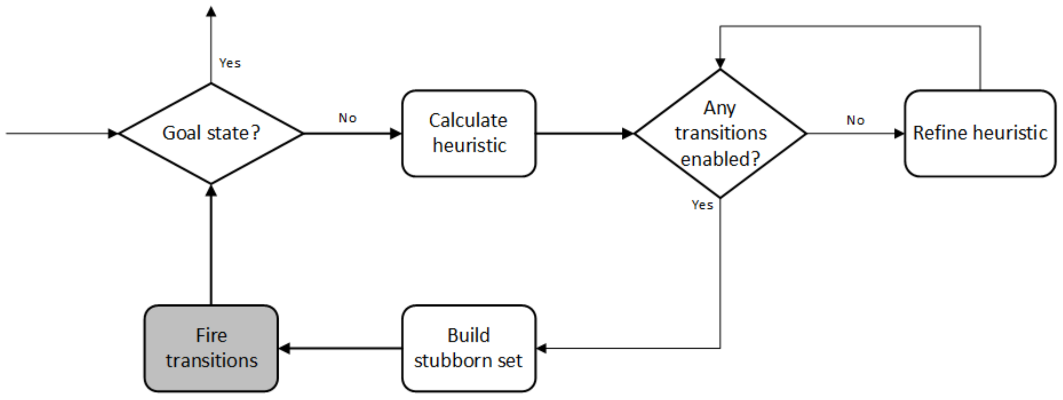

Figure 4.6. Place of state-space discovery in our workflow

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 1: Limited depth first search to produce reduced state-space

```
Input: the Petri net structure, the reachability formula ($\varphi$), and maxdepth the
       depth limit to the search
Output: trace to a goal state if reachable, an empty set if not
var Stack and ExampleStack are stacks storing states;
var DiscoveredStates is a set to store the discovered states;
var depth=0 is an integer;
Set the Petri net to the initial state; Put the initial state on Stack;
while Stack is not empty do
    var s = Stack.pop();
    try to insert s to DiscoveredStates;
    if DiscoveredStates already had s and we fired all of its UP layers then
        if s is the the top element in ExampleStack then
            ExampleStack.pop();
        end
        Stack.pop();
        continue;
    end
    else
        ExampleStack.push(s);
        ++depth;
    end
    if s satisfies $\varphi$ then
        return ExampleStack;
    end
    if depth < maxdepth then
        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
        foreach t in STUB do
            if t is not in DiscoveredStates then
                var ss = the state result of firing t from current state;
                Stack.push(ss);
            end
        end
    end
    if s is the the top element in Stack and we saw all of its stubborn sets then
        Stack.pop();
        ExampleStack.pop();
    end
end
retrurn ExampleStack;
```

Algorithm 1 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 4.1.

4.2.3 Strengths and Weaknesses of UP Layers

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 4.7.

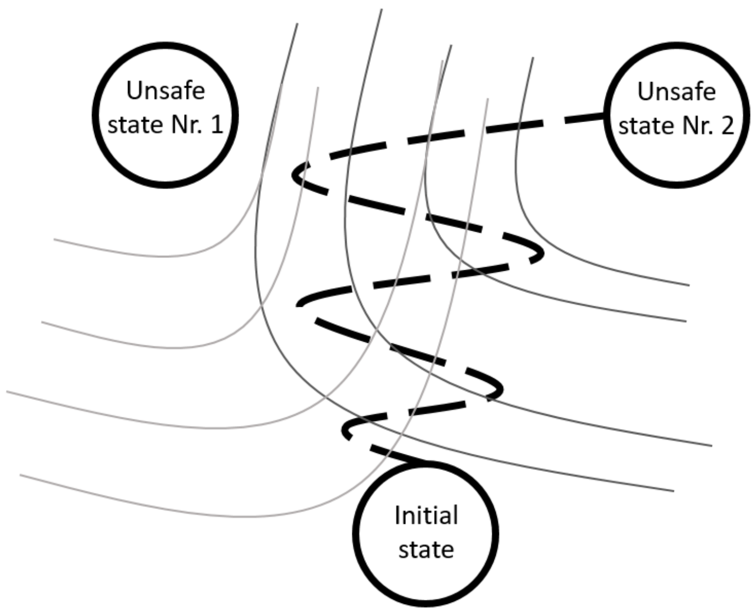

Figure 4.7. Example for two disjunct set of unsafe states and the interfering upsets

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, 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, and use our algorithm as an efficient trace generator.

Chapter 5

Implementation

In this chapter we describe the prototype in which we implemented the algorithm described in Chapter 4. Note that, although our algorithms comprise a complete model checking solution, it is optimized to be more efficient if we use it only for trace generation and perform a symbolic model checking beforehand [8].

5.1 Processing Constraints with ANTLR

In this section we briefly describe the C# program that processes the constraints and transforms them to a form that the model checker can understand. For this we use ANTLR [17] (ANother Tool for Language Recognition). With this tool we can easily describe our own grammar that suits our goals the most and we are capable of switching grammars easily, enabling us to support multiple syntaxes, facilitating integration as a trace generator into other model checkers.

In this work we evaulate Boolean expressions describing token counts on places as detailed in Section 2.3.1. For the ANTLR grammar that interprets our user input as constraints, see Appendix A.

This program gets the constraints as a command-line argument and generates a C# constraint tree on which we use a visitor to explore, transform and serialize it in a canonical form required by the model checker component. Note that this way the constraint processor and the model checker is independent,providing an easy-to-change interface above the solution.

We support various forms of logical operators(e.g., any of $\{\&, \&\&, *, and\}$ is interpreted as *logical and*)but the output is indifferent of syntactical differences, separating concerns between input processing and model checking.

It is important for the constraints to be in disjunctive normal form (DNF) and in negation normal form (NNF) at the same time. Although we could transform the expression tree into these forms, but because these requirements needed by our model checking algorithm only (and it can change in the future) we only employ the ANTLR language recognition to interpret the various user inputs and convert them to a form that our model checker can process easily and let the model checker do the DNF and NNF conversion in a preprocess step.

5.2 The Guided Model Checker

In this this section we briefly introduce the model checker prototype which implements the algorithms described in Chapter 4. The program is written in C++11. We chose C++11 because it is high-level enough that we do not have to make efforts to implement basic data structures, such as hash tables or vectors, but low-level enough to perform low-level memory operations (which we use often) efficiently.

5.2.1 Architecture

A simplified UML class diagram of the model checker tool is presented on Figure 5.1.

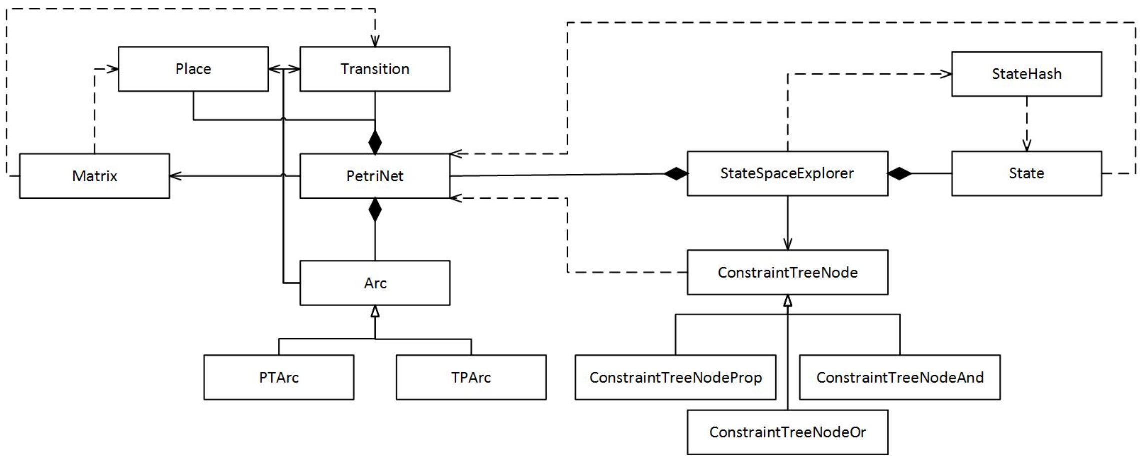

Figure 5.1. Simplified UML class diagram of the guided Mode checker tool

Brief description of each class in Figure 5.1:

- **State:** A class representation of one state of a Petri net system. We use the token counts on each place to describe a state as introduced in Section 2.1.
- **StateHash:** We store the discovered states of the model in a hash-table (std::unordered_set) to gain performance for big state-spaces, for that we need a hash function, this is a wrapper class for this hash function.
- **ConstraintTreeNode:** A base class for a reachability expression tree further described in Section 2.3.1. Note that there is no class implementing negation, because NNF always pushes them next to atomic propositions, which in turn can include negation by changing their inner operators (see more in Section 4.2.1).

- **ConstraintTreeNodeOr:** A child class for storing two expression nodes connected by a logical or.
- **ConstraintTreeNodeAnd:** A child class for storing two expression nodes connected by a logical and.
- **ConstraintTreeNodeProp:** A child class for representing an atomic proposition in the expression. See more in Section 2.3.1.
- **PetriNet:** A class representing of a Petri net described in Section 2.1.
- **Transition:** A class representing of a transition in a Petri net, see more in Section 2.1.
- **Place:** A class representing of a place in a Petri net, see more in Section 2.1.
- **Arc:** A base class for representing an arc in a Petri net, connecting a place and a transition. See more in Section 2.1.
- **PTArc:** A base class for representing a place-transition arc in a Petri net.
- **TPArc:** A base class for representing a transition-place arc in a Petri net.
- **Matrix:** For efficiency reasons we store a matrix representation in a Petri net. This matrix is actually the W^* structure for every place and transition described in Section 2.1.

5.2.2 Normal forms

The Boolean language recognition is performed by a separate program (see Section 5.1), so we can assume that they are well formed and only have to check the validity of place names. Then because of the properties of our new heuristic described in Chapter 4 we transform this expression into negation normal form and disjunctive normal form via Boolean rules before we perform the model checking.

5.2.3 The state-space traversal

The state-space exploration is performed with Algorithm 1 described in Section 4.2. During the exploration we monitor the memory usage and runtime of the application and terminate the model checker if it exceeds 10 minutes runtime or 4 GB memory consumption, to aid the benchmark testing. The detailed measurement setup is described in Section 6.1.

Chapter 6

Evaluation

In this chapter we provide detailed measurement data of our algorithm and compare our tool with another tool currently used in the field of model checking. The data we show in this chapter serves as a guideline for our future work, shows what we have reached so far and highlights the issues where we can further improve our approach.

6.1 Measurements

In this section, first we describe the measurement process, then we show detailed results demonstrating the performance of our tool. The analysis of the results is presented in Section 6.2.

We used selected models of the Model Checking Contest[1] as benchmarks to evaluate our work. The detailed description of the models and reachability criteria can be found in Appendix B.

6.1.1 Process of Measurements

We performed the measurements on the following system: Intel i7-3610-QM CPU @ 2.30 GHz, 6 GB DDR3-1333 MHz RAM, Microsoft Windows 10 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. We performed every measurement setup 3 times and display the median of the results. The results of our measurements is shown in Tables 6.1 and 6.2 in the columns of *GPOR*.

For a comparison, we measured the performance of a state-of-the-art tool, the ITS model checker[12] developed by *Laboratorie d'Informatique de Paris 6*[2] on the same configuration. Note that this is a symbolic model checker that uses *hierachical set decision diagrams*

[1]http://mcc.lip6.fr/2013/
[2]http://ddd.lip6.fr/index.php

to explore the state-space. Although it is a symbolic model checker it is capable of trace generation. This is one of the fastest state-of-the-art tool that we know of that is why we chose it to compare our model checker to. The results of our ITS measurements is shown in Tables 6.1 and 6.2 in the columns of *ITS*

6.1.2 Results of Our Approach

In this section we present our measurement results, 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 identifier of reachability criteria we asked from our model checkers. The description of identifiers can be found in appendix B
- The */S/* column shows the size of the state-space of the model, if it is unknown, it contains the "?" symbol.
- The *RT* columns shows the total runtime of the processes.
- The *PMU* columns shows the peak memory usage (working set) of the processes.
- The *TL/DS* columns shows the size of the discovered state-space (DS) in comparison to the length of trace generated (number of states on the trace including the initial and final state) (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.
- "–" means that the cell cannot contain valuable data due to termination or the data is not interpreted in the current context (e.g., trace length (TL) is not interpreted in unreachable cases)
- Asterisks in Table 6.1 means that ITS produced a trace, but it declares it "imprecise" e.g., the validity of the trace is not verified.

Model	Crit.	R?	S	PMU		RT		TL/DS	
				GPOR	ITS	GPOR	ITS	GPOR	ITS
Dekker-10	1	✓	6 144	2 MB	7 MB	< 1 ms	0.17 s	3/3	1/6 144
Dekker-20	1	✓	1.153×10^{7}	3 MB	39 MB	0.02 s	2.14 s	3/3	$1/1.1 \times 10^{7}$
Dekker-50	1	✓	?	3 MB	115 MB	0.02 s	12.64 s	3/3	$1/\ 4.85 \times 9^{14}$
FMS-10	1	✓	2.501×10^{9}	2 MB	68 MB	0.01 s	3.41 s	10/10	$10/2.5 \times 10^{6}$
FMS-100	1	✓	2.703×10^{21}	2 MB	–	0.01 s	> 10 m	100/100	–/–
FMS-500	1	✓	?	2 MB	–	0.01 s	> 10 m	500/500	–/–
Peterson-2	1	✓	20 754	11 MB	12 MB	1.6 s	0.6 s	43/8 923	36/20 754
Peterson-3	1	✓	3.408×10^{6}	1 611 MB	66 MB	383.3 s	3.66 s	71/732 013	$61/3.4 \times 10^{6}$
Peterson-4	1	✓	6.299×10^{8}	–	575 MB	> 10 m	50.19 s	–/–	$96/6.3 \times 10^{8}$
Kanban-10	1	✓	$1,006 \times 10^{9}$	2 MB	177 MB	< 1 ms	11.95 s	63/63	$55/10^{9}$
Kanban-100	1	✓	$1,726 \times 10^{19}$	2 MB	–	0.01 s	> 10 m	693/693	–/–
Kanban-1000	1	✓	?	7 MB	–	0.07 s	> 10 m	6 993/6 993	–/–
DPhil-10	1	✓	59 049	2 MB	7 MB	< 1 ms	0.08 s	3/3	3/59 049
DPhil-100	1	✓	$5,154 \times 10^{47}$	5 MB	14 MB	0.02 s	0.58 s	3/3	$3/5,1 \times 10^{47}$
DPhil-500	1	✓	3.64×10^{238}	74 MB	50 MB	0.40 s	5.01 s	3/3	$3/3.6 \times 10^{238}$
TokenRing-5	1	✓	166	2 MB	7 MB*	0.01 s	0.23 s*	9/48	7/166*
TokenRing-10	1	✓	58 905	6 MB	23 MB*	1.08 s	1.97 s*	9/1 558	7/58 905*
IBM B2S565S3960	1	✓	1.551×10^{16}	4 MB	–	0.01 s	> 10 m	226/226	–/–

Table 6.1. Measurement results of reachable cases

Model	Crit.	R?	S	PMU		RT		TL/DS	
				GPOR	**ITS**	**GPOR**	**ITS**	**GPOR**	**ITS**
Dekker-10	2	$\times$	6 144	6 MB	6 MB	10.77 s	0.23 s	–/6 144	–/6 144
Dekker-20	2	$\times$	1.153×10^7	> 4 GB	14 MB	–	2.53 s	–/–	–/1.1×10^7
Dekker-50	2	$\times$	?	> 4 GB	50 MB	–	13.40 s	–/–	–/2.9×10^{16}
FMS-10	2	$\times$	2.501×10^9	> 4 GB	9 MB	–	0.27 s	–/–	–/2.5×10^9
FMS-100	2	$\times$	2.703×10^{21}	> 4 GB	–	–	> 10 m	–/–	–/–
0.27 s FMS-500	2	$\times$	?	> 4 GB	–	–	10 m	–/–	–/–
Peterson-2	2	$\times$	20 754	12 MB	7 MB	3.00 s	0.51 s	–/20 754	–/20 754
Peterson-3	2	$\times$	3.408×10^6	–	22 MB	> 10 m	2.58 s	–/–	–/3.4×10^6
Peterson-4	2	$\times$	6.299×10^8	–	213 MB	> 10 m	36.08 s	–/–	–/6.29×10^8
Kanban-10	2	$\times$	$1,006 \times 10^9$	> 4 GB	6 MB	–	0.6 s	–/–	–/10^9
Kanban-100	2	$\times$	$1,726 \times 10^{19}$	> 4 GB	–	–	> 10 m	–/–	–/–
Kanban-1000	2	$\times$	?	> 4 GB	–	–	> 10 m	–/–	–/–
DPhil-10	2	$\times$	59 049	71 MB	5 MB	28.90 s	0.18 s	–/59 049	–/59 049
DPhil-100	2	$\times$	$5,154 \times 10^{47}$	> 4 GB	8 MB	–	215.83 s	–/–	–/5×10^{47}
DPhil-500	2	$\times$	3.64×10^{238}	> 4 GB	–	–	> 10 m	–/–	–/–
TokenRing-5	2	$\times$	166	2 MB	6 MB	0.05 s	0.10 s	–/166	–/166
TokenRing-10	2	$\times$	58 905	37 MB	11 MB	71.25 s	1.34 s	–/58 905	–/58 905
IBM B2S565S3960	2	$\times$	1.551×10^{16}	> 4 GB	—	–	> 10 m	–/–	–/–

Table 6.2. Measurement results of unreachable cases

6.2 Evaluation of Measurement Results

In this section we analyze the data we have collected from the measurements to highlight the current strengths and issues of the algorithm as well as we compare the results of the ITS tool with ours.

6.2.1 Strengths of Our Algorithm

The greatest strength of our algorithm is undoubtedly the capability of producing short traces using very little memory and processor-time, making our tool a very good trace generator for reachable cases. Note that we are able to produce short traces within a fraction of a second with only a few MB of memory usage for such huge models whose state-space size is unknown. These results are particularly showing the results of Kanban-1000, FMS-500, Dekker-50 in Table 6.1. The ITS model checker, in these cases, performed far worse in most reachable cases. Note, that the ITS produces shorter traces in many cases but our trace lengths are not very different.

6.2.2 Room for Development

We have showed that our tool is far better for reachable cases than ITS, but in unreachable cases the tables have turned. Our explicit approach could not compete with the symbolic ITS model checker and performed far worse in these cases. This is shown in Table 6.2.

It is especially important to notice that in unreachable cases our model checker had to discover the full state-space rendering our partial order reduction useless. This is because of the incompatibility of our directed model checker and partial order reduction. In the process of optimizing the heuristic search and preserving its soundness we lost the ability to reduce the state-space. In Section 7.2 we will show a new way of combining partial order reduction with our algorithm currently under development. We think this approach will preserve the much desired reductive properties of partial order reduction and still preserve the strengths of our directed model checking algorithm.

It is also important that in the Peterson model, our algorithm terminated due to timeout not memory limit. This means that the calculation of the heuristic can be further improved, because there are some models (In our measurements the Peterson model) that are so complex in structure that our heuristic calculation is too complex and consumes too much time.

Chapter 7

Conclusion

In this section we summarize the results of our research and give a brief overview of our future development plans, particularly focusing on the weaknesses of our approach focusing on partial order reduction.

7.1 Effect of Partial Order Reduction

As the measurement results and analysis shows in Chapter 6 in the process of optimizing and correcting our new heuristic for guiding the state-space exploration we lost the efficiency of the partial order reduction. This is because of the state-space exploration algorithm shown in Algorithm 1.

When we trace back to a node after processing all the members of our stubborn set, we build another stubborn set around the next best UP layer in the system. This causes that in most cases, sooner or later we discover all the states of the model. This way we gain mathematical correctness and soundness for our heuristic search and in cases of reachable problems we gain efficiency, but in cases of a goal state is not reachable we still have to explore the whole state-space leading to limited usage of our algorithm as a stand–alone tool. In previous works we combined our algorithm with symbolic model checkers and only used it for trace generation, but there is still room for development even if we only use the tool for trace generation. This way the stubborn sets hinders us in trace generation because of the larger size of states to explore states and we only need the directed model checker. However we can also combine our algorithm with partial order reduction inside our heuristic gaining the advantage of not exploring the full state-space. The next section presents our idea of employing the stubborn sets method differently.

7.2 Future Plans

Figure 7.1 shows a way to employ partial order reduction *inside* an UP layer, thus discovering only a reduced set of neighbors from every state, leading to a reduced state-space.

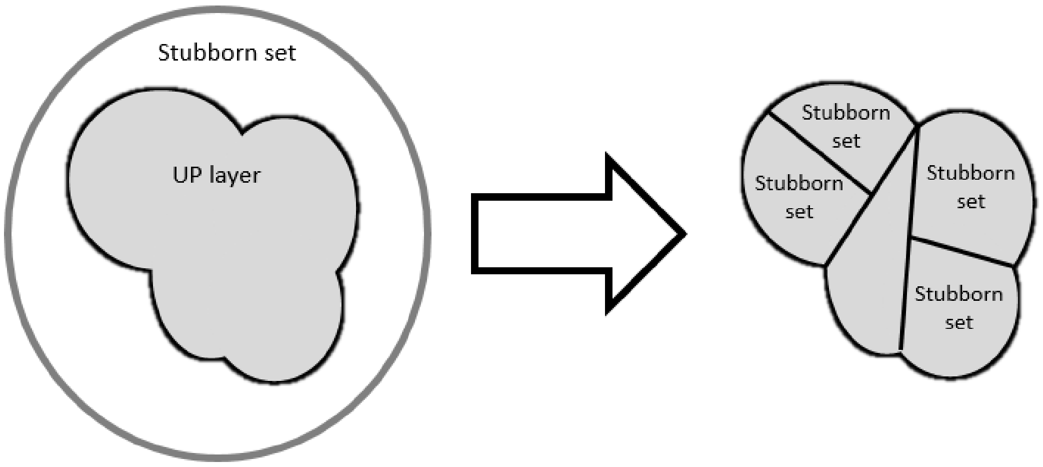

Figure 7.1. Plans for combination of partial order reduction and UP layers

In this scenario we would prioritize transitions by a heuristic (e.g., the closer UP layer the better, or the more atomic propositions it satisfies the better, or a combination of the above). And we try to fire the stubborn set containing the "best" transitions. Firing a stubborn set does not disable the rest of the transitions, that way if we did not fire a "good" transition that would help reaching a goal state we can fire it later, it will be enabled. Of course we have to look out not to fire stubborn sets representing a circle to avoid unnecessarily long traces. Our priority in the future is to implement this approach and evaluate its efficiency and performance compared to this work and other tools currently used in the field of model checking.

Furthermore we will try to implement a transition ordering heuristic, not just to be able to prioritize some stubborn sets over others, but to try to produce *shortest* traces, which is a valuable output in the field of model-based testing.

7.3 Summary

All in all we developed and implemented a very efficient explicit trace generator, which uses a new kind of heuristic and produces very short traces using minimal amounts of resources. Its efficiency is a result of sacrificing performance in unreachable cases thus reducing its usability as a stand-alone model checker tool. In the future, we hope to gain further advantage from a new approach of combining partial order reduction with our directed model checking algorithm. Nevertheless, the proposed solution already fills the gap of trace generation in the tools developed by our research group.

Acknowledgments

I would like to thank *IncQueryLabs Ltd.* for their support during my summer internship.

List of Figures

List of Tables

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 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.

[6] 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.

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

[8] Dániel Élő and Adrián Soltész. Symbolic model checking and trace generation by guided search, 2015. Student's Scientific Conference, Institution level,1st prize.

[9] 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.

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

[11] 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.

[12] 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.

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

[14] 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.

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

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

[17] Terence Parr. *The Definitive ANTLR 4 Reference*. Pragmatic Bookshelf, 2nd edition, 2013.

Appendices

Appendix A

ANTLR Grammar

The ANTLR4 grammar used to process constraints is the following:

```
grammar constraint;

/*
 * Parser Rules
 */
constraint
    : LPAR Expression=constraint RPAR                                           #ParenthesizedBoolean
    | Proposition=atomicProposition                                             #Proposition
    | NotOp Operand=constraint                                                  #Not
    | LeftOperand=constraint AndOp RightOperand=constraint                      #And
    | LeftOperand=constraint OrOp RightOperand=constraint                       #Or
    ;

atomicProposition
    : PlaceID RelationOp TokenCount
    ;

/*
 * Lexer Rules
 */

OrOp : ('|' | '||' | '+' | 'or') ;
AndOp : ('&' | '&&' | '*' | 'and') ;
NotOp: ('!' | '~' | 'not') ;

RelationOp : ('<' | '>' | '<=' | '>=' | '=' | '==' | '!=' | '<>');
TokenCount : ('0' | ([1-9] [0-9]*) );
PlaceID : ([-_\#/\\] | [0-9] | [a-z] | [A-Z] | '.')+ ;

LPAR : '(' ;
RPAR : ')' ;

WS : [ \t\r\n]+ -> skip;

Unknown
    : .
    ;
```

Appendix B

Description of Models

In this appendix we will enumerate and describe the basic properties of the tested models and show the reachability criteria we tested on them. We used selected benchmark models of the Model Checking Contest (MCC) from 2013.

In general all test cases are following the same pattern, for each model we ask our tool to find a reachable marking and an unreachable one.

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	$p3/1 = 1 \wedge p3/2 = 1$	×

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	$P2 > 1000$	×

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	$AskForSection\_0\_1 = 1 \wedge C\_S2 = 1$	✓
2	$CS\_1 = 1 \wedge CS\_2 = 1$	×

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	$Pm1 > 1000$	×

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	$Eat\_5 = 1 \wedge Eat\_6 = 1$	×

TokenRing-n

A model from the MCC 2013, parametrized with the number of processes it contains. We simulated TokenRing-5, TokenRing-10.

We tested the following criteria on TokenRing-n:

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

IBM

A complex industrial business model provided by IBM from the MCC 2013. Although all the place and transition names are obfuscated, we know that this Petri has a workflow structure (unique source and sink place).

We tested the following criteria on IBM:

No.	**Question**	**Reachable?**
1	$output = 7$	✓
2	$output > 8$	×