BOUNDDED MODEL CHECKING OF 1-BOUNDDED PETRI NETS USING A SATISFIABILITY BASED PLANNER

Matti Koskimies
## CONTENTS

1 Introduction 1

2 The Propositional Satisfiability Problem 1

3 Model Checking 2
   3.1 Approaches to Model Checking 2
   3.2 Bounded Model Checking 3
      Bounded Model Checking Problem as a Propositional Formula 3

4 Petri Nets 5
   4.1 Definition of Petri Nets 5
   4.2 On Behavioral Properties and Analysis of Petri Nets 7

5 Automated Planning 8
   5.1 Petri Net Deadlock Detection as a Planning Problem 9
   5.2 Classical Planning as Satisfiability 12

6 Experiments with Satplanner as a Bounded Model Checker 12
   6.1 On Performance Metrics and Measurements 13
   6.2 Test Equipment and Configurations 14

7 Results 15
   Discussion on the Results 16

8 Conclusions 18

A PDDL Presentation of the Running Example 19

References 20
1 INTRODUCTION

Model checking is an approach to verification of digital circuits and software systems. Within the software field model checking has proven to be a successful technology especially with embedded and safety-critical systems. Also distributed and concurrent systems are a promising application area for model checking, since with such systems it is often very difficult to replicate an observed faulty behavior and this often makes the testing process with conventional methods troublesome.

The biggest challenge for the model checking approach is the state explosion problem which informally refers to the exponential growth of number of possible states of a system to the size of the system description. This study is part of research whose goal is to produce more efficient methods for model checking and tackling of the state explosion problem. In this study we are focusing on bounded model checking in which informally the goal is to find counterexamples that violate the specified behavior of a system.

One widely studied approach to bounded model checking is satisfiability solving in which bounded model checking tasks are translated into satisfiability problems and then solved with a tool designed for this purpose. In this study we have taken another approach by first translating model checking tasks into another form, namely, into planning problems, and then solving them with a planner. The translation of different kinds of bounded model checking tasks into planning problems would be an interesting subject already in itself. Our goal, however, was to study the applicability of a certain specific planner called Satplanner to bounded model checking. The reason for this is that Satplanner, developed by Jussi Rintanen and Markus Büttner [11], has shown to be highly efficient system and it is also based on satisfiability solving. Therefore, we wanted to test how well Satplanner suits for bounded model checking tasks in order to gain insight whether the ideas used in Satplanner might also be fruitful with bounded model checkers based on satisfiability solving. The results show that to some extent this really seems to be the case.

2 THE PROPOSITIONAL SATISFIABILITY PROBLEM

The propositional satisfiability problem is one of the central concepts related to this study. Quite often the problem is referred to as the Boolean satisfiability problem, or simply as to SAT. The reason for its significance is, that with many computationally hard problems, the problem can often be transformed to a SAT problem and solved with a SAT-solver. This approach is useful because it is often easier and faster to think of the transformation than to design a special purpose algorithm for solving the problem directly. Moreover, since there are many general purpose SAT-solvers that have been highly tuned, it can be often quite difficult to design a special purpose algorithm that would outperform a high quality SAT-solver. Examples of highly efficient and well recognized SAT-solvers are zChaff [17] and MiniSAT [7]. In this study a satisfiability solver called Siege [21] is used.
Description of SAT. In the context of propositional logic, an atomic proposition is a statement that has a truth value assigned to it, i.e. the proposition is either true or false. Logical connectives like ¬ (negation, logical not), ∧ (conjunction, logical and) and ∨ (disjunction, logical or) can be used to form compound statements from the atomic propositions. A subset \( A \) of a set \( P \) of all atomic propositions is called a truth assignment if and only if all elements of \( A \) have value true and all elements of set \( P \setminus A \) have value false (i.e. truth assignments can be seen as possible divisions of the atomic propositions into two distinct sets based on truth value.) A given (compound) statement is satisfiable, if and only if there exists at least one truth assignment for which the truth value of the compound statement is true. If there does not exist such a truth assignment, the statement is unsatisfiable.

The SAT-solvers usually expect the propositional statement to be given in conjunctive normal form (CNF). This means that the statement is a conjunction of clauses, which in turn are disjunctions of atomic propositions or negations of atomic propositions (i.e., literals).

3 MODEL CHECKING

3.1 Approaches to Model Checking

The first model checking algorithms constructed the entire reachability graph of the system model and used graph traversal algorithms to check all the possible states of it in order to verify that the system fulfills the expected behavior [5]. This approach becomes infeasible with systems whose order of magnitude lays around a few tens of millions of states. Considering that a system with twenty four Boolean variables has about sixteen million different states, it is clear that this efficiency is not nearly sufficient for commercial model checker products.

One way to boost the graph traversal is symbolic model checking in which the idea is to encode the transition relation and the reachability graph of a system model as a Boolean function and to represent this function with BDDs (Binary Decision Diagrams) [3]. BDDs provide an efficient way of representing Boolean functions in a canonical form. The advantage of this approach in the context of model checking, is that BDDs make it possible to store sets of states and to compute transitions between sets of states rather than individual states. This method raises the processing capacity of a model checker to around \(10^{20}\) states which corresponds to a system with hundreds of variables [4]. This efficiency level is still far from satisfying all the needs, but nevertheless, there are working model checkers based on the BDD-technique. One of them is the symbolic model verifier (SMV) developed at the Carnegie Mellon University [14].

After BDD-based algorithms, research has suggested applying propositional satisfiability procedures to symbolic model checking. With these methods the system and the formal specifications of requirements are encoded as a SAT problem, which can then be solved with a SAT solver. The SAT based approaches have proven to be sometimes much more efficient com-
pared to BDD-based methods and they are capable of handling systems with thousands of variables [2]. With the use of the SAT techniques, the worst shortcomings of the BDD-based algorithms can also be avoided. These include exponential worst case memory consumption of BDDs and the fact that the efficiency of BDD-based algorithms is highly dependent of the variable reordering, which means that the BDD-based algorithms might need time consuming automatic dynamical reordering of variables or manual intervention by the user.

The first SAT-based approach to model checking was the bounded model checking, which will be covered more closely in the next section.

3.2 Bounded Model Checking

The objective of bounded model checking (BMC) is to search for counterexamples violating the specified behavior of a system. In this context the counterexamples are executions of a system with length bounded by a given integer $k$. By an execution of a system with length $k$ we mean a sequence of states $s_0, s_1, \ldots, s_k$ that the system could possibly go through. The idea in bounded model checking is to first verify a system for some initial bound (often a initial bound $k = 0$ might be chosen). If any counterexamples are not found, then the initial bound $k$ is increased and the search is repeated. This process is iterated until a counterexample is found or the problem has become too complex to be solved.

The approach of bounded model checking is convenient for many reasons [2]. First of all, as was mentioned earlier, the goal in software verification is often to find as many flaws as possible rather than to verify the complete correctness of the system. This is especially true in the beginning of the design process. Because bounded model checking searches counterexamples of bound lengths in increasing order, it often finds short ones very fast. This follows from the fact that in practice the time required to find a counterexample is usually observed to grow highly non-linearly to the path length. Also bounded model checking finds automatically the shortest counterexamples. This is a very useful feature, since besides giving the information of unwanted behavior, the counterexample is also used to track the source of a problem, so that it can be fixed.

Bounded Model Checking Problem as a Propositional Formula

In the following it is shown how bounded model checking of a state transition system can be formulated as a propositional formula. With a state transition system we mean a system accordant with Definition 1.

**Definition 1.** A state transition system is a 3-tuple $\langle S, T, I \rangle$, where $S$ is a finite set containing all the possible states of the system. Each state $s_i \in S$ is a vector $(x_1^i, x_2^i, \ldots, x_n^i)$ of the variables $x_j$ of the system at time step $i$. The function $T : S \times S \rightarrow \{\text{true, false}\}$ is called a transition relation and it specifies the possible transitions between the states. For states $s_i, s_{i+1} \in S$ of the system $T(s_i, s_{i+1})$ evaluates to $\text{true}$ if and only if transition $s_i \rightarrow s_{i+1}$ is possible at time step $i$. The characteristic function $I : S \rightarrow \{\text{true, false}\}$ specifies the possible initial states of the system. For each state $s \in S$ the function $I(s)$ evaluates to $\text{true}$ if and only if $s$ is a possible initial state of the system.
system.

Now, suppose we are given:

- a state transition system $M = \langle S, T, I \rangle$,
- an invariant property $S$ stating that the value of the proposition $P(s)$ should be true in every state of the system, and
- a bound $k$ for the length $l$ of the executions (for each execution it has to hold that $l \leq k$).

Now we want to verify that the following requirement holds: while starting from any of the states specified by the characteristic function $I(s)$, the system $M$ will not end up in a bad state $s_x$ where the property $P$ does not hold, along any execution with its length $l$ bounded by $k$ as specified above.

In order to verify the validity of this statement by means of satisfiability checking, we will create a propositional formula stating that the negation of the previous statement is true. That is, the formula should evaluate to true, if and only if there exists an execution of length $k$ and a state within this execution that does not satisfy the invariant property $S$. If this propositional formula can be shown to be satisfiable with a SAT-solver, the corresponding truth assignment of the variables gives us a counterexample violating the invariant property. On the other hand, if the propositional formula is shown to be unsatisfiable the correct behavior of the system is verified for executions with length bounded by $k$.

The propositional formula corresponding to the verification task in question consists of a conjunction of two parts: a compound statement $\|M\|_k$ encoding an execution with length bounded by $k$ and a proposition $\|\neg S\|_k$ stating that the invariant property holds for this particular execution. These statements are given as:

$\|M\|_k := I(s_0) \land \bigwedge_{i=0}^{k-1} T(s_i, s_{i+1})$

$\|\neg S\|_k := \bigvee_{i=0}^{k} \neg P(s_i)$

Hence, the final formula $\|M, \neg S\|_k$ is given as:

$\|M, \neg S\|_k := I(s_0) \land \bigwedge_{i=0}^{k-1} T(s_i, s_{i+1}) \land \bigvee_{i=0}^{k} \neg P(s_i)$
4 PETRI NETS

In this section an overview on Petri nets and some specific issues relevant to this work are covered. Aside from the running example the text follows closely a paper on Petri nets written by Tadao Murata [18].

Petri nets are a graphical and mathematical tool for modelling and analyzing of information systems. They are especially well suited for systems with concurrent, asynchronous, parallel, nondeterministic, and/or stochastic aspects. Besides Petri nets, there is a wide variety of different languages and tools designed for modelling purposes. These tools and methods may be categorized for example on the target domain they are designed for (e.g. telecommunications protocol design or logistics chain), the target group of users (business people, engineers or scientists) and the type of purpose of usage (analysis or modelling). The strength of Petri nets - and one of the reasons they are used as the basis of this study, is in their applicability to wide variety of different problem domains and user groups. Being a graphical modelling tool, they provide similar aid to real world system design as flow charts and block diagrams for instance. On the other hand, being a formal mathematical language, they are also well suited for thorough analyzing of systems and for academic research.

4.1 Definition of Petri Nets

Formally Petri nets are defined as bipartite, directed graphs. The nodes of the graph are divided into two distinct finite sets called places and transitions. The edges of the graph are called arcs. Each arc either starts from a place and ends to a transition or vice versa. For each time step \( i \), there is a function \( M_i \) which maps a nonnegative integer value to each place. The set of these values defines the state of the system at given time step and is called a marking. Furthermore, if value \( n \) is assigned to place \( p \), we say that there are \( n \) tokens in the place \( p \), or that the place \( p \) is marked with \( n \) tokens. Preceding is summarized in the Definition 2.

**Definition 2.** A Petri net is a 4-tuple \(<P, T, F, M_0>\), where \( P \) is a finite set of places, \( T \) is a finite set of transitions, \( F \subseteq (P \times T) \cup (T \times P) \) is a set of arcs, and \( M_0 : P \rightarrow \mathbb{N} \) is an initial marking.

Graphically the places are represented as circles and transitions as boxes. A marking \( M(p) = k \) of a place \( p \) is drawn as \( k \) black dots inside the circle corresponding to a place \( p \). These are depicted in the running example of Figure 1.

Semantically the transitions of a Petri net correspond to possible events of a system and the places represent conditions or available resources. The possible event sequences and changes between states (or markings) are based on the firing rule of transitions described in the Definition 4. The definition relies on the notion of pre- and postsets, see Definition 3.

**Definition 3.** A preset of a node \( x \) is \( \cdot x = \{ y \in P \cup T \mid (y, x) \in F \} \). Similarly a postset of a node \( x \) is \( x^* = \{ y \in P \cup T \mid (x, y) \in F \} \).
**Definition 4 (Transition Firing Rule).**

1) A transition \( t \in T \) is \( M \)-enabled, \( M[t] \), if for all places \( p \in t^* : M(p) \geq 1 \).

2) If a system is in a state defined by marking \( M \), a \( M \)-enabled transition may fire. The firing of a transition produces a new marking \( M' := [M]t \) by the rule: \( \forall p \in P : M'(p) = M(p) + S(p) \), where function \( S : P \mapsto \mathbb{Z} \) indicates the change in number of tokens for the place \( p \):

\[
S(p) = \begin{cases} 
-1, & \text{if } p \in t^* \text{ and } p \notin t^*, \\
1, & \text{if } p \in t^* \text{ and } p \notin t^*, \text{ and} \\
0, & \text{otherwise.}
\end{cases}
\]

The definition of the transition firing rule states that the precondition for a transition to be able to fire, is that there has to be at least one token in each of the places belonging to the preset of the transition. The effect of firing a transition is that one token is removed from each of the places in the preset of the transition and one token is added to each place in the postset of the transition.

Besides this rule, there is no limitations on how the nodes of the Petri nets should be interpreted conceptually. Instead, depending on the modelling purpose, the transitions for instance can be thought either as self-controlled operators (e.g. a mail man) or operations that are called externally (e.g. sending of an email). Likewise, the places can be interpreted for example as conditions (e.g. a person is happy), data buffers with tokens as data or available resources (e.g. number of free ships in a harbor).

![Running example](image_url)
4.2 On Behavioral Properties and Analysis of Petri Nets

One of the central behavioral properties of Petri nets is reachability. A marking \( M \) is reachable from marking \( M_0 \) if there exists a firing sequence of transitions leading to marking \( M \) while starting from the initial marking \( M_0 \). A firing sequence is denoted by \( \sigma = t_0 \ t_1 \ t_2 \cdots \ t_n \) or by a state-transition list \( \sigma = M_0[t_0]M_1[t_1] \cdots M_{n-1}[t_{n-1}]M_n \), where between each pair of transitions the marking corresponding to the intermediate state is shown. The set of all reachable markings from the initial marking \( M_0 \) is denoted by \( R(M_0) \).

There are two main approaches to the analysis of Petri nets. These are the reachability graph method and analysis with matrix-equations. This study is, however, concerned only with the reachability graph method. The idea of this approach is basically to enumerate all the reachable markings \( R(M_0) \) of a given Petri net and to represent them as a directed graph. The reachable markings constitute the set of nodes of this graph. An edge between two markings of the graph represents the transition which causes the shift between these markings. A formal definition of reachability graph is given in Definition 5.

**Definition 5.** A reachability graph of a Petri net \( \langle P, T, F, M_0 \rangle \) is a rooted, directed and edge-labeled graph \( \langle V, E, v_0 \rangle \), where the set of nodes \( V \) is the set of reachable markings \( R(M_0) \), set of edges \( E = \{(M, t, M') | M \in V \text{ and } M' = M[t] \} \) is a set of triples combining a pair of markings with the transition \( t \) firing in \( M \) and producing \( M' \), where the root node \( v_0 \) is the initial marking \( M_0 \).

Two other important behavioral properties of Petri nets that are especially significant for this study, are \( k \)-boundedness and absence of deadlocks. A Petri net is \( k \)-bounded if for all reachable markings the number of tokens in any of the places does not exceed \( k \). More formally, for all \( M \in R(M_0) \) and all places \( p \in P : M(p) \leq k \). The concept of boundedness is significant because if a Petri net can be shown to be \( k \)-bounded for some constant \( k \), the number of different markings is guaranteed to be finite. In the context of Petri nets, a deadlock is a marking for which none of the transitions is able to fire.

In Figure 2 is depicted the reachability graph of our running example. From this figure we can see that for the Petri net of the running example, there is only three reachable markings and that the marking no. 3 is a deadlock state.

![Figure 2: Reachability graph of the running example](image-url)
5 AUTOMATED PLANNING

In this section an overview on automated planning is given. The description is based on the well known and extensive text [20] on artificial intelligence written by Stuart Russell and Peter Norvig.

Automated planning is a subfield of artificial intelligence. It concerns of finding solutions to problems where the objective is to find an action sequence that satisfies a given set of constraints and leads to a given goal state from a given initial state. A good example of such a problem is the well known Russian puzzle with wolf, goat and cabbage. In the problem setting, one has to carry the animals and the cabbage to other side of the river with a rowing boat. The complication is that only one item at a time can fit in the boat besides the rower. Moreover the wolf cannot be left alone with the goat on neither of the shores, nor can the goat be left alone with the cabbage (for obvious reasons).

The automated planning approach to solving these kinds of problems involves using a planning system, or a planner for short. Planners provide essentially two things: a language for describing planning problems and a resolution system for solving these problems, i.e. finding plans. The languages used to specify planning problems are usually based on first order logic and they provide means for giving a declarative description of the problem situation. A problem description includes typically definitions of objects and actions as well as predicates describing the states and/or the types of the objects or sets of objects. In addition the problem description has to include specifications of the initial and goal states.

Most research with automated planning has so far focused on what is called classical planning. These properties state that the problem environment has to be fully observable, deterministic, finite, static and discrete. This implies that all the elements of the world describing the problem domain are known beforehand, the world constitutes of a finite number of states and that the effects of the actions are fully known before the action is executed. Moreover, the time consists of discrete steps and it is considered as an indexing method for the action sequence rather than actual time. If these characteristics are not met, the problem is classified as nonclassical planning problem.

The problem types that are best suited for planners restricted to conditions of classical planning include for instance various scheduling problems and decision problems rising from applications of artificial intelligence. Combinatorial optimization problems can also be formulated as planning problems. In this case the optimization is expressed as a decision problem of whether there exists a solution with given length. The search is repeated by improving the length each time a new solution is found.
5.1 Petri Net Deadlock Detection as a Planning Problem

In this section we will show how a bounded model checking task of detecting deadlocks in 1-bounded Petri nets can be represented as a planning problem (for definitions of $k$-boundedness and deadlocks see Section 4.2). The target language will be PDDL (Planning Domain Definition Language) as it is the language used with Satplanner. We will explain only the parts of PDDL which are needed with this particular example. More information on the language can be found from [1].

As an example case we will consider the running example of the Figure 1. A formal representation of the running example is given as $\langle P, T, F, M_0 \rangle$, where:

\[
\begin{align*}
P &= \{p_1, p_2, p_3, p_4, p_5\}, \\
T &= \{t_1, t_2, t_3\}, \\
F &= \{(p_1, t_1), (p_2, t_1), (p_3, t_1), (t_1, p_2), (t_1, p_4), (p_4, t_2), \\
&\quad (t_2, p_1), (t_2, p_3), (p_2, t_3), (p_3, t_3), (t_3, p_5)\}, \text{ and} \\
M_0 &= \{p_1, p_2, p_3\}.
\end{align*}
\]

The syntax of PDDL reminds somewhat of the syntax of Lisp programming language. The problem descriptions constitute of nested parenthesized clauses in prefix notation. The problem descriptions are divided into domain and problem parts. In our case the domain part will hold definitions for constants, predicates, and actions whereas the specifications of initial and goal states are given in the problem part.

The domain part of a PDDL definition begins with naming the domain and optionally defining a set requirements which might specify for instance a subset of PDDL that will be used. PDDL contains STRIPS and ADL as subsets and some planners might be restricted to one of these. In our case the STRIPS subset is sufficient so the outermost parenthesis enclosing the rest of the domain part looks like this:

```
(define (domain Running_example)
  (:requirements :strips)
)
```

Because we have restricted to Petri nets that are 1-bounded, the construction of the planning problem is rather straightforward. Since the number of tokens in any of the places can only be either 0 or 1, we can represent the places as constants and a marking of a place with a predicate having a place as a parameter. Then the value of this predicate will be $false$ for places with no token and $true$ for the places marked with a token. With the set of places of the running example and the predicate named as hasToken(place), this is expressed in PDDL syntax as:

```
(:constants p1 p2 p3 p4 p5)
(:predicates (hasToken ?place))
```
The question mark on the lower line indicates a definition of a parameter of a predicate.

Transitions will be presented in PDDL as actions. Actions consist of a precondition-clause and an effect-clause, which both are logical statements on one or more of the predicates defined in the domain part. The precondition has to be satisfiable in order that the action can be performed. The effect specifies the change caused by the usage of the action to the state of the system. In practice, as a result of performing an action the values of the predicates in the effect-clause are fixed so that the value of the effect-clause will be true.

In our case the precondition corresponds to the precondition of the firing rule of transitions (Definition 4 on page 6) which states that there has to be a token in each of the places belonging to the preset of a transition in order that the transition could fire. In practice this is expressed in PDDL as an and-clause whose arguments are the hasToken-predicates on each of the states belonging to the preset of the transition.

The effect of an action corresponds in our case to the effect of the firing rule of transitions, which informally states that one token is added to each place in the postset of the transition and one token is subtracted from each place in the preset of the transition (the outcome will be the net result of these actions). In PDDL this is expressed as an and-clause whose arguments are either hasToken predicates or negated hasToken predicates (which is achieved by surrounding the predicate with a not-clause) on the places belonging to the preset and the postset of the transition. If a place belongs only to the preset of the transition, the hasToken-predicate will not hold for it and a corresponding negated hasToken-predicate is added to the and-clause of the effect. If the place belongs only to the postset of the transition, the hasToken-predicate will hold for it after the action is used (which corresponds to the firing of a transition) and it is added to the and-clause of the effect. Also, if a place belongs to both, the preset and the postset of the transition, a hasToken-predicate holds for it and is added to the and-clause of the effect. In the latter case the same result would also be achieved if the hasToken-predicate would be left out of the and-clause of the effect, since the precondition-clause already guarantees that it will be true if the transition is fired. However, in this approach one has to deal with the special case in which the preset and postset are the same, since then the effect clause would be empty which is not legal in PDDL syntax. This problem can be solved by leaving out all actions corresponding to transitions whose preset and postset are the same. This can be done, because presence of such an action would not have any kind of an effect to the result. In this case the goal statement still has to include a requirement also for these transitions - even they are left out - that they cannot be enabled in a deadlock state. In our transformation we did not leave out those predicates which are already guaranteed to be true by the precondition-clause from the and-clause of the effect, but we did leave out the actions corresponding to transitions for which the preset and postset are the same from the PDDL definition.
Here is the definition of the action corresponding to the transition $t_1$ in PDDL syntax:

```
(:action t1
 :precondition (and
 (hasToken p1)
 (hasToken p2)
 (hasToken p3)
 )
 :effect (and
 (not (hasToken p1))
 (not (hasToken p3))
 (hasToken p2)
 (hasToken p4)
 )
)
```

The problem part specifies that the initial state corresponds to the initial marking $M_0$ of the running example and that the goal state should correspond to a deadlock state. In PDDL syntax the initial state is defined with an init-clause which simply contains a list of all the predicate statements that are true in the initial state (in our case this would be a list of hasToken predicates on those places that have token in $M_0$). A deadlock state (i.e. a state in which there are no enabled transitions) is represented with a logical statement necessitating that for each transition there is at least one place with no token belonging to the preset of the transition. That is, we want to formulate in PDDL a sentence: “It does not hold that there exists a transition so that all the places who belong to it’s preset are marked with a token.” In PDDL syntax this is achieved by constructing a not-clause including a or-clause over a list of and-clauses, so that there is exactly one and-clause per each transition stating that the hasToken-predicate holds for all places belonging to the preset of the transition.

Here is the complete problem definition of our running example in PDDL:

```
(define (problem marking_M0)
 (:domain Running_example)
 (:init (hasToken p1) (hasToken p2) (hasToken p3))
 (:goal (not
 (or
 (and
 (hasToken p1)
 (hasToken p2)
 (hasToken p3)
 )
 (and (hasToken p4))
 (and (hasToken p2) (hasToken p3))
 )
)
)
)
```

The complete PDDL description of our running example is given in Appendix A.

5.2 Classical Planning as Satisfiability

Currently satisfiability solving is considered to be one of the best approaches to solving planning problems [19]. In this approach a planning problem instance is encoded to a propositional satisfiability problem and solved with a SAT-solver [13]. If a satisfying truth assignment is found, it is used to construct a solution to the planning problem. One of the strengths of the SAT-based approach is that it makes possible to construct parallel plans, which generally reduce the search time of a satisfying truth assignment. A plan without parallelism would be a sequence of actions with one action executed at each time step. Instead, with parallel plans there can be several operations executed at one time step as long as the operations fulfill certain conditions. There are different semantics which specify the necessary conditions for that a set of actions could be executed parallel. In the standard state-based encoding [13] the condition is informally, that the actions should be possible to be executed in any given order and that the resulting state must not depend on the execution order (whether or not a pair of actions can be executed one after the other depends on the preconditions and effects of the actions.)

Satplanner, the planner considered with this study, is based on a more relaxed 1-linearization semantics [19]. It allows a set of actions to be executed parallel at one time step if there exists at least one possible execution order of the actions. This approach makes it possible to execute even more actions parallel.

6 EXPERIMENTS WITH SATPLANNER AS A BOUNDED MODEL CHECKER

The purpose of this study is to examine the applicability of Satplanner in solving bounded model checking problems while compared to traditional bounded model checking of Petri nets. For the specific model checking problems we have chosen the detection of deadlocks in 1-bounded Petri nets (for definitions of k-boundedness and deadlocks see Section 4.2).

The approach to solving bounded model checking problems with Satplanner involves describing the problem in PDDL and then solving it with Satplanner. In the Section 5.1 it was shown how the problem of detecting deadlocks in 1-bounded Petri nets can be described in PDDL. The construction of the PDDL description together with application of Satplanner will be referred to as the Satplanner method. As the point of comparison for the Satplanner method we used 1b-pn-bmc [9] by Keijo Heljanko and bc2cnf [10] by Tommi Junttila with the Siege SAT-solver. Siege was chosen to improve the comparability of the approaches since Siege is the SAT-solver used by Satplanner. 1b-pn-bmc is a tool for creating a propositional formula of the problem of detecting deadlocks in 1-bounded Petri nets and it is based on Heljanko’s research on encoding bounded reachability checking as propositional formula [8]. The bc2cnf tool is needed to transform the output of 1b-pn-bmc tool into conjunctive normal form which is the input form ex-
pected by Siege. We will refer to the application of the 1b-pn-bmc, bc2cnf, and Siege tools as the *reference method*.

The benchmarks used in this study are originally made by James C. Corbett of University of Hawaii [6] and they have been translated into 1-bounded Petri nets by Toni Jussila for his doctoral thesis [12]. The only exception is *byzagr4_2a* which is from a technical report by Stephan Merkel [16].

### 6.1 On Performance Metrics and Measurements

The things we measure in this study are the length of the shortest counterexample for each of the benchmarks and the search times needed to find these counterexamples in the case that their lengths were already known. That is, for both approaches we first searched and recorded the lengths of the shortest counterexamples and only after that measured the search times by starting the search with initial lengths corresponding to the recorded shortest lengths.

In the case of the Satplanner method the total search time consists mainly of the time used by Satplanner. That is, the time needed for creating the PDDL description is insignificant and it is not included in the results presented below. As described in the Section 5.2, the operation of Satplanner consists of encoding the planning problem into a propositional satisfiability formula and the time used by Siege. Time consumption of these two phases was measured and recorded separately. Similarly with the reference method, the proportion of the time consumed by 1b-pn-bmc tool is negligible and therefore it is not included in our results. The times measured in the case of the reference method were the processing times of the bc2cnf and Siege tools. As with the Satplanner method, these times were recorded separately. Because Siege uses randomized techniques the time used by Siege varies largely. For this reason our results are actually averaged over fifty iterations.

In the case of the timings of the Satplanner method there was an additional thing to be considered. By default Satplanner calculates certain invariants of the planning problem instance while encoding it into a SAT problem instance. This is done in order to produce a more efficient SAT encoding of the planning problem. However, in the case of our benchmarks, the time needed to calculate the invariants was in most of the cases larger (in some cases significantly) than the time gained in consequence of the reduced search time of Siege. Because neither the Satplanner configuration or the site distributing Satplanner did not contain any documentation describing the invariant calculation, there is not any obvious reason seen why it should be left out or included in the case of a particular problem instance. Therefore in the case of the Satplanner method we carried out the measurements previously described with both ways, with and without the calculation of invariants. Satplanner reports the exact time that it uses to calculate invariants, so with the measurements with invariants along, we recorded calculation times used to create plan, calculate the invariants, time used by Siege and the total processing time. With measurements without the invariants along, we recorded calculation times used to create plan, time used by Siege and the total processing time as already described. With these methods we also iterated the

---

1The state spaces of the Petri nets here match Corbett’s SMV models, unlike the Petri net translations used in [8, 15].
measurements fifty times and report the average results.

6.2 Test Equipment and Configurations

All experiments on the benchmarks are carried out with a PC with AMD Athlon 64 2.0 GHz processor and 2 GiB of RAM. Available virtual memory was limited to 1.5 GiB. The operating system is Debian GNU/Linux. The versions of Satplanner and 1b-pn-bmc could not be elicited but the configurations used were downloaded on 15.1.2007. Siege version 4 and bc2cnf version 0.26 were used. Satplanner and bc2cnf are compiled with the gcc 3.3.5 compiler.

In the case of the reference method all the tools involved were used with default settings (i.e., without specifying any command line parameters). Timings of the reference method were carried out with the unix system command /usr/bin/time. Only time used by the process in the user mode was measured.

Satplanner was configured to use only one process (-A1 flag) and to invoke Siege only once (-n=1 flag). While measuring the search time for counterexamples with fixed length $k$, the plan length was specified with the command line parameter -$T=k$. Since Satplanner invokes Siege automatically, the /usr/bin/time command could not be used in timings with the Satplanner method. Instead, the processing times reported by Satplanner itself were recorded. This makes little difference however, since Satplanner also records only the time used by the process in the user mode. Also Satplanner specifies clearly how much time it has used in different phases of the processing as well as the time used by Siege.

As described in the previous section, the timings of Satplanner were carried out with and without the calculation of invariants. The latter could be achieved by using the -inv command line parameter. This parameter was not documented anywhere but the functionality is implemented as was confirmed by Markus Büttner. However, with -inv flag the time keeping is left for the user. When -inv flag is used, Satplanner still calculates the invariants but it does not use them in the encoding process. Moreover, Satplanner reports the time used to calculate the invariants but it does not subtract this time from the total processing time, thus we subtracted it manually in the results reported below.
7 RESULTS

Our measurements are presented in Tables 1, 2, and 3. Table 1 contains timings of the Satplanner method with and without the calculation of invariants. Table 2 contains comparison on timings of the Satplanner method and the reference method and the shortest counterexamples of the methods are shown in the Table 3. Meaning of the columns of Tables 1 and 2 are described below the tables respectively. In the case of Tables 1 and 2 the times reported are averaged over fifty iterations.

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Plan_{inv}</th>
<th>Inv</th>
<th>Plan_{inv} + Inv</th>
<th>Plan</th>
<th>SAT_{inv}</th>
<th>SAT</th>
<th>Total_{inv}</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>byzagr4_2a</td>
<td>1.14</td>
<td>24.58</td>
<td>25.72</td>
<td>0.55</td>
<td>0.42</td>
<td>0.15</td>
<td>26.19</td>
<td>0.68</td>
</tr>
<tr>
<td>dac_15</td>
<td>0.05</td>
<td>0.18</td>
<td>0.22</td>
<td>0.05</td>
<td>0.01</td>
<td>0.01</td>
<td>0.24</td>
<td>0.04</td>
</tr>
<tr>
<td>darts_1</td>
<td>0.57</td>
<td>9.58</td>
<td>10.16</td>
<td>0.16</td>
<td>1.12</td>
<td>0.17</td>
<td>11.28</td>
<td>0.34</td>
</tr>
<tr>
<td>dp_12</td>
<td>0.02</td>
<td>0.02</td>
<td>0.04</td>
<td>0.02</td>
<td>0.00</td>
<td>0.00</td>
<td>0.04</td>
<td>0.02</td>
</tr>
<tr>
<td>elevator_3</td>
<td>0.49</td>
<td>4.18</td>
<td>4.67</td>
<td>0.38</td>
<td>0.13</td>
<td>0.13</td>
<td>4.80</td>
<td>0.51</td>
</tr>
<tr>
<td>elevator_4</td>
<td>3.31</td>
<td>33.05</td>
<td>36.37</td>
<td>2.35</td>
<td>0.54</td>
<td>1.15</td>
<td>36.91</td>
<td>3.30</td>
</tr>
<tr>
<td>hartstone_100</td>
<td>0.98</td>
<td>25.66</td>
<td>26.64</td>
<td>0.23</td>
<td>7.82</td>
<td>23.84</td>
<td>34.45</td>
<td>24.07</td>
</tr>
<tr>
<td>hartstone_75</td>
<td>0.47</td>
<td>10.93</td>
<td>11.39</td>
<td>0.14</td>
<td>5.49</td>
<td>8.33</td>
<td>16.88</td>
<td>8.27</td>
</tr>
<tr>
<td>key_2</td>
<td>0.05</td>
<td>0.39</td>
<td>0.44</td>
<td>0.03</td>
<td>0.28</td>
<td>10.96</td>
<td>11.00</td>
<td></td>
</tr>
<tr>
<td>key_3</td>
<td>0.08</td>
<td>0.75</td>
<td>0.83</td>
<td>0.05</td>
<td>0.52</td>
<td>14.83</td>
<td>14.88</td>
<td></td>
</tr>
<tr>
<td>key_4</td>
<td>0.13</td>
<td>1.27</td>
<td>1.40</td>
<td>0.07</td>
<td>0.55</td>
<td>24.18</td>
<td>24.25</td>
<td></td>
</tr>
<tr>
<td>key_5</td>
<td>0.18</td>
<td>1.81</td>
<td>1.99</td>
<td>0.09</td>
<td>0.84</td>
<td>24.50</td>
<td>24.59</td>
<td></td>
</tr>
<tr>
<td>mmgt_3</td>
<td>0.08</td>
<td>0.26</td>
<td>0.34</td>
<td>0.05</td>
<td>0.04</td>
<td>0.04</td>
<td>0.08</td>
<td>0.09</td>
</tr>
<tr>
<td>mmgt_4</td>
<td>0.13</td>
<td>0.46</td>
<td>0.59</td>
<td>0.07</td>
<td>0.06</td>
<td>0.14</td>
<td>0.65</td>
<td>0.21</td>
</tr>
<tr>
<td>q_1</td>
<td>0.09</td>
<td>1.18</td>
<td>1.27</td>
<td>0.07</td>
<td>0.06</td>
<td>0.06</td>
<td>1.33</td>
<td>0.09</td>
</tr>
<tr>
<td>sentest_100</td>
<td>0.52</td>
<td>8.50</td>
<td>9.02</td>
<td>0.13</td>
<td>3.98</td>
<td>13.00</td>
<td>13.00</td>
<td></td>
</tr>
<tr>
<td>sentest_75</td>
<td>0.28</td>
<td>3.89</td>
<td>4.17</td>
<td>0.09</td>
<td>1.86</td>
<td>0.19</td>
<td>6.03</td>
<td>0.28</td>
</tr>
<tr>
<td>speed_1</td>
<td>0.03</td>
<td>0.00</td>
<td>0.03</td>
<td>0.03</td>
<td>0.00</td>
<td>0.00</td>
<td>0.03</td>
<td>0.03</td>
</tr>
</tbody>
</table>

Table 1: Timings of the Satplanner method with and without the calculation of invariants. Times are averaged over fifty iterations.

Plan_{inv} : Average time used to create the planning problem instance while invariants were used.

Inv : Average time used to calculate the invariants with the timings in which they were used.

Plan_{inv} + Inv : Combined average time of the previous two columns. This column represents the total time used by the Satplanner before it invokes Siege in the case where invariants were used.

Plan : Average time used to create the planning problem instance while invariants were not used.

SAT_{inv} : Average time used to solve the SAT problem instance by Siege while invariants were used.

SAT : Average time used to solve the SAT problem instance by Siege while invariants were not used.

Total_{inv} : Average total time when invariants were used.

Total : Average total time when invariants were not used.

RM_{cnf} : Average time used while creating the CNF problem instance in the reference method.
Table 2: Comparison on timings of the Satplanner method and the reference method. Times are averaged over fifty iterations.

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>RM\textsubscript{cnf}</th>
<th>SM\textsubscript{plan}</th>
<th>RM\textsubscript{sat}</th>
<th>SM\textsubscript{sat}</th>
<th>RM\textsubscript{tot}</th>
<th>SM\textsubscript{tot}</th>
</tr>
</thead>
<tbody>
<tr>
<td>byzagr4_2a</td>
<td>0.41</td>
<td>0.53</td>
<td>0.08</td>
<td>0.15</td>
<td>0.49</td>
<td>0.68</td>
</tr>
<tr>
<td>dac_15</td>
<td>0.02</td>
<td>0.03</td>
<td>0.02</td>
<td>0.01</td>
<td>0.04</td>
<td>0.04</td>
</tr>
<tr>
<td>dartes_1</td>
<td>1.48</td>
<td>0.16</td>
<td>0.01</td>
<td>0.17</td>
<td>1.49</td>
<td>0.34</td>
</tr>
<tr>
<td>dp_12</td>
<td>0.02</td>
<td>0.02</td>
<td>0.02</td>
<td>0.00</td>
<td>0.04</td>
<td>0.02</td>
</tr>
<tr>
<td>elevator_3</td>
<td>0.33</td>
<td>0.38</td>
<td>0.04</td>
<td>0.13</td>
<td>0.37</td>
<td>0.51</td>
</tr>
<tr>
<td>elevator_4</td>
<td>1.27</td>
<td>2.35</td>
<td>0.21</td>
<td>1.15</td>
<td>1.48</td>
<td>3.50</td>
</tr>
<tr>
<td>hartstone_100</td>
<td>9.80</td>
<td>0.23</td>
<td>21.69</td>
<td>23.84</td>
<td>31.48</td>
<td>24.07</td>
</tr>
<tr>
<td>hartstone_75</td>
<td>3.98</td>
<td>0.14</td>
<td>6.63</td>
<td>8.13</td>
<td>10.60</td>
<td>8.27</td>
</tr>
<tr>
<td>key_2</td>
<td>0.51</td>
<td>0.03</td>
<td>17.92</td>
<td>10.96</td>
<td>18.43</td>
<td>11.00</td>
</tr>
<tr>
<td>key_3</td>
<td>0.82</td>
<td>0.05</td>
<td>22.20</td>
<td>14.83</td>
<td>23.02</td>
<td>14.88</td>
</tr>
<tr>
<td>key_4</td>
<td>1.23</td>
<td>0.07</td>
<td>24.32</td>
<td>24.18</td>
<td>25.54</td>
<td>24.25</td>
</tr>
<tr>
<td>key_5</td>
<td>1.73</td>
<td>0.09</td>
<td>30.69</td>
<td>24.50</td>
<td>32.42</td>
<td>24.59</td>
</tr>
<tr>
<td>mmgt_3</td>
<td>0.05</td>
<td>0.05</td>
<td>0.02</td>
<td>0.04</td>
<td>0.08</td>
<td>0.09</td>
</tr>
<tr>
<td>mmgt_4</td>
<td>0.11</td>
<td>0.07</td>
<td>0.12</td>
<td>0.14</td>
<td>0.23</td>
<td>0.21</td>
</tr>
<tr>
<td>q_1</td>
<td>0.11</td>
<td>0.07</td>
<td>0.02</td>
<td>0.03</td>
<td>0.13</td>
<td>0.09</td>
</tr>
<tr>
<td>sentest_100</td>
<td>7.93</td>
<td>0.13</td>
<td>0.22</td>
<td>0.36</td>
<td>8.15</td>
<td>0.49</td>
</tr>
<tr>
<td>sentest_75</td>
<td>3.26</td>
<td>0.09</td>
<td>0.10</td>
<td>0.19</td>
<td>3.36</td>
<td>0.28</td>
</tr>
<tr>
<td>speed_1</td>
<td>0.02</td>
<td>0.03</td>
<td>0.02</td>
<td>0.00</td>
<td>0.04</td>
<td>0.03</td>
</tr>
</tbody>
</table>

**SM\textsubscript{plan}:** Average time used while preprocessing the planning problem instance and encoding it into CNF form in the Satplanner method.

**RM\textsubscript{sat}:** Average time used to solve the SAT problem instance by Siege with the reference method.

**SM\textsubscript{sat}:** Average time used to solve the SAT problem instance by Siege with the Satplanner method.

**RM\textsubscript{tot}:** Average total time used by the reference method.

**SM\textsubscript{tot}:** Average total time used by the Satplanner method.

**Discussion on the Results**

Table 1 shows that the calculation of invariants is reducing total processing time only in the case of key\textsubscript{x} series. With these benchmarks the invariants seem to significantly lower the time used by Siege and also the time needed to calculate the invariants is very small compared to the gained time. Also with hartstone_100 the Siege part is much faster when invariants are used but in this case the calculation of invariants takes more time than is gained by them. In nearly all the rest of the cases the reduced processing time of Siege is very small and the time needed to calculate the invariants is very large leading to net increase in the total processing time. In the case of sentest_100 the processing time of Siege is actually significantly larger when the invariants are used.

Table 2 shows timings of the reference method and the Satplanner method without the calculation of invariants. Our results show that nearly with half of our benchmarks the Satplanner method seems to be clearly faster than...
Table 3: Shortest counterexamples for the methods compared.

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Reference method</th>
<th>Satplanner method</th>
</tr>
</thead>
<tbody>
<tr>
<td>byzagri_2a</td>
<td>8</td>
<td>8</td>
</tr>
<tr>
<td>dac_15</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>dartes_1</td>
<td>32</td>
<td>32</td>
</tr>
<tr>
<td>dp_12</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>elevator_3</td>
<td>8</td>
<td>8</td>
</tr>
<tr>
<td>elevator_4</td>
<td>10</td>
<td>10</td>
</tr>
<tr>
<td>hartstone_100</td>
<td>201</td>
<td>201</td>
</tr>
<tr>
<td>hartstone_75</td>
<td>151</td>
<td>151</td>
</tr>
<tr>
<td>key_2</td>
<td>36</td>
<td>36</td>
</tr>
<tr>
<td>key_3</td>
<td>37</td>
<td>36</td>
</tr>
<tr>
<td>key_4</td>
<td>38</td>
<td>36</td>
</tr>
<tr>
<td>key_5</td>
<td>39</td>
<td>36</td>
</tr>
<tr>
<td>mmgt_3</td>
<td>7</td>
<td>7</td>
</tr>
<tr>
<td>mmgt_4</td>
<td>8</td>
<td>8</td>
</tr>
<tr>
<td>q_1</td>
<td>9</td>
<td>9</td>
</tr>
<tr>
<td>sentest_100</td>
<td>108</td>
<td>108</td>
</tr>
<tr>
<td>sentest_75</td>
<td>83</td>
<td>83</td>
</tr>
<tr>
<td>speed_1</td>
<td>4</td>
<td>4</td>
</tr>
</tbody>
</table>

The results shown in the Table 3 show that the Satplanner method finds shorter counterexamples than the reference method only in the case of key_x series. In the rest of the cases the length of the shortest counterexamples are equal between the methods. Unfortunately in the case of key_x series there does not seem to be any correlation between the difference in the lengths of the shortest counterexamples of the methods and the difference of total processing time of the methods, and therefore we cannot make any conclusions on how the reduced length of a counterexample affects the processing time.
This study clearly shows that the ideas used in Satplanner might be suitable also for bounded model checkers. Our results are twofold as they show that in some cases Satplanner outperforms our reference method in consequence of faster encoding phase and in some cases in consequence of better encoding of a problem instance to a CNF instance resulting faster processing by a SAT solver. For this reason there is clearly a need for closer study on the causes of the better performance of the Satplanner method.

One of the interesting questions and subjects for further study this study rises is related to the implementation of invariant calculation in Satplanner and whether it could be modified to suit better model checking tasks. This is because our results show that in the case of our benchmarks the invariant calculation of Satplanner is very time consuming process but it seems to provide significant efficiency increase only in few cases. This might result from the fact that 1-bounded Petri net benchmarks used in this work are by nature such that they contain lots of invariants. In fact, in the case of the benchmarks of this study it would be easy to list many of the invariants directly based on the a priori knowledge that they are synchronizations of state machines. Therefore, as Satplanner uses some general algorithm for calculating the invariants, this process could possibly be speeded up substantially by exploiting a priori knowledge on the specific problem domain.
A PDDL PRESENTATION OF THE RUNNING EXAMPLE

Here is a complete PDDL presentation of the running example of Figure 1.

Domain definition:

(define (domain Running_example)
 (:requirements :strips)
 (:constants p1 p2 p3 p4 p5)
 (:predicates (hasToken ?x))

(:action t1
   :precondition (and
                     (hasToken p1)
                     (hasToken p2)
                     (hasToken p3)
                   )
   :effect (and
             (not (hasToken p1))
             (not (hasToken p3))
             (hasToken p2)
             (hasToken p4)
           )
  )
)

(:action t2
   :precondition (hasToken p4)
   :precondition (hasToken p4)
   :effect (and
             (not (hasToken p4))
             (hasToken p1)
             (hasToken p3)
           )
  )
)

(:action t3
   :precondition (and
                   (hasToken p2)
                   (hasToken p3)
                 )
   :effect (and
             (not (hasToken p2))
             (not (hasToken p3))
             (hasToken p5)
           )
  )
)
Problem definition:

(define (problem marking_M0)
  (:domain Running_example)
  (:init (hasToken p1) (hasToken p2) (hasToken p3))
  (:goal (not
    (or
      (and
        (hasToken p1)
        (hasToken p2)
        (hasToken p3)
      )
      (and (hasToken p4))
      (and (hasToken p2) (hasToken p3))
    )
  )
)
)

REFERENCES


