On the Use of Planning Technology for Verification

Aws Albarghouthi Jorge A. Baier Sheila A. McIlraith
Department of Computer Science,
University of Toronto,
Toronto, Ontario. Canada.
{aws,jabaier,sheila}@cs.toronto.edu

Abstract

Formal verification of hardware and software systems involves proving or disproving the correctness of the intended behaviour of the system with respect to a formal specification of the system. An effective means of automating this process is by representing properties to be verified in a temporal logic and exploiting model checking or theorem proving. In this paper we examine the use of planning techniques to perform automated verification. In particular, we exploit recent advances in heuristic search-based planning for temporally extended goals. We translate a specification of the behaviour of our system into planning domains, and specify safety, liveness, and fairness properties and constraints in linear temporal logic (LTL). Verification of these properties and constraints can then be achieved by some form of planning where LTL formulae are treated as temporally extended goals. To illustrate our approach, we translate models from SMV to planning domains and problems in PDDL. Our initial experimental results show that planners can provide significant time improvements when checking safety and liveness properties compared to state-of-the-art model checkers.

Introduction

Automated verification of hardware and software systems has become increasingly prevalent, particularly in the design and development of safety-critical systems. Formal verification is the act of proving or disproving designated properties of a system with respect to an abstract mathematical model of the system. Properties are often expressed in a temporal logic such as linear temporal logic (LTL) or computational tree logic (CTL). The mathematical model of the system is typically expressed as automata, a labelled transition system, a Petri net, or for programming languages, in terms of the operational, denotational, or axiomatic semantics of the language. Automation of the verification process is commonly done via model checking or theorem proving. In the case of theorem proving, the automation is often only partial.

In this paper we explore the hypothesis that state-of-the-art planning techniques can improve the efficiency of a number of automated verification tasks. To this end, we propose to encode verification problems as planning problems and to apply state-of-the-art planning techniques to construct formal proofs.

We are not the first to explore the application of planning techniques to automated verification. Edelkamp, Lafuente, and Leue (2001) explored the use of directed search algorithms in explicit state model checking – so-called *directed model checking*. They applied this idea to Promela, the input language of the SPIN model checker (Holzmann 1997), verifying safety properties and general temporal properties in LTL. In 2003, Edelkamp showed how model checking problems in a subset of Promela can be converted to the Plan Domain Description Language (version 2.1) PDDL2.1 (Fox and Long 2003). He then showed how simple safety properties of these models can be checked using existing heuristic search planners. He showed that the Metric-FF planner was competitive with directed model checkers and superior to standard model checkers for checking safety properties.

Here we explore three broad classes of verification problems: safety properties, liveness properties, and fairness constraints. As demonstrated by Edelkamp, it is straightforward for a planner to check simple safety properties by viewing the plan as a finite counterexample of a safety invariant. Thus the main open challenges lie in checking complex safety properties, liveness properties, and in the application of fairness constraints. Unlike safety properties, liveness properties require counterexamples with loops, but standard heuristic search planners are incapable of producing plans with loops.

Central to our approach is the exploitation techniques for planning with temporally extended goals (TEGs). In particular, we exploit a technique proposed by Baier and McIlraith (2006) which enables the use of fast heuristic search planners for planning with TEGs, often leading to orders-of-magnitude improvements over approaches that do not exploit heuristics. We formally establish the relationship between our three verification tasks and planning with TEGs. From this characterization, we are able to use TEGs to find plans that act as counterexamples of safety properties. In the case of liveness properties and fairness constraints, we employ techniques introduced by Schuppan and Biere (2004) to reduce liveness checking to safety checking.

We demonstrate our proposed approach by translating a subset of the SMV modeling language used by model checkers such as NuSMV (Cimatti et al. 2002) to the Planning Domain Definition Language (PDDL) (McDermott 1998), and comparing the performance of NuSMV against the heuristic search planner Fast Forward (FF) (Hoffmann and Nebel 2001) using our TEG planning domain encoding. We

checked safety and liveness properties for various instances of the dining philosophers problem, increasing the number of philosophers. In the case of safety checking, FF significantly outperformed NuSMV on the verification of false properties solving far more problems, while NuSMV minimally outperformed FF on the verification of true properties. In the case of liveness, FF significantly outperformed NuSMV on the verification of a false liveness property, scaling well as the number of philosophers increased. While our experimental analysis is preliminary, it is sufficient to demonstrate the strong potential of our approach and supports the hypothesis that state-of-the-art planning techniques can improve the efficiency of a number of automated verification tasks.

In the next section we provide background on verification and planning. We follow by describing our translation from Kripke models, used to model systems by model checkers, to planning domains and problems. Then, we introduce our approach to checking safety properties along with experimental results. We follow by describing our reduction from liveness properties and fairness constraints to reachability properties accompanied by experimental results. This is followed by a summary of our contributions and a discussion of future work.

Background

In this section we review verification and classical planning.

Verification

In the general case, the formal specification of a system to be verified is described in terms of a potentially infinite state space. However, many approaches to automated verification construct a system model that is finite or for which infinite sets of states can be effectively represented finitely. We do likewise here, following the model checking literature, and represent systems to be verified as *Kripke structures*. This restriction to finite state spaces will enable us to exploit state-of-the-art propositional planners.

Following Clarke, Grumberg, and Peled's notation (2000), we define a *Kripke structure* M as a four tuple $M = (S, I, R, L)$ where S is a finite set of states comprising the model; I is a set of initial states where $I \subseteq S$; and R is a transition relation $R \subseteq S \times S$, where for every $s \in S$, there exists $s' \in S$ such that $R(s, s')$ holds. $L : S \rightarrow 2^{AP}$ is an injective labeling function from states to atomic propositions, where AP is a set of atomic propositions. For a state $s \in S$, $L(s)$ is the set of atomic propositions true in s. $R(s, s')$ holds iff there is a transition from state s to state s'.

Since L is injective, for a particular set of atoms T there exists at most one state s in M such that $L(s) = T$. Thus, to simplify notation, in the rest of the paper we use s instead of $L(s)$ when the meaning can be disambiguated from the context.

An LTL formula is comprised of atomic propositions, logical operators, and temporal operators. LTL formulae can be one of the following:

1. an atomic proposition p.
2. $\neg\phi, \phi \vee \psi, \phi \wedge \psi, \mathbf{F}\phi, \mathbf{G}\phi, \mathbf{X}\phi, \mathbf{F}\phi, \phi\mathbf{U}\psi$, where ϕ and ψ are LTL formulae.

The semantics of LTL formulae are defined over paths. A path π is a possibly infinite sequence of states $s_0 s_1 s_2 \cdots$, where $(s_i, s_{i+1}) \in R$ for all $i < |\pi|$. $|\pi|$ denotes the number of states in the path. Moreover, $\pi(i)$ denotes the i-th state in π, and π_i denotes the suffix of π that starts from the i-th state. We use the notation $\pi \models \phi$ to express that formula ϕ is satisfied by a path π. The $\models$ relation is defined as follows.

1. $\pi \models p$ iff $p \in L(\pi(0))$
2. $\pi \models \neg\phi$ iff $\pi \not\models \phi$
3. $\pi \models \phi \wedge \psi$ iff $\pi \models \phi$ and $\pi \models \psi$
4. $\pi \models \mathbf{F}\phi$ iff $\exists i < |\pi|.\pi_i \models \phi$
5. $\pi \models \mathbf{X}\phi$ iff $|\pi| > 1$ and $\pi_1 \models \phi$
6. $\pi \models \mathbf{G}\phi$ iff $\forall i < |\pi|.\pi_i \models \phi$
7. $\pi \models \phi\mathbf{U}\psi$ iff $\exists i < |\pi|.\pi_i \models \psi \wedge \forall j < i.\pi_j \models \phi$

When checking if a Kripke structure M satisfies an LTL formula ϕ (written $M \models \phi$), we check if the formula holds on *all* maximally expanded paths starting from all initial states. If there is a maximally expanded path where the formula does not hold, then that path is a counterexample to the satisfiability of the formula in the Kripke structure under consideration.

Fairness constraints are a set of subsets of states $\{F_1, ..., F_n\}$, where $F_i \subseteq S$. Let

$$inf(\pi) = \{s \mid s \in S \text{ and } s \text{ occurs infinitely often on } \pi\}$$

A path π is considered fair if it is of infinite length and for every F_i, $F_i \cap inf(\pi) \neq \emptyset$. In the presence of fairness constraints, we only check maximally expanded paths that are fair.

Safety properties specify that "something bad will never happen" while liveness properties specify that "something good will eventually happen" (Lamport 1977). Deadlock freeness and invariants (e.g. $\mathbf{G}p$) are examples of safety properties. Termination and request-response (e.g. $\mathbf{G}(p \rightarrow \mathbf{F}q)$) are examples of liveness properties. A formal definition of safety and liveness is presented by Alpern and Schneider (1987) where they show how a safety or liveness property can be identified by examining the Büchi automaton representing the property. A counterexample of a safety property is a finite path where an irremediable "bad thing" happens. A counterexample of a liveness property is an infinite path where a "good thing" never happens. When verifying non-safety properties, we will be looking for lasso-shaped counterexamples following the approach by Schuppan and Biere (2004). A lasso-shaped path is an infinite path that can be represented as ab^ω, where a and b are finite paths. a is called the stem and b is the loop.

We define a verification task V to be a tuple $V = (M, \phi, \{F_1, ..., F_n\})$. M is a Kripke structure describing the system. ϕ is the LTL property to be verified. $F_1, ..., F_n$ are fairness constraints. ϕ is satisfied by M under the fairness constraints $F_1, ..., F_n$ iff there does not exist a maximally expanded fair path π in M such that $\pi \models \neg\phi$ and π starts from an initial state in M.

A *Büchi automaton* is a finite state automaton extended to infinite inputs. Given an infinite input, the automaton accepts it iff at least one accepting state is visited infinitely often.

Classical Planning

Essentially following Ghallab, Nau, and Traverso (2004), we define a planning problem to be a tuple (S_0, F, A, G) where F is a finite set of atomic facts, $S_0 \subseteq F$ is the *initial state*, and A is a finite set of deterministic *actions*. Each action $a \in A$ is itself described by a tuple $(pre(a), add(a), del(a))$ where $pre(a)$ is a pair $(pre^+(a), pre^-(a))$ of disjoint subsets of F that define, respectively, the positive and negative preconditions of action a. On the other hand, $add(a)$ and $del(a)$ are disjoint subsets of F that define, respectively, the positive and negative effects of action a. A *planning state* is a subset of elements in F. Classical planning assumes complete information about the planning state. As such, every $f \in F$ that is not explicitly mentioned in a planning state, including the initial state, is assumed to be false in that state. An action a is applicable in a planning state $s \subseteq F$ iff $pre^+(a) \subseteq s$ and $pre^-(a) \cap s = \emptyset$. The state resulting from applying an action a in a state s is $succ(a, s) = (s \setminus del(a)) \cup add(a)$. Finally, the goal G corresponds to a set of planning states.

A *plan*, $\vec{a}$, is a finite sequence of actions $a_0, ..., a_n$ such that the application of $\vec{a}$ in S_0 yields a state s in G.

TEGs Extensions of the classical planning paradigm include planning for TEGs (e.g. Bacchus and Kabanza 1998). A TEG is a goal defined by a temporal logic formula, usually expressed in LTL. TEGs describe properties or constraints that must hold on the sequence of states traversed by the plan's execution, not just in the final state. For example, the TEG $p\mathbf{U}q$ specifies that the goal of the system is to reach a state where q is true, but on all states of the plan before achieving q, the fact p must hold.

In this paper we exploit Baier and McIlraith's approach to planning with TEGs (2006). The approach maps a planning instance with a TEG into a classical planning instance. It does so by first representing the LTL formula as a nondeterministic, parametrized finite state automaton. Then this automaton is modelled within the planning domain. To this end, auxiliary predicates are added to the planning instance that represent the states of the automaton. In particular, a subset of those predicates – the so-called *accepting predicates* – are such that they are true in a state s iff the sequence of states that lead to s satisfies the TEG. Baier and McIlraith's approach has the advantage that it can be used with heuristic search planners, which usually results in orders-of-magnitude improvements over approaches that do not use heuristic search (e.g., TLPLan).

PDDL The Planning Domain Definition Language (PDDL) (McDermott 1998) is the *de facto* standard language for representing classical planning problems. It allows describing planning actions as schemas, allowing a compact representation for families of actions that have similar preconditions and effects. For example, $drive(v, loc_1, loc_2)$ may be used as an action schema to represent a family of actions that move some vehicle v from one location loc_1 to some other location loc_2. Thus, a PDDL description of a planning problem may be more compact than an explicit representation of all the actions in A.

From System Models to Planning Problems

Our goal in this paper is to view verification as planning. As such, there are two key challenges that we need to address: (1) how we can model the system dynamics using a planning language, (2) and how we model safety/liveness properties and fairness constraints using a planning language so that we can check them with a planner.

In this section we deal with the first of those challenges. We show how a system, modeled by a Kripke structure, can be represented via planning actions and a particular initial state. First we justify theoretically that it is possible to construct a correspondence between Kripke structures and partial planning problems (for which we do not define a goal). In the rest of the section we describe a specific translation of a subset of SMV programs into PDDL.

System Models Are Partial Planning Problems

We show below two simple theoretical translations of Kripke structures into partial planning problems. Formally, a partial planning problem is a tuple (S_0, F, A), where S_0, F, and A are defined as in classical planning problems.

Our theoretical translations is used to establish a full correspondence between planning and verification. The translation we propose, however, is not practical in most cases, since its size is proportional to the cardinality of the transition function R. In the next subsection, we show a more efficient translation for a subset of the pragmatic SMV language that is much more compact.

Given a Kripke structure M, we first define the facts of the planning problem. We define $F := AP$, i.e., the facts of the planning problem correspond to the propositions of the Kripke structure. For every transition in $(s, s') \in R$, we add an action a to A by setting the set of preconditions of a to s and its effects to s'. That is to say, $pre^+(a) = s, pre^-(a) = AP \setminus s, add(a) = s' \setminus s$, and $del(a) = s \setminus s'$.

Now we turn our attention to the definition of the initial state. As opposed to Kripke structures, classical planning problems consider a unique initial state. We now present two alternative mappings of Kripke structures into classical planning. The first one maps a Kripke structure into a family of partial planning problems; the second maps it into a single planning instance. In the rest of the paper we use our first formulation to establish the theoretical connections between planning and verification, whereas the second translation is used as a foundation for the method used to compile SMV into PDDL.

Kripke Structures to Families of Partial Problems The intuition of this translation is that for a given Kripke structure we generate one partial problem for each initial state. Formally,

Definition 1 *Given a Kripke structure $M = (S, I, R, L)$, the family of partial planning problems associated with M*

is

$$\mathcal{P}_M = \{P_i = (i, F, A) \mid i \in I\},$$

where F and A are respectively the facts and actions as defined above.

There is a correspondence between paths of a Kripke structure and plans found for problems in this family. The following theorem establishes such a relation.

Theorem 1 (Correctness) *Given a Kripke structure M and the corresponding family of planning problems $\mathcal{P}_M$, then: π is a finite path starting from an initial state in M iff there exists a partial planning problem $P_i \in \mathcal{P}_M$ that has a plan $\vec{a}$ for the goal $\pi(|\pi| - 1)$, such that S_0 of P_i equals $\pi(0)$, $|\vec{a}| = |\pi| - 1$, and for all $1 \leq i < |\pi|$, $\pi(i)$ equals the result of applying the plan $\vec{a}[0..i-1]$ to S_0 in P_i.*

Proof sketch: The proof follows from the translation defined above, $(s, s') \in R$ iff there exists an action a such that $succ(a, s) = s'$. And from Definition 1 which states that for every initial state in the Kripke structure, there is a partial planning problem $P_i \in \mathcal{P}_M$ that starts from that state. □

Kripke Structures to a Single Partial Problem To generate a single classical planning problem we need to represent the fact that we could "choose" the initial state from which we wish to start. This can be simulated by having a unique initial state, in which one can perform $|I|$ actions, each of which will lead to a different state in I. To that end, we perform the following modification to handle multiple initial states:

1. Add the fact $init$ to the set of facts F.
2. Let $S_0 = \cap_{i \in I} i$
3. Add $init$ to the positive precondition of all actions in A. I.e., for all actions $a \in A$ set $pre(a) := pre(a) \cup \{init\}$.
4. For each element i in the set I of initial states of M, add an action a_i to A such that:

$$\begin{aligned} pre^+(a_i) &= \{\} \\ pre^-(a_i) &= \{init\} \\ add(a_i) &= (S_0 \setminus i) \cup \{init\} \\ del(a_i) &= \emptyset \end{aligned}$$

This procedure creates an initial state with only the facts that all initial states agree on. After that it adds the predicate $init$ to the precondition of every action. For every state in I, an action going from the new initial state to that state is added by the procedure. We call the resulting problem P^d_M.

Theorem 2 *Given a Kripke structure M, the corresponding family of partial planning problems $\mathcal{P}_M$, the corresponding partial planning problem P^d_M using algorithm above, and a goal G with $init$ false in all of its states, then: A plan exists for the partial planning problem P^d_M with the goal G iff there exists a plan for some $P_i \in \mathcal{P}_M$ and the goal G.*

Proof sketch: ($\Rightarrow$) Let $a_0, \cdots, a_n$ be a plan for P^d_M with the goal G. $succ(a_0, S_0)$, where S_0 is the initial state of P^d_M, will be the initial state of some $P_i \in \mathcal{P}_M$. The rest of the plan $a_1, \cdots, a_n$ will be a plan for P_i with the goal G.
($\Leftarrow$) Let $a_0, \cdots, a_n$ be a plan for some $P_i \in \mathcal{P}_M$ with the goal G. By the algorithm above, there exists an action a in P^d_M, such that $succ(a, S_0)$, where S_0 is the initial state of P^d_M, is the initial state of P_i. The plan $a, a_0, \cdots, a_n$ will then be a plan for P^d_M with the goal G. □

Later, we will show that in checking liveness and safety properties using planners, we will be looking for a counterexample (plan) that starts from *any* initial state in I and ends in a state violating the property. Therefore, the initial state formulation defined above creates a separate path going through every initial state in I. Even though the number of initial actions increases exponentially with the increase in the number of undefined predicates, most of the time unknown predicates specify which thread or process has control. When this is the case, we add n actions to specify which of the n processes runs first.

The goal G depends on the property we want to check and this will be addressed in the following sections.

From SMV to PDDL

We now show the source-to-source translation from a subset of SMV to PDDL. SMV is an input language for a family of model checkers. Originally it was used in the SMV model checker (McMillan 1992). The SMV language allows for compact representations of systems as state spaces. The language allows declaring variables of different types. Here we restrict our translation to models with variables of Boolean types, enumerated types, subrange types, and arrays. We disregard set operations, vectors, and constructor loops. To simplify the conversion to PDDL, we use the `build_Boolean_model` feature provided by NuSMV to convert all variables to Boolean type.

The constructs used to define the values a variable can have are `init` and `next`. `init` defines the value a variable has in the initial state. For example, `init(x)=true` specifies that the variable `x` has the value `true` in the initial state. `next` defines the value a variable holds in subsequent states. For example, `next(x)=!x` means that in the next state `x` holds a value opposite to the current state. `next` declarations can be made more complex by using switch-case statements, referring to other variables in the model, and using non-deterministic assignments.

In our translation, we add `init` definitions to the PDDL problem file as this is where initial state information is placed. On the other hand, we translate `next` definitions into actions that change the values of predicates. The examples presented in this section give an overview of how we perform this translation.

SMV models can include more than one module which usually refer to processes where each process gets chance to execute in an interleaving concurrency model. For an SMV model with more than one module, we transform it into an equivalent single module SMV model using the `flatten_hierarchy` feature in the NuSMV model checker. This transformation and the `build_Boolean_model` break down the hierarchy and

```
MODULE main
VAR
   x : {s0,s1};
ASSIGN
   init(x) := s0;
   next(x) := case
         x=s0 : s1;
         x=s1 : s0;
      esac;
```

Figure 1: A simple SMV model

```
(define (domain main)
(:predicates (s0)
             (s1))
(:action move1
   : precondition (s0)
   : effect (and (s1) (not (s0)))
(:action move2
   : precondition (s1)
   : effect (and (s0) (not (s1))))
```

Figure 2: PDDL domain

change the names of state variables, therefore, the property to be checked is also transformed.

Figure 1 represents a simple SMV model consisting of two possible states, $x = s0$ and $x = s1$. The SMV model specifies that the variable x initially holds the value $s0$. The `case` statement specifies that if in the current state $x = s0$, then x will be set to $s1$ in the following state. And if $x = s1$ in the current state, it will be set to $s0$ in the next state. The only possible path in this model is $x = s0, x = s1, x = s0, x = s1, ...$ Figure 2 shows an equivalent PDDL domain description with two predicates describing the two possible states. The two conditions of the case statement are handled by the preconditions of the of the actions $move1$ and $move2$ in PDDL. state and the goal are defined in the PDDL problem definition, which is not shown here. In the case of Figure 1, the initial state is $x = s0$ and the goal depends on the property to be verified.

Non-deterministic Transitions To handle non-determinism, we add more than one action to capture the different possible transitions. Figure 3 specifies a model that starts at state $x = s0$ with every subsequent state non-deterministically set to either $x = s0$ or $x = s1$. In PDDL, Figure 4, this is modeled by the two actions `move1` and `move2` which can be executed from any state (i.e. no preconditions). The initial state definition for PDDL is not shown here.

Handling Multiple Initial States If the SMV model in Figure 1 did not have the line `init(x) := s0`, this

```
MODULE main
VAR
   x : {s0,s1};
ASSIGN
   init(x) := s0;
   next(x) := {s0,s1};
```

Figure 3: A simple SMV model with non-determinism

```
(define (domain main)
(:predicates (s0)
             (s1))
(:action move1
   : effect (and (s1) (not (s0)))
(:action move2
   : effect (and (s0) (not (s1))))
```

Figure 4: PDDL domain emulating non-determinism

```
(define (domain main)
(:predicates (s0)
             (s1)
             (init))
(:action move1
   : precondition (and (s0) (init))
   : effect (and (s1) (not (s0)))
(:action move2
   : precondition (and (s1) (init))
   : effect (and (s0) (not (s1))))
(:action init1
   : precondition (not (init))
   : effect (and (s0) (init)))
(:action init2
   : precondition (not (init))
   : effect (and (s1) (init))))
```

Figure 5: PDDL domain to handle incomplete initial state

would create two initial states, $x = s0$ and $x = s1$, since x will be non-deterministically set initially. To convert this to PDDL, we use the procedure we define above. The resulting PDDL code is shown in Figure 5. In the initial state, all predicates are set to false, and the only applicable actions are `init1` and `init2` which act as transitions from the new initial state to the two possible initial states ($x = s0$ and $x = s1$).

Handling Multiple Variables The previous examples showed models with only one variable: x. Suppose the model in Figure 1 included another variable y which has the same type as x and the same definitions of `init` and `next`. The equivalent PDDL domain has to handle the different combinations of conditions for both variables. The translation is shown in Figure 6.

```
(define (domain main)
(:predicates (xs0) (ys0)
             (xs1) (ys1))
(:action move1
   : precondition (and (xs0) (ys0))
   : effect (and (xs1) (ys1) (not (xs0)) (not (ys0)))
(:action move2
   : precondition (and (xs1) (ys1))
   : effect (and (xs0) (ys0) (not (xs1)) (not (ys1)))
(:action move3
   : precondition (and (xs1) (ys0))
   : effect (and (xs0) (not (xs1)) (ys1) (not (ys0))))
(:action move4
   : precondition (and (xs0) (ys1))
   : effect (and (xs1) (not (xs0)) (ys0) (not (ys1))))
```

Figure 6: PDDL domain for SMV model with more than one variable

Safety Checking

In this section, we describe the approach we adopt for checking arbitrary LTL safety properties. We finish by presenting a preliminary evaluation of our approach.

Approach

Given a verification task $V = (M, \phi, \emptyset)$, where the formula ϕ is a safety property, we look for a finite path π in M starting in an initial state that is a counterexample for the property (i.e., $\pi \not\models \phi$). There is thus a clear analogy between searching for a plan and searching for a finite path, since both of them encode a finite trajectory of states.

Our approach builds on previous work by Edelkamp (2003) and by Baier and McIlraith (2006). Edelkamp (2003) casts the verification of simple safety properties $\mathbf{G}\psi$, where ψ is a propositional formula, as a planning task. The planning task is formulated as one in which the objective is to find a plan for the negation of the property: $\mathbf{F}\neg\psi$. The achievement of $\mathbf{F}\neg\psi$ is encoded as a *classical*, final state goal $\neg\psi$, and a planner simply plans for $\neg\psi$. Any plan for the goal $\neg\psi$ eventually achieves $\neg\psi$ and thus is a counterexample for the original property.

Our approach coincides with Edelkamp's in that we translate the model of the system into a partial planning problem in PDDL, and then consider as a goal the negation of the safety property. However, as opposed to Edelkamp's approach, ours can handle *any* LTL safety property.

Given a partial planning problem P^d_M for a system description M, as described in previous section, we define the goal for P^d_M as $\neg\phi$, where ϕ is an arbitrary LTL safety property. Unlike Edelkmap, the resulting planning problem is not a classical planning problem, and thus we require an additional step to generate a classical task. To that end, we use Baier and McIlraith's approach (2006) to transform the problem into an equivalent classical problem (i.e., one in which the goal is represented as a property of the final state).

Theorem 3 establishes the connection between verifying a safety property ϕ in a Kripke structure M and finding a plan to the classical planning problem generated by Baier and McIlraith's approach (2006) as described above.

Proposition 1 *Given a partial planning problem P^d_M and a TEG ϕ, there exists a plan for P^d_M and ϕ iff there exists a plan for the classical planning problem P^c_M obtained from P^d_M and ϕ using Baier and McIlraith's translation (2006). (This follows trivially from (Baier and McIlraith 2006))*

Theorem 3 *Let $V = (M, \phi, \emptyset)$ be a verification task, where ϕ is a safety property, and $\mathcal{P}_M$ be the family of partial planning problems corresponding to M. Let $\mathcal{P}^c_M$ be the family of classical planning problems corresponding to $\mathcal{P}_M$ with the TEG $\neg\phi$ using Baier and McIlraith's translation (2006). Then: $M \models \phi$ iff there doesn't exist $P \in \mathcal{P}^c_M$ that has a plan.*

Proof sketch: Following (Alpern and Schneider 1987), since ϕ is a safety property, it has a finite counterexample. So if $M \models \phi$, then there does not exist a finite path π in M starting from an initial state such that $\pi \not\models \neg\phi$. From Theorem 1, this means that there is no plan in the family of partial planning problems that corresponds to such a path. And therefore, following Proposition 1, there is no plan for any $P \in \mathcal{P}^c_M$. □

Figure 7: Safety checking via temporally extended goals

Consider the example in Figure 7. It represents a Kripke structure with the propositions p, q, and r. In the initial state, marked with double circles, only p is true. To check if the safety property $\mathbf{XG}p$ holds on this model, we convert the model to a planning problem and find a plan satisfying the negation of the property. So we will plan for the TEG $\mathbf{XF}\neg p$. In this case, the planner returns a plan from the initial state p to the state labeled r. So a counterexample of $\mathbf{XG}p$ is the path p, pq, r.

Results

Figure 8 shows experimental results of checking safety properties using the FF planner (Hoffmann and Nebel 2001) and the NuSMV model checker (Cimatti et al. 2002). We translated the dining philosophers problem from SMV to PDDL and verified 3 properties. We performed our experiments under Mac OS X running on a 1.83GHz Core Duo processor with 2GB of RAM.

FF performed significantly better when the properties were false. It was able to find counterexamples even for the 30 philosophers instance in less than the allotted 250 seconds. In the case of the true property p_3, NuSMV solves more instances than FF. This is due to the fact that FF is biased towards finding a counterexample (plan). Initially it starts with the Enforced Hill Climbing (EHC) algorithm (Hoffmann and Nebel 2001) and once EHC fails, it resorts to the exhaustive best first search. This two stage search and the explicit representation of the state space leads FF to be less efficient at checking true properties than NuSMV that represents the state space symbolically.

Liveness Checking and Fairness Constraints

In the previous section we proposed a method for finding finite-length counterexamples of arbitrary LTL safety properties using planning. In this section, we address the more challenging problem of finding counterexamples of *infinite* length using planning. As previously noted, infinite counterexamples are needed when falsifying liveness properties and when fairness constraints are present.

The planning problems we generate with the approach we developed above for finite counterexamples is not suitable for finding infinite counterexamples. Since in such problems we cannot aim at finding an infinite plan that represents an infinite counterexample, we are bound to aim at finding "lasso-shaped plans"; i.e., plans whose execution will visit some state twice. The goal condition for finding such a plan cannot be expressed as a TEG however. Even worse, such a goal is not expressible in terms of the predicates of the domain. Indeed, such a goal requires to refer to the existence of a loop in the states visited by the plan – a condition that cannot be expressed in terms of state predicates. This second alternative has another problem: planners explicitly *prune* from the search space those plans that visit a state twice.

Property	p_1		p_2		p_3	
#Phils	NuSMV	FF	NuSMV	FF	NuSMV	FF
2	0.033	0.01	0.054	0.01	0.027	0.01
3	0.105	0.01	0.138	0.02	0.30	0.09
4	0.668	0.02	1.011	0.04	0.0649	0.85
5	5.172	0.04	8.846	0.07	0.382	7.60
6	25.224	0.07	60.853	0.12	1.600	162.06
7	146.810	0.12	248.358	0.20	8.998	-
8	-	0.19	-	0.30	42.164	-
9	-	0.31	-	0.43	167.649	-
10	-	0.45	-	0.59	-	-
11	-	0.67	-	0.84	-	-
12	-	0.97	-	1.12	-	-
13	-	1.36	-	1.46	-	-
14	-	1.88	-	1.94	-	-
15	-	2.57	-	2.48	-	-
16	-	3.35	-	3.18	-	-
17	-	4.47	-	3.89	-	-
18	-	5.91		4.81	-	-
19	-	7.61	-	5.79	-	-
20	-	9,76	-	7.20	-	-
21	-	12.44	-	8.57	-	-
22	-	15.55	-	10.72	-	-
23	-	19.43	-	11.97	-	-
24	-	24.03	-	14.15	-	-
25	-	29.60	-	16.57	-	-
26	-	37.23	-	19.43	-	-
27	-	45.06	-	24.00	-	-
28	-	54.21	-	27.87	-	-
29	-	64.65	-	33.63	-	-
30	-	83.96	-	41.33	-	-

Figure 8: Time in seconds for using NuSMV and FF to check the deadlock-freeness properties p_1 and p_3, and the property $p_2 = \mathbf{G}(\text{all philosophers done} \rightarrow \mathbf{X}token \neq 0)$. p_1 and p_3 are the same property, but p_1 was verified on a model with a deadlock and p_3 on a deadlock free one. The $token$ signals which philosopher can perform an action. When $token = 0$, the token is not with any philosopher. '-' means that the planner or model checker did not return a result during the 250 second allotted execution time.

The reason for this is efficiency, as visiting a state twice always implies that the plan can be made shorter by removing the chunk of actions that produce the state repetition.

Below we present an approach that can be used to find infinite counterexamples by finding finite plans. We achieve this by considering an enhanced planning problem containing additional predicates which allow to address the issues that we have sketched above. Building upon Schuppan and Biere's approach (2004), we construct an enhanced planning problem that allows referring to the existence of a loop and the existence of a counterexample as a classical planning goal. The approach works for ω-regular properties, and therefore arbitrary LTL properties.

From Liveness to Safety

Schuppan and Biere (2004) showed how liveness properties can be reduced to safety properties in finite state systems. Their approach modifies the Kripke structure by adding "temporary stores" for the state variables and additional state transitions to find lasso-shaped paths. For simplicity, we view atomic propositions as *state variables* that hold a value of true or false.

The following gives a general overview of the state recording translation. The approach uses the negation of the Büchi automaton $B^{\neg\phi}$ of the LTL property ϕ to be checked and takes the product of the Kripke structure and $B^{\neg\phi}$ (Vardi and Wolper 1986).

1. For each state variable v we consider an additional variable v_s, intended to save the value of v.
2. The value of all variables is saved by a special $save$ event, which corresponds to a nondeterministic transition that saves the values of the state variables in the current state in the copies of state variables. Once a save event occurs, no further saves are allowed and we assume that we are on the loop of a lasso-shaped path. (Nondeterministic save)
3. The product of the modified Kripke structure (with temporary-stores and non-deterministic save) and $B^{\neg\phi}$ is taken. (Property observation)
4. The loop closes once the current state is equal to the saved state. (Loop detection)
5. If the loop closes and each accepting state of $B^{\neg\phi}$ occurs on the loop then the path is considered a counterexample.

Note that items 1 to 3 define the encoding of a new Kripke structure M_s in which we want to find a counter example that is a lasso-shaped path that visits an accepting state of $B^{\neg\phi}$. Schuppan and Biere (2004) showed that there exists a *safety* property for this new Kripke structure whose counter examples have precisely this form. Specifically, they construct an LTL safety property ϕ_s such that ϕ_s is violated by M_s iff ϕ is violated by M. As such, we can use the method described in the previous section to use planning for checking this new property in this new Kripke structure.

Theorem 4 *Let $V = (M, \phi, \emptyset)$ be a verification task, where ϕ is an arbitrary LTL property. Let $V_s = (M_s, \phi_s, \emptyset)$ be the translation of V using Schuppan and Biere's approach (2004) to safety checking, where ϕ_s is a safety property. Let $\mathcal{P}_{M_s}$ be the family of partial planning problems corresponding to M_s. Then:*
$M \models \phi$ iff there does not exist a $P \in \mathcal{P}_{M_s}$ that has a plan for the TEG $\neg\phi_s$

Proof sketch: We can establish that $M \models \phi$ iff $M_s \models \phi_s$ from Theorem 5 in (Schuppan 2006) and the fair loop detection they borrow from (Vardi and Wolper 1986). The rest follows from Theorem 3. □

Even though our theorem is stated in terms of our theoretical translation, which can be exponential in the number of propositions, that given a pragmatic PDDL domain description of a domain, one can efficiently generate another compact PDDL description that models Schuppan and Biere's translation (2004). In fact, the save event can be modelled by a new action, the copies of the variables as new predicates. Finally, the Büchi automaton for $\neg\phi$ can be modelled within the planning domain following either methods proposed by Edelkamp (2006) or by Baier and McIlraith (2006). This way we can represent the goal condition (which talks about visiting states of $B^{\neg\phi}$) using the predicates of the planning domain. In the next section we illustrate how we perform this translation for an example program.

The Planning Problem in PDDL We adopt an approach similar to Schuppan and Biere's by adding loop detection

Figure 9: Liveness to safety reduction

and property observation directly to the PDDL domain and problem files.

Consider the finite state machine in Figure 9. It represents the SMV model in Figure 1 and the PDDL domain in Figure 2. Suppose we want to check the property **GF**$s0$. In the Büchi automaton of **GF**$s0$, any occurrence of $s0$ takes the automaton to an accepting state, and any occurrence of $s1$ takes it to an unaccepting state. Therefore, we will be looking for a lasso-shaped counterexample where $s0$ never occurs on the loop so as not to visit an accepting state infinitely many times. The following shows how we encode this in PDDL:

1. Add predicates $ts0$ and $ts1$. (Copies of state variables)
2. Add a save action that takes values of $s1$ and $s2$ and places them in $ts0$ and $ts1$ respectively. This action is such that it can at most be performed once in every legal plan. (Nondeterministic save)
3. Add predicates $loop$, $save$, and $live$.
4. Modify actions to check whether $live$ becomes true in the loop. $live$ becomes true if $s0$ occurs on the loop. (Property observation)
5. Loop closure is added as the goal. (Loop detection)

The resulting domain and problem files are shown in Figures 10 and 11. $save$ signals the beginning of the loop and then $loop$ signals the rest of the loop. The safety property to be verified becomes in this case $\mathbf{G}(looped \rightarrow live)$, where $looped$ is a propositional formula refering to the loop closing condition (the saved predicates are equal to the current predicates). So the planning problem will have the goal $looped \wedge \neg live$ that indicates that we are looking for a lasso-shaped path where the $s0$ does not occur on the loop.

The loop detection part is always the same regardless of the liveness property being verified. The property observation part changes depending on the property. For instance, in the case of **GF**p, the counterexample should have p false on the loop states, whereas for **F**p the counterexample should have p false in all states of the lasso shaped counterexample.

Fairness Constraints Fairness constraints are properties that must hold infinitely often on a path. For example, in figure 9 any infinite path will pass through $s0$ infinitely often. Therefore, if $s0$ was a fairness constraint, it would hold on any infinite path in Figure 9. To apply a fairness constraint f, where f is a propositional formula representing a set of states, we only consider infinite paths satisfying the property **GF**f (i.e., in lasso-shaped paths, there is at least one state on the loop where f holds).

To add the fairness constraint $s1$ to the example in figures 10 and 11, we have to check if $s1$ occurs on the loop of a counterexample for it to be considered valid. The resulting PDDL domain and problem descriptions are shown in the Appendix. If more than one fairness constraint was included, each one of them has to occur on the loop of a lasso-shaped counterexample.

```
(define (domain example)
(:predicates (s0)
             (s1))
             ; Added predicates
             (ts0)
             (ts1)
             (live)
             (save)
             (loop))

(:action move1
   : precondition (s0)
   : effect (and (s1) (not (s0))
   ; Property Observation
      (when (and (loop) (not (s0))) (not (live)))
      (when (and (save) (not (s0))) (not (live)))
      (when (save) (and (loop) (not (save)))))

(:action move2
   : precondition (s1)
   : effect (and (s0) (not (s1)))
   ; Property Observation
      (when (and (loop) (not (s0))) (not (live)))
      (when (and (save) (not (s0))) (not (live)))
      (when (save) (and (loop) (not (save)))))

(:action save
   : precondition (and (not (loop)) (not (save)))
   : effect (and (save)
      (when (s0) (ts0))
      (when (s1) (ts1))))
```

Figure 10: PDDL domain for checking **FG**$s0$

```
(define (problem prob)
(:domain (example))
(:init (s0) (live))
(:goal (and (not (live)) (loop)
   ; Loop detection
   (or (and (s0) (ts0)) (and (not (s0)) (not (ts0))))
   (or (and (s1) (ts1)) (and (not (s1)) (not (ts1))))))
```

Figure 11: PDDL problem for checking **FG**$s0$. Loop detection is stated as part of the problem

Results

We verified the false liveness property **GF**$(token = 0)$ on the dining philosophers problem with the fairness constraints that each philosopher executes. The results are shown in Figure 12. FF showed a significant improvement over NuSMV, taking 2.5 seconds to find a counterexample in the 30 philosophers instance.

Summary and Future Work

In this paper, we demonstrated that heuristic search planning techniques have the potential to outperform model checkers with respect to a diversity of automated verification tasks. A key enabler of our work, and one that distinguishes it from previous related work by Edelkamp, is the ability to exploit heuristic search planning for goals expressed as arbitrary LTL formulae – TEGs. We do so by appealing to a translation proposed by Baier and McIlraith that transforms an LTL formula into a NFSA whose accepting condition is

#Phils	NuSMV	FF	#Phils	NuSMV	FF
2	0.01	0.01	15	-	0.38
3	0.05	0.01	16	-	0.48
4	0.23	0.02	17	-	0.55
5	2.07	0.03	18	-	0.64
6	20.19	0.04	19	-	0.74
7	153.94	0.06	20	-	0.85
8	-	0.08	21	-	0.95
9	-	0.10	22	-	1.09
10	-	0.14	23	-	1.21
11	-	0.17	24	-	1.37
12	-	0.20	25	-	1.52
13	-	0.26	26	-	1.70
14	-	0.34	27	-	1.89
15	-	0.38	28	-	2.08
16	-	0.48	29	-	2.29
17	-	0.55	30	-	2.52

Figure 12: Time in seconds for using NuSMV and FF to check the property $\mathbf{GF}(token = 0)$ with the fairness constraints that all philosophers execute on the dining philosophers problem. '-' means that the planner or model checker did not return a result during the 250 seconds allotted execution time.

the satisfaction of the LTL formula. This NFSA is then encoded in the planning domain, creating a classical planning problem for which heuristic search techniques can be applied.

Although this work is still in its preliminary stages, the contributions of this paper are both theoretical and experimental in nature. In particular, we show how to characterize finite-state verification problems as planning problems, providing a translation from a subset of SMV to PDDL. To check arbitrary safety properties we search for finite counterexamples by attempting to construct a plan for the negation of the property. Since safety properties are expressed in LTL, a property's negation is also an LTL formula, which we characterize as a TEG. The Baier-McIlraith translation then allows us to translate this to a classical planning problem for which highly-efficient heuristic search can be applied. If no counterexample is found, then there is no state violating the invariant that is reachable from the initial state(s), and therefore the property holds.

The verification of liveness properties and fairness conditions proved more challenging because they necessitated finding an infinite counterexample. To address these tasks, we appealed to an approach proposed by Schuppan and Biere (2004) to reduce liveness checking to safety checking. The approach non-deterministically chooses a state to be the start of a loop, and then tries to find a path that would return to that state while not satisfying the liveness property. For example, to verify if the property $\mathbf{F}p$ holds on a model, we check if there is a path where p never occurs on any state and the path ends in a loop (i.e. an infinite path).

To evaluate the effectivenss of our proposed approaches, we performed experiments with a parametrized instance of the dining philosophers problem. Each verification problem was translated to a classical planning problem following the proposed approach and we compared the performance of the FF planner to the performance of the NuSMV model checker on the same problem. FF performed significantly better than NuSMV, finding deadlocks in problems with 30 philosophers while NuSMV was only able to find counterexamples for up to and including 7 philosophers. With respect to verification of safety properties, FF again significantly outperformed NuSMV on false properties, solving all 30 instances to NuSMV's 7, with respect to two different properties. However, NuSMV performed slightly better with true properties, solving 3 more instances than FF.

Our experimental evaluation illustrates the promise of our approach to verification. In the near future, we plan to perform more experimental evaluation. In particular, we hope to experiment with several other verification domains, and also to benchmark our techniques against explicit state model checkers. Due to the non-deterministic save when verifying liveness, the planner is essentially guessing where a loop might start. This leads heuristic search to be ineffective when choosing the state to save. We are working on modifying search heuristics to guide the planner towards finding loops.

References

Alpern, B., and Schneider, F. B. 1987. Recognizing safety and liveness. *Distributed Computing* 2:117–126.

Bacchus, F., and Kabanza, F. 1998. Planning for temporally extended goals. *Annals of Mathematics and AI* 22(1-2):5–27.

Baier, J. A., and McIlraith, S. A. 2006. Planning with first-order temporally extended goals using heuristic search. In *Proc. AAAI*, 788–795.

Cimatti, A.; Clarke, E.; Giunchiglia, E.; Giunchiglia, F.; Pistore, M.; Roveri, M.; Sebastiani, R.; and Tacchella, A. 2002. NuSMV Version 2: An OpenSource Tool for Symbolic Model Checking. In *Proc. International Conference on Computer-Aided Verification (CAV 2002)*, volume 2404 of *LNCS*. Copenhagen, Denmark: Springer.

Clarke, E. M.; Grumberg, O.; and Peled, D. A. 2000. *Model Checking*. The MIT Press.

Edelkamp, S.; Lafuente, A. L.; and Leue, S. 2001. Directed explicit model checking with hsf-spin. In *In Proceedings of the 8th International SPIN Workshop on Model Checking of Software*, 57–79. Springer-Verlag.

Edelkamp, S. 2003. Promela planning. In *Proceedings of SPIN-03*, 197–212.

Edelkamp, S. 2006. On the compilation of plan constraints and preferences. In *Proc. ICAPS*, 374–377.

Fox, M., and Long, D. 2003. PDDL2.1: An extension to PDDL for expressing temporal planning domains. *Journal of AI Research* 20:61–124.

Ghallab, M.; Nau, D.; and Traverso, P. 2004. *Automated Planning Theory and Practice*. Elsevier.

Hoffmann, J., and Nebel, B. 2001. The FF planning system: Fast plan generation through heuristic search. *Journal of AI Research* 14:253–302.

Holzmann, G. J. 1997. The model checker SPIN. *Software Engineering* 23(5):279–295.

Lamport, L. 1977. Proving the correctness of multiprocess programs. *IEEE Trans. Softw. Eng.* 3(2):125–143.

McDermott, D. V. 1998. PDDL — The Planning Domain Definition Language. Technical Report TR-98-003/DCS TR-1165, Yale Center for Computational Vision and Control.

McMillan, K. L. 1992. Symbolic model checking: an approach to the state explosion problem. Technical Report CMU-CS-92-131, CMU.

Schuppan, V., and Biere, A. 2004. Efficient reduction of finite state model checking to reachability analysis. *STTT* 5(2-3):185–204.

Schuppan, V. 2006. *Liveness Checking as Safety Checking to Find Shortest Counterexamples to Linear Time Properties*. Ph.D. Dissertation, ETH Zürich.

Vardi, M. Y., and Wolper, P. 1986. An automata-theoretic approach to automatic program verification. In *Proceedings of the 1st Annual Symposium on Logic in Computer Science (LICS'86)*, 332–344. IEEE Comp. Soc. Press.

Appendix: PDDL Domain and Problem with Fairness Constraints

Figures 13 and 14 show the PDDL domain and problem files, respectively, for checking the property **FG**s_0 with the fairness constraint s_1 for the state machine shown in figure 9.

```
(define (domain example)
(:predicates (s0)
             (s1))
             ; Added predicates
             (ts0)
             (ts1)
             (live)
             (save)
             (loop)
             (fair))

(:action move1
   : precondition (s0)
   : effect (and (s1) (not (s0))
   ; Fairness constraints
      (when (and (loop) (s1)) (fair))
      (when (and (save) (s1)) (fair))
   ; Property observation
      (when (and (loop) (not (s0))) (not (live)))
      (when (and (save) (not (s0))) (not (live)))
      (when (save) (and (loop) (not (save)))))

(:action move2
   : precondition (s1)
   : effect (and (s0) (not (s1)))
   ; Fairness Constraints
      (when (and (loop) (s1)) (fair))
      (when (and (save) (s1)) (fair))
   ; Property observation
      (when (and (loop) (not (s0))) (not (live)))
      (when (and (save) (not (s0))) (not (live)))
      (when (save) (and (loop) (not (save)))))

(:action save
   : precondition (and (not (loop)) (not (save)))
   : effect (and (save)
      (when (s0) (ts0))
      (when (s1) (ts1))))
```

Figure 13: PDDL domain for checking **FG**s_0 with fairness constraint s_1

```
(define (problem prob)
(:domain (example))
(:init (s0) (live))
(:goal (and (not (live)) (loop) (fair)
   ; Loop detection
   (or (and (s0) (ts0)) (and (not (s0)) (not (ts0))))
   (or (and (s1) (ts1)) (and (not (s1)) (not (ts1))))))
```

Figure 14: PDDL problem for checking **FG**s_0. Loop detection is stated as part of the problem. $fair$ is the fairness constraint that has to hold on the counterexample (plan)