Singapore Management University

Institutional Knowledge at Singapore Management University

Research Collection School Of Computing and Information Systems | School of Computing and Information Systems

7-2012

Translating PDDL into CSP# - The PAT approach

Yi LI

Jing SUN

Jin Song DONG

Yang LIU

Jun SUN
Singapore Management University, junsun@smu.edu.sg

Follow this and additional works at: https://ink.library.smu.edu.sg/sis_research

Part of the Programming Languages and Compilers Commons, and the Software Engineering Commons

Citation

LI, Yi; SUN, Jing; DONG, Jin Song; LIU, Yang; and SUN, Jun. Translating PDDL into CSP# - The PAT approach. (2012). *Proceedings of the 17th IEEE International Conference on Engineering of Complex Computer Systems, ICECCS 2012, Paris, France, July 18-20*. 240-249.
Available at: https://ink.library.smu.edu.sg/sis_research/5019

This Conference Proceeding Article is brought to you for free and open access by the School of Computing and Information Systems at Institutional Knowledge at Singapore Management University. It has been accepted for inclusion in Research Collection School Of Computing and Information Systems by an authorized administrator of Institutional Knowledge at Singapore Management University. For more information, please email cherylds@smu.edu.sg.

Translating PDDL into CSP# – the PAT Approach

Yi Li*, Jing Sun†, Jin Song Dong‡, Yang Liu‡ and Jun Sun§
*Department of Computer Science, University of Toronto, Canada
Email: liyi@cs.toronto.edu
†Department of Computer Science, The University of Auckland, New Zealand
Email: j.sun@cs.auckland.ac.nz
‡School of Computing, National University of Singapore, Singapore
Emails: {dongjs, liuyang}@comp.nus.edu.sg
§Singapore University of Technology and Design, Singapore
Email: sunjun@sutd.edu.sg

***Abstract*—Model checking provides a way to automatically verify hardware and software systems, whereas the goal of planning is to produce a sequence of actions that leads from the initial state to the desired goal state. Recently research indicates that there is a strong connection between model checking and planning problem solving. In this paper, we investigate the feasibility of using a newly developed model checking framework, Process Analysis Toolkit (PAT), to serve as a planning solution provider for upper layer applications. We first carried out a number of experiments on different planning tools in order to compare their performance and capabilities. Our experimental results showed that the performance of the PAT model checker is comparable to that of state-of-art planners for certain categories of problems. We further propose a set of translation rules for mapping from a commonly used planning notation – PDDL into the CSP# modeling language of PAT. Finally, we provide evaluations on the translated models against other approaches in the planning domain to demonstrate the effectiveness of using the PAT model checker for planning.**

***Keywords*-Formal Verification; Model Checking; Planning.**

I. INTRODUCTION

Model checking [1] is an automatic technique for verifying models of software or hardware systems against their specification. The system model is exhaustively explored and checked by model checkers to ensure that desired properties are guaranteed in all cases. In general, what we care the most about the system model is whether some safety or liveness properties, usually described in temporal logics such as *Linear Temporal Logic (LTL)* and *Computation Tree Logic (CTL)*, are satisfied. Given a system model $\mathcal{M}$, an initial state s, and a formula φ which specifies the property, the model checking process can be viewed as computing an answer to the question of whether $\mathcal{M}, s \models \varphi$ holds. Invariant which can be expressed using LTL formula ($\mathbf{G}\neg p$) is an example of safety properties, where $\mathbf{G}$ reads as always. Typically, a counterexample is given by model checkers when the property is found to be violated.

Model checking has emerged as a promising and powerful approach to automatically verify software and hardware systems. Recently, research indicates that model checking can also be applied to the AI planning domain. Berardi and Giacomo [2] compared the performance of two well-known model checkers, Spin [3] and SMV [4], with some state-of-the-art planners (IPP [5], which was one of the best performers in AIPS'98 competition; FF [6], which was among the best performers in AIPS'00; and TLPLAN [7], which accepts temporally extended goals used as control knowledge to prune the search space). The experiment results suggest that the two model checkers are comparable to IPP in terms of performance, instead that FF performs much better than both. In other words, Spin and SMV used as planners are competitive with the best performing planners at the AIPS'98 competition. There is still large space for improvement in solving planning problems using model checkers. Spin can indeed improve its performance by exploiting additional control knowledge, which consists of suitable constraints on state transitions and thus can be used to reduce the state space explored during searching.

Hörne and Poll [8] investigated the feasibility of using two different model checking techniques for solving a number of classical AI planning problems. The two model checkers use different reasoning techniques. ProB is based on mathematical set theory and first-order logic. It is specifically designed for the verification of program specifications written in the B specification language. The other model checker used is NuSMV [9], an extension of the symbolic model checker SMV. With NuSMV the problem is represented using Binary Decision Diagrams (BDDs) [10]. For both model checkers, the state space is explored exhaustively: if there exists a plan, it will be found, and they always terminate. However, they do not provide all possible plans but terminate after one is found, if it exists. The experiment results suggest that several options were found suitable to solve the type of planning problems considered in the paper. These are the Constraint Logic Programming (CLP) based ProB, running in either temporal model checking mode or performing a breadth-first search, and the tableaux-based NuSMV using an invariant.

Another source of interest for this topic is that with the

978-2-9541-8100-4/12 $26.00 © 2012 IEEE
DOI 10.1109/ICECCS.2012.41

IEEE computer society

capability of solving planning problems, model checkers can be used as an underlying service provider to provide planning solutions for upper layer applications. Newly developed model checkers usually have more sophisticated techniques for handling large state spaces, which is critical in the real world setting. Therefore, using model checking as service should work well for real world planning problems, such as trip planning, scheduling, etc. In this paper, we further explore the synergy between the two separate domains, namely *model checking* and *planning*. They are both important techniques used in system designs. For example, one can obtain a workable design under the environment and resource constraints via planning and verify that the required properties are all satisfied by model checking. Our goal is to find a way to connect them together such that the tools that support model checking can also be used to find solutions for planning problems.

In this paper, we consider classical planning problems that have only deterministic actions and assume complete information about the planning states. Essentially following [11], we define a classical planning problem to be a three-tuple (S_0, G, A) where S_0 represents the initial state, G represents the set of goal states and A represents a finite set of deterministic actions. Each *state* is represented as a conjunction of fluents that are ground, functionless atoms. Each *action* $a \in A$ itself is described by a tuple $(pre(a), add(a), del(a))$ where $pre(a)$ represents the precondition to be satisfied before the action can be executed, $add(a)$ and $del(a)$ represent the positive and negative effects after the action is executed. Therefore the state resulting from executing action a in state s can be expressed as $Result(s, a) = (s - del(a)) \bigcup add(a)$. Finally, the goal G is a set of planning states satisfying a propositional property specifying the final states of a plan. Therefore, a plan p is a finite sequence of actions $\langle a_0, a_1, ..., a_n \rangle$, such that the execution of p yields a state $s \in G$.

The Planning Domain Definition Language (PDDL) [12] is currently the standard language for representing classical planning problems and is widely used by many planners. Actions are grouped as a set of action schemas in PDDL. The schema consists of the action name, a list of all the variables used in the schema, a precondition and an effect.

```
(:action TakeBus
  :parameters (?p ?b ?from ?to)
  :precondition (and At(b,from)
     At(p,from) Bus(b) Passenger(p)
     Stop(from) Stop(to))
  :effect (and
      (not At(p,from) At(b,from))
      At(b,to) At(p,to)))
```

The PDDL code above is an example of an action schema for taking a bus from a bus stop *from* to another bus stop *to*. The precondition for the action schema is that both the bus and the passenger are at *from* and the effect is that they are transferred to a new location *to*. In the later extensions of PDDL, such as PDDL 2.1 where typing system is added, the type predicates like *Bus(b)* is not needed anymore. PDDL 2.1 also allows for optimization criteria to be specified. The optimization criterion, also called *plan metric*, consists of numerical expressions to be maximized or minimized.

Clearly, a classical planning problem can be easily converted into a model checking problem. The fact that this approach is feasible and supported by [13], which states that, planning should be done by semantically checking the truth of a formula, planning as model checking is conceptually similar to planning as propositional satisfiability. Given a planning problem (S_0, G, A), one can construct a system model $\mathcal{M}$ by translating every action $a \in A$ into a corresponding state transition function first. The initial state S_0 can also be mapped to the initial state s of model $\mathcal{M}$ by assigning value to each variable accordingly. Then for the goal state G, which can be expressed using a propositional formula φ, we can construct a safety property $\mathbf{G}\neg\varphi$ that requires the formula φ never to hold, such that the model checker is able to search for a counterexample path that leads to a state where φ holds. The resulting plan is optimal in terms of make-span when the counterexample path is the shortest. We shall discuss the detailed translation process in section 3.

This research is divided into two stages, corresponding to the two closely related problems that we considered, i.e., *planning via model checking* and *PAT as planning service*. We first conducted a number of experiments on different planning domains in order to compare the performance and capabilities of various tools. Our experimental results indicate that the performance of some model checkers is comparable to that of state-of-the-art planners for certain categories of problems and the performance of model checking can even be further improved by exploiting domain-specific knowledge. In particular, a newly developed model checking framework - Process Analysis Toolkit (PAT) out-performs most of the existing tools in the problem domain. We further investigated the possibility of developing a new planning module with specifically designed searching algorithm on top of the PAT framework, to serve as a planning solution provider for upper layer applications. We present a set translation rules that maps the commonly used planning notation – PDDL into its corresponding models in the CSP# language of PAT, where evaluations have been conducted on the translated models.

The rest of the paper is organized as follows. Section 2 presents the review on performance of different planning tools. In Section 3, we introduce the idea of using the PAT model checker for planning by translating the PDDL domain descriptions into system models in CSP# which is supported by the tool. Section 4 presents the evaluation on performance of the translated model in comparison to those of other

planning tools. Section 5 concludes the paper and outlooks the future directions.

II. Review on model checking tools for planning

In this section, we conduct a performance review on three commonly used model checkers together with two well-known planners as benchmarks in solving planning problems. A background description of the tools investigated are listed as follows.

A. NuSMV

NuSMV is an extension of the symbolic model checker SMV [4] developed at the Carnegie Mellon University known as CMU SMV. Like CMU SMV, NuSMV uses the CUDD-based BDD package, a state-of-the-art BDD package developed at Colorado University. During model construction, NuSMV builds a clusterised BDD-based Finite State Machine (FSM) using the transition relation. A model is described in terms of a hierarchy of modules. Module instantiations are semantically similar to call-by-reference. NuSMV allows for Boolean, integer and enumerated types for state variables [9]. However, array indices in NuSMV must be statically evaluated to integer constants. This constraint largely limits the expressiveness of the model. The modelling for common operations on a list of state variables is sometimes cumbersome in NuSMV. In general, such operations have to be manually coded by enumerating all the possible cases.

The descriptions of transition relations between the current and next state pairs can be done by either using the `ASSIGN` constraint where a system of equations labelled as `next(identifier):=expression` describing how the FSM evolves over time, or the `TRANS` constraint [9]. Specifications can be expressed in both CTL and LTL. NuSMV supports several kinds of model checking modes, namely CTL checking, LTL checking, invariant checking and bounded model checking. We will compare the performance of using different model checking modes for planning in the later section.

B. Spin

Spin is an established explicit state model checker developed at Bell Labs in the original Unix group of the Computing Sciences Research Center, starting in 1980. Spin models are described in a modelling language called "Promela" (Process Meta Language). The language allows for the dynamic creation of concurrent processes. Communication via message channels can be defined to be synchronous or asynchronous [3]. Promela loosely follows CSP and hence our models in CSP# can be converted to it with minimal efforts. Guarded expressions are well supported, so that preconditions for actions can be easily enforced in the model. Promela also allows C-style macro definitions, which reduces the code length and facilitates the generalization of the model.

Spin has a number of runtime options for simulation as well as verification that can be explored. The maximum search depth can be adjusted according to the size of the model. Spin also allows users to prune the search space using "never-claims" which are equivalent to safety properties. With this method it becomes possible to verify quickly whether a given safety property holds in the context of the model, even when a complete verification is considered to be infeasible [3]. After verification is finished, Spin is able to perform a simulation guided by the error trail. In simulation mode, step-by-step display of the counterexample trace is better supported by its user interface compared with that of NuSMV.

The specifications of properties can also be written in LTL and Spin will translate the formulas into "never-claims" and perform the verification. However, the counterexamples produced by Spin are not guaranteed to be in the minimum size, so we are not able to produce shortest plans using Spin.

C. PAT

Process Analysis Toolkit (PAT) [14] is a self-contained framework for specification, simulation and verification of concurrent and real-time systems developed in School of Computing, National University of Singapore. It supports efficient trace refinement checking, LTL model checking with various fairness assumptions. PAT is designed to verify event-based compositional models specified using CSP# [15], which is an extension to Communicating Sequential Process (CSP) [16] by embedding data operations. CSP# combines high-level compositional operators from process algebra with program-like codes, which makes the language much more expressive.

One of the unique features of PAT is that it allows users to define static functions and data types as C# libraries. These user defined C# libraries are built as DLL files and are loaded during execution. This makes up for the common deficiencies of model checkers on complex data operations and data types. For instance, priority queue and set can be implemented to meet the need of models that deal with special algorithms.

PAT is designed as an flexible and modularized framework. It allows users to build customized model checking modules easily. The language syntax, semantics, model checking algorithms, reduction techniques, and abstraction techniques can all be tailored for a specific domain. We shall explore this feature later in section 4 to customize searching algorithms for planning purpose. PAT also has a more user-friendly user interface both for verification and simulation compared with other tools that we have experimented on.

D. Metric-FF

Metric-FF [17] is a domain independent planning system developed by Jörg Hoffmann. It is an extension of FF that

supports numerical plan metrics. The system has participated in the numerical domains of the 3rd International Planning Competition, demonstrating very competitive performance. Two input files, namely the domain file and problem file are needed to run Metric-FF. Metric-FF accepts domain and problem specifications written in PDDL 2.1 level 2. As mentioned, PDDL 2.1 allows numerical plan metrics. The following shows an example of plan metrics used in domain descriptions.

```
(:action TakeBus
  :parameters (?p - passenger ?b - bus
    ?from - stop ?to - stop)
  :precondition (and (at ?x south)
    (at ?y south))
  :effect (and (not (at ?p from))
    (not (at ?b from)) (at ?p to)
    (at ?b to) (increase (time-cost) 10)
    (increase (money-cost) 2))))
```

Now the parameters have their own types, specified right behind the variable identifiers. We also add in updates of plan metrics *time-cost* and *money-cost* within the effect statement. When the action *TakeBus* is executed, a time cost of 10 and a money cost of 2 will be incurred. Optimization criteria can be identified inside the problem file, with the statement (: `metric` `minimize`(`cost`)) that means the value of the state variable *cost* should be minimized. Note that the *cost* here can also be a linear combination of several variables. We are able to modify the two searching parameters *g* and *h* to assign weights to plan metrics optimization and heuristic functions respectively. By increasing the value of *g*, the system will assign a higher priority to the minimization of the given plan metrics, despite that the returned solutions are not guaranteed optimal.

E. SatPlan

SatPlan [18] is an award winning planner for optimal planning created by Henry Kautz, Jörg Hoffmann and Shane Neph. SatPlan2004 took the first place for optimal deterministic planning at the International Planning Competition at the 14th International Conference on Automated Planning & Scheduling. SatPlan accepts the STRIPS subset of Planning Domain Definition Language (PDDL) and finds plans with the shortest make-span. It encodes the planning problem into a SAT formulation with length *k* and checks the satisfiability using SAT solvers. If the searching times out, then *k* is increased by one and the process is repeated.

In SatPlan, the optimality of plan is restricted to its length or make-span. However, in many cases, especially real life applications, the length of the solution is not the only criterion to be considered. The quality of the plan also depends on other factors. For instance, the quality of the suggested routes produced by a route planning system should be judged by the users' preferences, the total distance of the trip, the total cost of time and money, etc. This kind of problems are often solved by adding non-negative cost to actions, and the goal becomes to find a plan with the minimum total action cost.

F. Performance Comparison

In this subsection, we compare the performance of NuSMV (pre-compiled version 2.5.2), Spin (pre-compiled version 6.0.1) and PAT (3.3.0 academic version) on solving a classic planning problem – *the sliding game problem*. SatPlan2006 and Metric-FF are also used as benchmarks in the experiments. The optimal solution of this puzzle solving problem is not trivial. The descriptions of the problems are as follows.

The sliding game problem, is sometimes also referred as *the eight-tiles problem*. We have eight tiles, numbered from 1 to 8, that are arranged in a 3×3 matrix. The first tile, which is at the top-left corner is empty and marked by 0. A tile can only be shifted horizontally or vertically into the empty space. The goal of the puzzle is to arrange the eight tiles into the setting shown in Figure 1.

0	1	2
3	4	5
6	7	8

Figure 1: Initial setting of *the sliding game problem*

Optimal AI planning is a PSPACE-complete problem in general. For many problems studied in the planning literature, the plan optimisation problem has been shown to be NP-hard [19]. The *eight-tiles game* is the largest puzzle of its type that can be completely solved. It is simple, and yet obeys a combinatorially large problem space of 9!/2 states. The $N \times N$ extension of the *eight-tiles game* is NP-hard [20]. The difficulties of the problem instances are measured by the lengths of their optimal solutions. There is also an approximated measurement named the *Manhattan distance* or *Manhattan length*, which is defined as $\mid x_1 - x_2 \mid + \mid y_1 - y_2 \mid$ where (x_1, y_1) and (x_2, y_2) are two points on a plane. We have experimented on 6 problem instances in total. Two of them ("Hard1" and "Hard2") are the hardest with an optimal solution of 31 steps. Two of them ("Most1" and "Most2") have the most optimal solutions and a slightly shorter solution length of 30 steps. The last two problem instances ("Rand1" and "Rand2") are randomly generated with optimal solutions of length 24 and 20 steps respectively. The initial configurations of all the six problem instances are shown in Figure 2.

This set of experiments are designed to show how different model checkers perform on optimal deterministic planning problems. To collect the execution time data more

Table I: Experimental results for *the sliding game problem*

Problem	L*	H	SatPlan	PAT	NuSMV			Spin
				BFS	INVAR	CTL	LTL	suboptimal
Hard1	31	21	444.42	9.60	45.2	> 600	> 600	2.25
Hard2	31	21	438.34	10.05	41.6	> 600	> 600	2.06
Most1	30	20	152.76	9.84	42.8	> 600	> 600	1.99
Most2	30	20	152.24	10.01	42.0	> 600	> 600	2.47
Rand1	24	12	33.70	7.00	30.0	> 600	> 600	2.63
Rand2	20	16	2.89	3.54	16.8	505.6	> 600	2.13

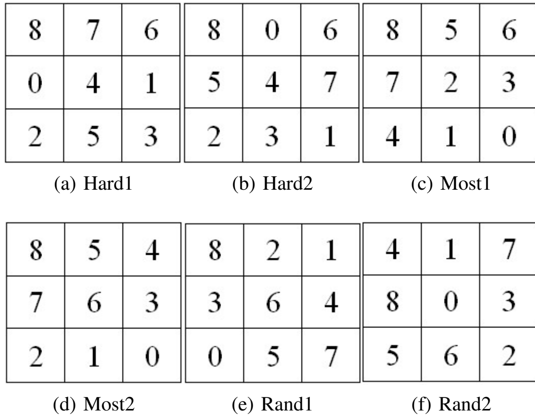

Figure 2: Initial configurations of *the sliding game problem* instances

accurately, we performed each experiment three times and calculated the average to avoid possible fluctuations caused by the overhead imposed by operating systems. To run Spin, we used the Unix simulator Cygwin. Spin displays the execution time in a separate window using an embeded Tcl/Tk environment. PAT and NuSMV were tested in Windows XP SP3, while SatPlan and Metric-FF were tested in Ubuntu 10.04 environment. Except for NuSMV, all other tools provide accurate statistics including the execution time at the end of each session. For NuSMV, we made use of the *source* command to invoke the *time* command right before and after the model checking sessions to record the execution time. Unfortunately, the *time* command in NuSMV provides time data that is accurate to only one decimal place. On contrast, execution time data got from other tools was rounded to two decimal places.

All the experimental results were collected on an Dell desktop with an Intel Core 2 Duo E6550 2.33GHz processor and 3.25GB RAM. The experimental results are presented in Table I, where INVAR denotes using invariant mode of NuSMV, LTL/CTL denotes using LTL/CTL model checking mode of NuSMV, WITH denotes PAT under "reachability-with" mode, and DFS/BFS denotes PAT using depth-first/breadth-first search. Time is in seconds unless otherwise indicated. The results got from SatPlan are used for reference. Inside the table, "> 600" indicates that no solution was found after 10 minutes. The column "L*" records the length of the optimal solutions and the column "H" shows the *Manhattan distance* of the problem. Also note that the solutions found by Spin are not optimal.

The CTL and LTL checking mode of NuSMV can hardly find a solution within 10 minutes. The invariant checking mode performs much better compared to the other two modes. From Figure 3, we can conclude that the execution time of SatPlan for different problem instances varies greatly. The performance of SatPlan depends largely on the length of the optimal plans. "Hard1" and "Hard2" which take only 1 step more than "Most1" and "Most2", spend nearly 3 times longer to find a solution. For simpler instances, SatPlan performs the best among the three tools. However, when the length of the optimal plans increases, the size of the SAT instances created by SatPlan grows fast. The resulting execution time increases quickly as well.

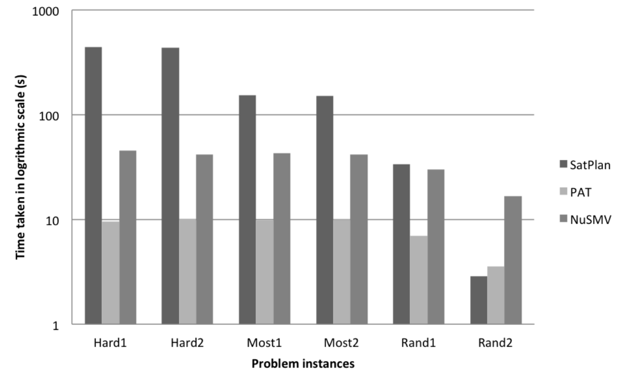

Figure 3: Execution time comparison of PAT, NuSMV and SatPlan on *the sliding game problem*, shown on a logarithm scale

The performance of PAT and NuSMV is relatively stable. PAT using breadth-first search mode takes shorter time for all the problems. This comparison indicates that PAT that belongs to the category of explicit state model checkers performs better than symbolic model checker NuSMV and SAT based planner SatPlan on plan optimization problems. Although we cannot generalize the argument without fur-

ther experiments and justifications, this empirical finding still proves the feasibility of applying PAT to the optimal deterministic planning domain.

III. PAT for Planning – Translating PDDL into CSP#

When performing the experiments in Section 2, we realised that the generalization of the problems should be a priority because the encoding of the planning problems in the respective model description languages is cumbersome. This gives rise to the idea of using model checkers as service. Considering planning problems in more realistic environment, the variables and parameters in the model descriptions are usually subject to change over time. In some cases, the goals and cost/reward functions could also be different when the environment variables vary. This is where the concept of *replan* comes into play. Using model checkers as service enables real-time *replanning* by generating problem descriptions dynamically at runtime, and modifying models with the most updated parameters. However, some modifications to the model checking algorithms are necessary to finally realize this goal. As a newly developed model checking framework, PAT out-performed most of the tools in previous section on the proposed problem domain. Using PAT as a planning service has several advantages over other alternatives.

- The searching algorithms of PAT is highly efficient and ready to be used, as is proved in the comparisons with other tools. Therefore, the performance of planning is ensured with no extra effort. It also saves the time of implementing a different planning algorithm for every new problem.
- CSP# is a highly expressive language for modelling various kind of systems. The tools we experimented on, including SatPlan and Metric-FF, are all restricted to a certain area of problems. For instance, SatPlan is not able to solve planning problems with numerical plan metrics and Metric-FF lacks support for plan optimization problems. With a number of sophisticated model checking options, such as "reachability-with" and "BFS/DFS", PAT is ready to solve all kinds of planning problems.
- PAT is constructed in a modularized fashion. Modules for specific purposes can be built to give better support for the domains that are considered. For example, using "Probability CSP Module", it is even possible to solve nondeterministic planning problems with PAT. Of course, we can also build our own planning modules with customized searching algorithms. We shall further discuss this in section 4.

To make the PAT model checker effectively working for general planning problems, it is essential to establish translation mechanism between a commonly used planning description language into the one that PAT understands. In the following section, we describe a source-to-source translation from PDDL to CSP#. Our goal is to formulate mapping rules that can act as a guide when performing the translation. The translation is based on two basic assumptions:

- The PDDL domain descriptions are written in the subset of PDDL 2.1 that includes STRIPS-like operators with literals having typed arguments and numerical plan metrics. The typing can be easily done by hand or a tool such as TIM [21] when the original model is written without typed arguments.
- The translation should keep, as far as possible, the naming as well as the structures of the original PDDL domain descriptions.

In the following subsections, we shall explain the translation process using a classic planning problem as the running example. The *bridge crossing* problem is described as follows. Four wounded soldiers find themselves behind enemy lines and try to flee to their home land. The enemy is chasing them and in the middle of the night, they arrive at a bridge that spans a river which is the border between the two countries at war. The bridge has been damaged and can only carry two soldiers at a time. Furthermore, several landmines have been placed on the bridge and a torch is needed to sidestep all the mines. The enemy is on their tail, so the soldiers know that they have only 60 minutes to cross the bridge. The soldiers only have a single torch and they are not equally injured. The extent of their wounds have an effect on the time it takes to get across. So the time needed for each soldier are 5, 10, 20, 25 minutes respectively. The goal is to find a solution to get all the soldiers to cross the bridge to safety in 60 minutes or less.

A. Types and Objects

A PDDL model consists of two types of files for a problem description, i.e., the domain file and the problem file. The former gives the domain description of the problem such as the types, predicates and behavioral actions involved in the model; the latter simply defines the problem to be solved in terms of the objects used, initial states and the final goal states.

PDDL domain description has a special syntax for declaring parameter types. If types are to be used in a domain, the domain file should include a declaration: `(: types NAME1 ... NAME_N)`. For example, the following shows the typing declaration for the *bridge crossing* problem.

```
(:types place locatable - object
        soldier torch - locatable)
```

Note that it uses a hierarchical typing system, where *place* and *locatable* are of primitive type *object*, while *soldier* and *torch* belong to *locatable*.

After the declaration of types in the domain definition file, objects can be defined with types in the problem description

file. The following shows the object definitions for *the bridge crossing problem*.

```
(:objects
  soldier0 soldier1 soldier2
    soldier3 - soldier
  torch - locatable
  north south - place)
```

Object *torch* and *soldier0...3* are of *locatable* type, while *north* and *south* are of *place* type. To translate PDDL type and object declarations into CSP#, we declare a constant enumeration in CSP# for every group of objects of the same type. For example, the above type and object declaration in PDDL can be mapped to CSP# as follows.

$$enum\ \{north, south\};$$
$$enum\ \{soldier0, soldier1, soldier2, soldier3, torch\};$$

B. Predicates

In PDDL, preconditions and effects are expressed as logic expressions of predicates. To represent predicates in CSP#, we construct a self-defined data-type ⟨*Predicate*⟩ in PAT. ⟨*Predicate*⟩ has three methods that can be directly called from CSP# models, including:

1) *void setPredicate(predicateName, x, y, value);*
2) *bool tryPredicate(predicateName, x, y);*
3) *int snapShot();*

The example above only shows the methods for predicates with an arity of two. *setPredicate* is used to set the value of a predicate with its name specified as the first parameter. *tryPredicate* returns the value of a predicate and *snapShot* returns an integer that represents the current snapshot of the predicate database. To use the self-defined data-type, the C# library has to be imported and instantiated first. For each predicate declared in the domain definition, we also need a corresponding enumeration type in CSP#. For example, the following PDDL predicate can be mapped to CSP# as shown below.

```
(:predicates
  (at ?x - locatable ?y - place))
```

$$\#import\ \text{“}Predicate\text{”};$$
$$var < Predicate >\ pre\ =\ new\ Predicate();$$

$$enum\ \{At\};$$
$$\ldots$$
$$pre.setPredicate(At, x, y, true);$$

C. Actions

In a PDDL domain description, actions are defined to specify the behavioral aspects of the model in terms of preconditions and effects. With object types and predicates ready, the translation of actions into CSP# is straightforward. The preconditions are translated as guard conditions of processes. The effects are translated as statement blocks after event names. Conditional effects can also be easily converted into conditional branches that are well supported in CSP#. The updates for plan metrics can be mapped into simple data operations. The followings show the semantically equivalent action definitions for “south to north” in PDDL and its translation in CSP# respectively.

```
(:action StoN
  :parameters (?x - soldier
    ?y - soldier)
  :precondition (and (at ?x south)
    (at ?y south) (at torch south))
  :effect (and (not (at ?x south))
    (not (at torch south))
    (not (at ?y south)) (at ?x north)
    (at ?y north) (at torch north)
    (when (>= (time ?x) (time ?y))
      (increase (time-cost)(time ?x)))
    (when (< (time ?x) (time ?y))
      (increase (time-cost)(time ?y)))))
```

$$
\begin{aligned}
StoN(x, y)\ =\ [\ & x! = y \\
&\&\&\ pre.tryPredicate(At, x, south) \\
&\&\&\ pre.tryPredicate(At, y, south) \\
&\&\&\ pre.tryPredicate(At, torch, south)\ ] \\
&s.x.y\{pre.setPredicate(At, x, north, true); \\
&\quad pre.setPredicate(At, x, south, false); \\
&\quad pre.setPredicate(At, y, north, true); \\
&\quad pre.setPredicate(At, y, south, false); \\
&\quad pre.setPredicate(At, torch, north, true); \\
&\quad pre.setPredicate(At, torch, south, false); \\
&\quad if(time[x] > time[y]) \\
&\quad\ \ \{time_cost = time_cost + time[x];\ \} \\
&\quad else\{time_cost = time_cost + time[y];\ \} \\
&\ \}\ \rightarrow\ Trans();
\end{aligned}
$$

In the above example, the predicates inside the square brackets represents a guard condition for the process, which are translated from the precondition in the PDDL action definition. The *x!=y* in the guard condition is to ensure the two parameters *x* and *y* are distinct, which is implicitly enforced in PDDL. Furthermore, the predicates in the effect part of a PDDL action definition can be easily mapped to their corresponding predicate definitions in a CSP# process definition with time constraints as shown above.

More importantly, to establish a state transition system on the model, we also need another process to choose among different actions. As shown in the following, the process *Trans()* first makes a snapshot of the current predicate database, then nondeterministically chooses one action, i.e., either *StoN* or *NtoS*, and proceeds. The parameters for the actions are also nondeterministically chosen among the available objects that are of the suitable types. This is done

by using the syntax sugar "indexed event list" that takes in parameters within the corresponding enumeration range.

$$\begin{aligned} \mathit{Trans}() \;=\; & \tau\{snap \;=\; pre.snapshot()\} \rightarrow \\ & (\Box \;\; z : \{0..3\}@(\Box \;\; y : \{0..3\}@StoN(z, y))) \\ & \Box \; (\Box \;\; x : \{0..3\}@NtoS(x)); \end{aligned}$$

D. Initial State

In PDDL, the initial state of a model description is captured by the problem file, where the objects and goal states are also defined. To translate the initial state definition in PDDL into CSP#, a corresponding initial process can be constructed. The following shows the translation of initial state specifications from PDDL to CSP# of the *bridge crossing* problem.

```
(:init
  (at soldier0 south)
  (at soldier1 south)
  (at soldier2 south)
  (at soldier3 south)
  (at torch south)
  (= (time soldier0) 5)
  (= (time soldier1) 10)
  (= (time soldier2) 20)
  (= (time soldier3) 25)
  (= (time-cost) 0))
```

$$\begin{aligned} & \mathit{var}\ \mathit{time}[4] = [5, 10, 20, 25]; \\ & \mathit{var}\ \mathit{time_cost} \;=\; 0; \end{aligned}$$

$$\begin{aligned} \mathit{ini}() \;=\; & \mathit{initial}\{ \\ & \quad pre.setPredicate(At, soldier0, south, true); \\ & \quad pre.setPredicate(At, soldier1, south, true); \\ & \quad pre.setPredicate(At, soldier2, south, true); \\ & \quad pre.setPredicate(At, soldier3, south, true); \\ & \quad pre.setPredicate(At, torch, south, true)\} \\ & \rightarrow \; Skip; \end{aligned}$$

Note that for the functions `time` and `time-cost` in PDDL, we simply use an integer array and an integer variable in CSP# to represent them respectively. The initialization of predicates are done within the process *ini()*.

$$Plan = ini(); \; Trans();$$

With the initial state and actions translated into CSP#, the complete state transitions (behaviors) of the *bridge crossing* model can be defined as the initial state followed by the *Trans* process as defined in the above *Plan* process.

E. Goal

The goal of a PDDL problem description contains logic formulas of predicates and possibly also specifies the plan optimization criteria. The translation of optimization criteria can be achieved by using the keyword "*reaches ... with ...*". The following shows the corresponding goal states definitions of PDDL and CSP# in the *bridge crossing* problem.

```
(:goal (and
        (at soldier0 north)
        (at soldier1 north)
        (at soldier2 north)
        (at soldier3 north)))
(:metric minimize (time-cost)))
```

$$\begin{aligned} \#\mathit{define\ goal}\ \; & (pre.tryPredicate(At, soldier0, north) \\ & \&\&\ pre.tryPredicate(At, soldier1, north) \\ & \&\&\ pre.tryPredicate(At, soldier2, north) \\ & \&\&\ pre.tryPredicate(At, soldier3, north)); \end{aligned}$$

$$\#\mathit{assert\ Plan\ reaches\ goal\ with}\ min(time_\ cost);$$

Note that the above assertion defines process *Plan* can reach the goal within a certain time restriction. The PAT model checker will search its state transition paths to explore a possible trace of the model that satisfies this condition, which is essentially a solution to the planning problem. Clearly, using the newly defined $\langle Predicate \rangle$ type, the translation from PDDL to CSP# is a straightforward mapping as it can be shown in the above example. Translation tools can even be developed to automate the entire process.

IV. Performance Evaluation

In this section, we exam the performance side of the translation. We conducted similar experiments as in section 2, which executes the translated *bridge crossing* model in PAT against those of other planning tools and compared their performance. The *bridge crossing* problem is a plan existence problem with a constraint on the total time. A workable plan that can be finished within 60 minutes is already good enough. There is no need to literally "calculate" an optimal solution. PAT can find the "Shortest Witness Trace" by using the breadth-first search in the state space, i.e., the returned counterexample trace is guaranteed to be the shortest one. Otherwise, a depth-first search is performed and the first counterexample trace encountered is displayed. Therefore, for *the bridge crossing problem* where shortest witness trace is not needed, we used the depth-first search mode; for *the sliding game problem*, for which an optimal solution is expected, we enabled the "Shortest Witness Trace" option instead. The counterexample provided by NuSMV is always shortest, so it can also be used to generate optimal solutions for *the sliding game problem*. Unfortunately, as mentioned, the counterexample produced by Spin is not always shortest. However, we still collected the performance data for reference.

To generalize the problem and get the experimental results in a broader range, we expanded the original *bridge crossing problem* to versions with up to 9 soldiers. Except the breadth-first and depth-first search, PAT also supports

Table II: Experimental results for *the bridge crossing problem*

Soldiers	Time	Metric-FF	PAT		NuSMV			Spin
			WITH	DFS	INVAR	CTL	LTL	
4	60	0.00	0.05	0.04	0.0	0.1	0.1	0.02
5	90	0.00	0.19	0.04	0.1	0.9	0.4	0.02
6	130	0.03	1.12	0.22	0.2	14.4	2.5	0.06
7	175	0.16	6.18	0.25	0.5	330.8	71.3	0.11
8	235	0.94	33.19	10.26	m	m	m	10.50
9	300	5.30	145.51	16.40	m	m	m	19.50

"reachability-with" checking which is a reachability test with some state variables reaching their maximum/minimum values. Hence PAT can be used to find the minimum amount of time needed to finish the bridge crossing. The time limits were first calculated by PAT using the "reachability-with" mode. Other model checkers were then tested taken the time limits as given. Of course, to be fair, PAT was also run one more time using the depth-first search mode. We also ran Metric-FF on *the bridge crossing problem* with parameters $g = 100$ and $h = 1$, which emphasises the plan quality over the performance to increase the possibility of getting an solution within the time limit.

Table III: Time cost of each soldier

Soldier	1	2	3	4	5	6	7	8	9
Time Cost	5	10	20	25	30	45	60	80	100

This set of experiments are tailored to show how the model checkers compete on plan existence problems that deal with time constraints. The results are summarized in Table II. Inside the table, the column "Soldiers" indicates the number of soldiers in the problem instance and the column "Time" indicates the time limit used in that test. A symbol *m* is there to show that the system ran out of memory and did not get a solution. Although the configurations for Metric-FF ($g = 100$ and $h = 1$) have put a much higher weight on plan quality, the optimality of the results got form Metric-FF is still not guaranteed. So the data is only used as a benchmark for comparisons. The time cost of each soldier is listed in Table III above.

When the number of soldiers reaches 8, NuSMV is not able to build a model according to the model descriptions due to memory shortage. The invariant checking mode performs generally better than CTL and LTL checking mode because CTL and LTL model checking algorithms' searching space involves both the model and the property, but reachability checking only explore the model's space[1]. With regard to Temporal model checking in NuSMV, the performance is better using LTL than CTL.

[1]PAT will automatically detect the safety LTL properties and convert them into reachability problems. Hence, we do not include the LTL checking model for PAT in this experiment.

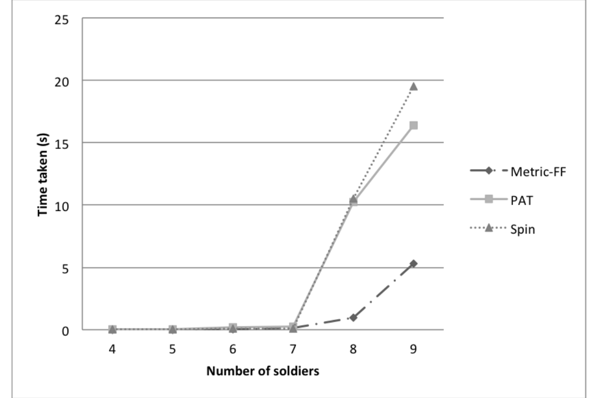

Figure 4: Execution time comparison of PAT, Spin and Metric-FF on *the bridge crossing problem*

Figure 4 shows that the time needed for *the bridge crossing problem* increases rapidly when the number of soldiers increases. For example, the execution time for Spin increases by nearly 100 times when the number of soldiers increases from 7 to 8. It is clear that the state space expands in a very fast speed. Planners such as Metric-FF handle this kind of problem in a very different way from model checkers. Metric-FF performs a standard weighted A* search which exploits the power of heuristics and sacrifices the optimality to speed up the searching. That is the reason why Metric-FF performs much better than the other two. The performance of PAT and Spin is similar on this problem domain. For smaller instances, for example, when the number of soldiers ranges from 4 to 7, Spin performs better than PAT, although the difference is relatively small. For larger instances like the problem with 8 or 9 soldiers, PAT starts to perform better that Spin.

V. Conclusion

In this paper, we focused on exploring the use of model checking techniques on the AI planning domain. We believe our work established a good start point in this direction towards more practical applications. We first examined the feasibility of using different model checkers on solving classic planning problems. In our experiments, we compared the performance and capabilities of different tools including PAT, NuSMV and Spin. PAT is proved to be the most

suitable one for solving various kind of planning problems. The experimental results also indicate that some model checkers, e.g. PAT, can even compete with sophisticated planners in certain domains.

Based on the performance evaluation, we further suggested the approach of developing PAT as a planning service. We presented translation guidelines from PDDL to CSP#, which could serve as a basis for facilitating the verification. We defined mapping rules on different constructs of a PDDL model and presented them through a running example of the *bridge crossing* problem. Finally, we compared the performance of the translated model with that of other planning tools, which are attempted based on their execution time and memory efficiency as well as their planning quality. The results showed that our approach provides a good solution towards the problem.

Although experiments have been carried out on three model checkers and two planners so far, we would like to extend the comparisons to a larger range of model checking as well as planning tools to get a more general view of the subject. We also observe that, in some of the models, there is a lot of room for improvement. By either fine tuning the way of modelling or exploiting domain specific knowledge, we could further optimize the models. In addition, we are interested in implementing an automated translator for the translation from PDDL to CSP#. Large amount of work has to be done to ensure the correctness and efficiency of the translation. Last but not least, we recommend that more research should be done on applying PAT as a planning service. The applications of this technique should be extended to a larger range on real problems in various fields.

REFERENCES

[1] D. Peled, P. Pelliccione, and P. Spoletini, *Wiley Encyclopedia of Computer Science and Engineering*. John Wiley & Sons, 2009, ch. Model Checking.

[2] D. Berardi and G. D. Giacomo, "Planning via model checking: Some experimental results," 2000, unpublished. [Online]. Available: http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.144.1377

[3] G. J. Holzmann, *The SPIN Model Checker: Primer and Reference Manual*. Addison-Wesley Professional, Sep. 2003.

[4] K. L. McMillan, "Symbolic model checking: an approach to the state explosion problem," Ph.D. dissertation, Carnegie Mellon University, 1992.

[5] J. Koehler, B. Nebel, J. Hoffmann, and Y. Dimopoulos, "Extending planning graphs to an ADL subset." Springer-Verlag, 1997, pp. 273–285.

[6] J. Hoffmann and B. Nebel, "The FF planning system: Fast plan generation through heuristic search," *Journal of Artificial Intelligence Research*, vol. 14, pp. 253–302, 2001.

[7] F. Bacchus, F. Kabanza, and U. D. Sherbrooke, "Using temporal logics to express search control knowledge for planning," *Artificial Intelligence*, vol. 16, pp. 123–191, 2000.

[8] T. Hörne and J. A. van der Poll, "Planning as model checking: the performance of ProB vs NuSMV," in *Proceedings of the 2008 annual research conference of the South African Institute of Computer Scientists and Information Technologists on IT research in developing countries: riding the wave of technology*, ser. NY, USA: ACM, 2008, pp. 114–123.

[9] R. Cavada, A. Cimatti, C. A. Jochim, G. Keighren, E. Olivetti, M. Pistore, M. Roveri, and A. Tchaltsev, *NuSMV 2.5 User Manual*, CMU and ITC-irst, 2005.

[10] R. E. Bryant, "Symbolic boolean manipulation with ordered binary-decision diagrams," *ACM Computing Surveys*, vol. 24, pp. 293–318, 1992.

[11] S. Russell and P. Norvig, *Artificial Intelligence*. Pearson, 2010, ch. Classical Planning.

[12] D. V. McDermott, *PDDL - The Planning Domain Definition Language*, Yale Center for Computational Vision and Control, 1998.

[13] F. Giunchiglia and P. Traverso, "Planning as model checking," in *Recent Advances in AI Planning*, ser. Lecture Notes in Computer Science, S. Biundo and M. Fox, Eds. Springer Berlin / Heidelberg, 2000, vol. 1809, pp. 1–20.

[14] J. Sun, Y. Liu, J. S. Dong, and J. Pang, "PAT: Towards flexible verification under fairness," in *The 21th International Conference on Computer Aided Verification (CAV 2009)*. Grenoble: Springer, 2009, pp. 709–714.

[15] J. Sun, Y. Liu, J. S. Dong, and C. Chen, "Integrating specification and programs for system modeling and verification," in *Proceedings of the third IEEE International Symposium on Theoretical Aspects of Software Engineering (TASE'09)*, W.-N. Chin and S. Qin, Eds. IEEE Press, 2009, pp. 127–135.

[16] C. A. R. Hoare, "Communicating Sequential Processes," *Communications of ACM*, vol. 21(8), pp. 666–677, Aug. 1978.

[17] J. Hoffmann, "Extending FF to numerical state variables," in *Proceedings of the 15th European Conference on Artificial Intelligence (ECAI-02)*, Lyon, France, Jul. 2002, pp. 571–575.

[18] H. A. Kautz, B. Selman, and J. Hoffmann, "SatPlan: Planning as satisfiability," in *Abstracts of the 5th International Planning Competition*, 2006.

[19] P. Gregory, D. Long, and M. Fox, "A meta-CSP model for optimal planning," in *Proceedings of the 7th International conference on Abstraction, reformulation, and approximation*, ser. Berlin, Heidelberg: Springer, 2007, pp. 200–214.

[20] A. Reinefeld, "Complete solution of the eight-puzzle and the benefit of node ordering in IDA*," in *Proceedings of the 13th international joint conference on Artificial intelligence - Volume 1*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1993, pp. 248–253.

[21] M. Fox and D. Long, "The automatic inference of state invariants in TIM," *Journal of Artificial Intelligence Research*, vol. 9, pp. 367–421, 1998.