# Planning as Model Checking Tasks

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 states. 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 different model checking tools and techniques for solving classic planning problems. To achieve this, we carried out 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. In particular, a new planning module with specifically designed searching algorithm is implemented on top of the established model checking framework, Process Analysis Toolkit (PAT), to serve as a planning solution provider for upper layer applications. A case study on a public transportation management system has been developed to demonstrate the idea of using the PAT model checker as a planning service.**

## 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 $\varphi$ 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, several research indicates that model checking can also be applied to AI planning domain. Berardi and Giuseppe [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. And 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). 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 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 [10], 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 $s \in G$.

Clearly, a classical planning problem can be easily converted into a model checking problem. The fact that this approach is feasible was supported by [11], which states that planning should be done by semantically checking the truth of a formula and 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 $\varphi$, we can construct a safety property $\mathbf{G}\neg\varphi$ that requires the formula $\varphi$ never to hold, such that the model checker is able to search for a counterexample path that leads to a state where $\varphi$ holds. The resulting plan is optimal in terms of make-span when the counterexample path is the shortest.

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 checkers can even be further improved by exploiting domain-specific knowledge. In particular, a newly developed model checking framework - Process Analysis Toolkit (PAT) [12] 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 demonstrate our approach through a case study on a public transportation management system that uses the PAT model checker as a planning service.

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 as a planning service by developing a route planning module for the Transport4You system. It realizes the practical application of the approach with concrete evaluation results, where different model checking algorithms designed for the module are compared and analyzed. Section 4 concludes the paper and outlooks the future directions.

## II. Review on 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.

SatPlan [13] 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 $14^{th}$ 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.

Metric-FF [14] 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 $3^{rd}$ 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, which allows numerical plan metrics.

NuSMV [9] 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.

Spin [3] 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. Promela loosely follows CSP and its 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.

Process Analysis Toolkit (PAT) [12] 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.

### A. 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 (version 3.3.0) on solving two classical planning problems: *the bridge crossing problem* and *the sliding game problem*. SatPlan2006 and Metric-FF are also used as benchmarks in the experiments. The two problems selected can be regarded as puzzle solving problems and the optimal solutions are not trivial. The descriptions of the problems are as follows.

- *The bridge crossing problem*: Four wounded soldiers find themselves behind enemy lines and try to flee to their home land. 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 land mines have been placed on the bridge where a torch is needed to sidestep all the mines. The soldiers only have a single torch and 60 minutes to cross the bridge, 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.
- *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 $3 \times 3$ 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 increasing setting.

Note that *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, the counterexample produced by Spin is not always shortest. However, we still collected the performance data for reference.

To collect the execution time data more accurately, we performed each experiment three times and calculated the average to avoid possible fluctuations caused by the overhead imposed by operating systems. All the experimental results were collected on an Dell desktop with an Intel Core 2 Duo E6550 2.33GHz processor and 3.25GB RAM. Spin, 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.

The experimental results are presented in the following subsections, 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 or breadth-first search respectively. Time is in seconds unless otherwise indicated.

*1) The Bridge Crossing Problem:* To generalize the problem and obtain experimental results in a broader range, we expanded the original *bridge crossing problem* to versions with up to 9 soldiers. Apart from the breadth-first and depth-first searches, PAT also supports a "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 emphasizes the plan quality over the performance to increase the possibility of getting an solution within the time limit.

TABLE I: 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 |

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 |

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. The time cost of each soldier is listed in Table I above. 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.

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. Figure 1 shows that the time

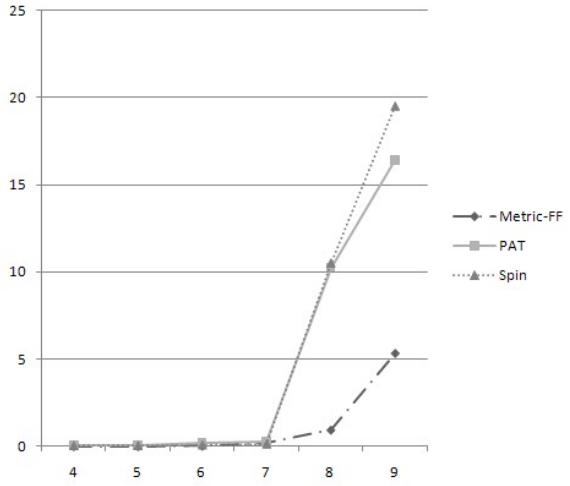



Fig. 1: Execution time comparison of PAT, Spin and Metric-FF on *the bridge crossing problem*

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.

*2) 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 [17]. 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 [18]. 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.

This set of experiments are designed to show how different model checkers perform on optimal deterministic planning problems. The results got from SatPlan are used for reference. The results are summarized in Table III. 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.

[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.

TABLE III: 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 |

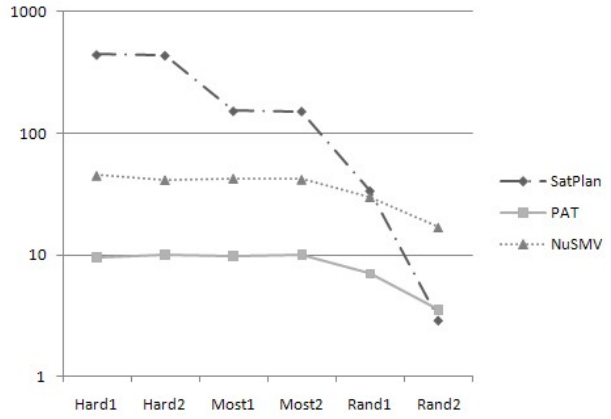



Fig. 2: Execution time comparison of PAT, NuSMV and SatPlan on *the sliding game problem*, shown on a logarithm scale

From Figure 2 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.

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

## III. PAT as a Planning Service

When performing the experiments in Section 2, it is shown 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 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. Additionally, we can also build our own planning modules with customized searching algorithms. We shall further discuss the later section.

In this section, we present a case study on "Transport4You" which is a project submission by our research group to the $33^{rd}$ International Conference on Software Engineering (ICSE) - Student Contest on Software Engineering (SCORE). The project won the "Formal Methods Award" [2] out of 56 submissions, which was presented for the final round of the

[2]The awards page of the $33^{rd}$ International Conference on Software Engineering (ICSE 2011) in Hawaii, USA – http://2011.icse-conferences.org/content/awards.

competition at ICSE 2011 in Hawaii. The "Transport4You" Intelligent Public Transportation Manager (IPTM) is a specifically designed municipal transportation management solution which is able to simplify the fare collection process and provide customized services to each subscriber. To be specific, a system that is able to provide customized trip information and timely responses to each subscriber is to be built to satisfy the increasing needs. In other words, the new system should not only play the role of a bus conductor, but also be a trip advisor who informs the users of changes in the lines and possibly suggests optimized routes for them.

The "Transport4You" IPTM system consists of two sub systems, namely the bus embedded system (BES) and the central mainframe (CM). The bus system is responsible for passenger detection, part of the fault correction and detection results report to the central server. In contrast, the server system deals with all kinds of service requests from users and administrators, information management, as well as user notification. The two sub systems communicate via TCP connections and at the same time interact with users and administrators. A significant component of the "Transport4You" IPTM system is the *Route Plannig* module which makes use of the model checking capability of PAT as a planning service. This function provides a guide for users who are not familiar with the bus routes and need suggestions for choosing bus lines. This can also be applied to suggest alternative optimal routes to subscribers, based on the behavioural data analysed in the *User Behavior Analysis* module. To further illustrate the idea of using PAT as planning service, we have built a simulator for the IPTM system.

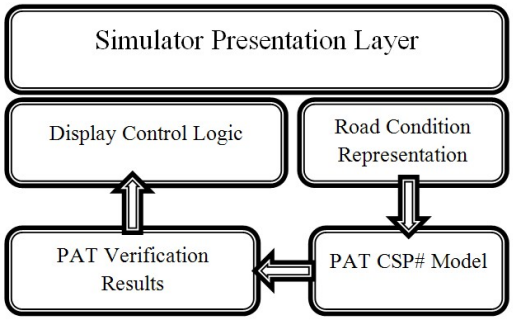



Fig. 3: Simulator architecture diagram

As is shown in Figure 3, the simulator generates a CSP# model during execution according to the current road conditions and bus line configurations, whenever a subscriber is querying on which route to choose. Users can choose their starting point as well as destination on the simulator interface. After clicking on the "Plan" button, the underlying support modules generate a CSP# model according to what have been chosen and pass it to PAT. After interpreting the returned results from PAT, the system is able to display the planned route and detailed instructions to users as shown in Figure 4. The *route planning* module can work correctly even when there are real time changes on road conditions. When the interrupted road or bus service is detected, the administrators will update the road condition database immediately. All subsequent queries will be processed according to the newly updated road conditions. The planning results are, therefore, guaranteed to be accurate based on the most updated data.

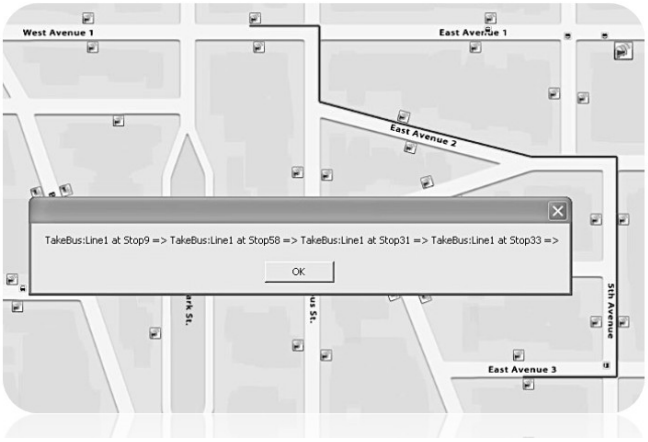



Fig. 4: Simulator screen shot of route planning results

### A. Route Planning Model Design

In this subsection, we discuss the design of the route planning CSP# model. We will look at two different approaches for improving the solution quality and compare the performance of them. To construct a CSP# model for route planning, we have to first formally define the problem. There are 14 bus lines travelling among 61 bus stops on our simulated city map. In addition, each bus line has a sequence of bus stops that it must reach one by one.

*Definition 1:* A Route Planning **task** is defined by a 5-tuple (S,B,t,c,L) with the following components:

- S is a finite, non-empty set of **bus stops**. Terminal stops include start terminals $s_{start} \subseteq S$, and end terminals $s_{end} \subseteq S$, where $s_{start} \cap s_{end} = \varnothing$.
- B is a finite set of **bus lines**, and for every bus line $b_i \in B$, $b_i : S \to S$ is a partial function. $b_i(s)$ is the next stop taking bus i from stop s. $\forall s \in s_{start} \forall b \in B, s \in dom(b) \longrightarrow b^{-1}(s) = \alpha$. $\forall s \in s_{end} \forall b \in B, s \in dom(b) \longrightarrow b(s) = \beta$. $\forall b \in B, b^{-1}(\alpha) = \alpha \wedge b(\beta) = \beta$.
- $t : S \to B_S$ is a function where $B_S \subseteq B$. $t(s)$ is the set of available bus lines at stop s, i.e., $B_S = \{b_i \in B \mid s \in dom(b_i)\}$.
- $c : S \to S$ is a partial function. $c(s)$ is the stop one can get to by crossing the road at stop s.
- L is a unary predicate on S. L(s) is true when the current location of user is stop s.

The definition should be intuitive enough and require little additional explanation. The tuple can be constructed from the evaluation of the bus line and road configurations that are stored in the ITPM central mainframe. Now we can define the *Route Planning domain*.

*Definition 2:* Given initial location $s_0$ and destination $s_g$, a Route Planning **domain** maps a Route Planning task to a classical planning problem with close-world assumption as follows:

States: Each state is represented as a literal $s \in S$, where L(s) holds.
Initial State: $s_0$
Goal States: $s_g$
Actions: 1) ($TakeBus(b_i, s)$, PRECOND: $b_i \in t(s)$, EFFECT: $\neg L(s) \wedge L(b_i(s))$)
2) ($Cross(s)$, PRECOND: $s \in dom(c)$, EFFECT: $\neg L(s) \wedge L(c(s))$)

After defining the problem, we shall look at a basic CSP# model that solves the route planning problem. According to the problem definitions, the model includes four parts, namely the environment variables (bus stops and bus lines), the initial state, the state transition functions (actions) and the goal states. The design of each part will be discussed as follows.

*1) Environment Variables:* In the description of the environment variables, we first declare an enumeration that lists all the bus stops for later use:

$$enum\{TerminalA,\ Stop5,\ Stop7,\ Stop9\ ...\ Stop26,\ Stop11,\ Stop35,\ Stop34\};$$

Then we use a self-defined data type $\langle BusLine \rangle$ to keep track of the bus line configurations and provide useful helper methods.

$$var\ sLine1 = [TerminalA, Stop5, Stop7, Stop9, Stop58, Stop31, Stop33, Stop53, Stop57, TerminalC];$$
$$var\langle BusLine\rangle Line1 = new\ BusLine(sLine1, 1);$$
$$var\ sLine2 = [TerminalC, Stop56, Stop52, Stop32, Stop30, Stop59, Stop10, Stop8, Stop6, TerminalA];$$
$$var\langle BusLine\rangle Line2 = new\ BusLine(sLine2, 2);$$
$$...$$
$$var\ sLine14 = [TerminalC, Stop34, Stop32, Stop30, Stop16, TerminalB];$$
$$var\langle BusLine\rangle Line14 = new\ BusLine(sLine14, 14);$$

In the above code, the instantiation of $\langle BusLine \rangle$ takes in two parameters, including an integer array that contains a sequence of bus stops as well as an integer that is the line number. After declaration, we are able to use the bus line variable to look up useful information of a particular bus line including the previous stop and the next stop with respect to the current stop.

*2) Initial State:* In the description of the initial states, we declare two variables, *currentStop* and *currentBus*. The variable *currentStop* corresponds to the state variable *s* mentioned before, while *currentBus* is only for record in the current model.

$$var\ currentStop = Stop5;$$
$$var\ B0 = [-2];$$
$$var\langle BusLine\rangle currentBus = new\ BusLine(B0, -1);$$

The initial value of *currentStop* is set to be *Stop5* in this example. The *currentBus* is also a variable of type $\langle BusLine \rangle$ and its initial value is set to some negative integer to avoid confusion.

*3) State Transition Functions:* Now we are coming to the most critical part of the model. We need to translate the action schema mentioned before to a state transition function that can be further converted to CSP# processes with the help of the "*case*" statement. The description of transition functions can be further divided into two parts. In the first part, a process named *takeBus()* is defined to capture the state transitions caused by taking bus. The second part deals with a process *crossRoad()* which is defined to capture the state transitions caused by walking to the opposite side of the road.

$$takeBus() = case\{$$
$$currentStop == TerminalA : BusLine1[]$$
$$BusLine3[]BusLine5[]BusLine7$$
$$currentStop == Stop5 : BusLine1[]BusLine5$$
$$currentStop == Stop7 : BusLine1[]BusLine5$$
$$currentStop == Stop9 : BusLine1$$
$$...$$
$$currentStop == Stop11 : BusLine12$$
$$currentStop == Stop35 : BusLine13$$
$$currentStop == Stop34 : BusLine14$$
$$\};$$

This process *takeBus()* simply hands over the control to another process according to which bus lines are available in the current bus stop. For example, at *Stop5*, there are two bus lines available, namely *BusLine1* and *BusLine5*. Then we still need to define processes *BusLine1* to *BusLine14* which have very similar events.

$$BusLine1 = TakeBus.1\{$$
$$currentStop = Line1.NextStop(currentStop);$$
$$currentBus = Line1;\ \} \rightarrow takeBus();$$
$$...$$
$$BusLine14 = TakeBus.14\{$$
$$currentStop = Line14.NextStop(currentStop);$$
$$currentBus = Line14;\ \} \rightarrow takeBus();$$

This is where the actual state transitions happen. Each bus line process invokes *TakeBus.n* event, and at the same time, updates the value of *currentStop* and *currentBus*. Finally, the bus line process returns the control to the process *takeBus()*. Notice that there is another version of this process that also allows road crossing at any bus stop. We shall look at it later after the discussion of the *crossRoad()* process.

$$crossRoad() = case\{$$
$$currentStop == Stop5 : cross\{currentStop = Stop6\} \rightarrow takeBus()$$
$$currentStop == Stop7 : cross\{currentStop = Stop8\} \rightarrow takeBus()$$
$$...$$
$$currentStop == Stop35 : cross\{currentStop = Stop34\} \rightarrow takeBus()$$
$$currentStop == Stop34 : cross\{currentStop = Stop35\} \rightarrow takeBus()$$
$$\};$$

The process *crossRoad()* also makes use of the key word "*case*". Depending on the value of *currentStop*, a common event *cross* will be evoked and the hidden effect is the update of *currentStop* to the stop opposite to it. For instance, when the user is at *Stop5*, event *cross* can happen and the user's location is changed to *Stop6*. After the state transition, the process also hands over its control to *takeBus()* and searches for further transitions. Combining two processes by an external choice operator gives us the final transition function:

$$plan = takeBus() [] crossRoad();$$

As mentioned before, to enable road crossing, we have to modify the bus line process. Instead of returning the control to *takeBus()*, we have to return to *plan* which may also invoke the process *crossRoad()*. This could increase the search space of the model, however the increase of verification time is not significant.

*4) Goal States:* The goal states of the model are fairly easy to define. Very similar to the initial state description, we only need to specify the goal to be that the value of *currentStop* equals to the destination stop chosen by user that is *Stop53* in our example.

$$\#define\ goal\ currentStop == Stop53;$$

### B. The Cost Function Approach

The basic model we discussed before is able to solve the Route Planning problem. It even provides optimal plans in terms of the make-span if the "BFS" mode is used. However, the quality of the plan is not always guaranteed. The plan quality depends on several factors, including the length of the suggested route, the total walking distance, the number of buses changed, etc. To measure the plan quality, we introduce cost function into the model. It is fairly intuitive to assign a non-negative integer value to each action. For instance, we assign a cost of 10 for $TakeBus(b_i, s)$ and a cost of 2 for $Cross(s)$. In addition, we also assign a cost of 5 for two consecutive *TakeBus* actions with different $b_i$, which implies there is a bus change occurring. The plans produced by the basic model are sometimes suboptimal in terms of the total cost. There are two causes for the inefficiency:

- The basic model treats action *Cross* and *TakeBus* as the same. However, in real life, different subscribers may have their own preferences on the minimization of the number of bus stops or the walking distance.
- The basic model does not have penalties on bus changes when producing the route plan. The number of bus changes is considered a critical factor when judging the quality of the plan.

To ensure high plan quality in our new model, we use a cost function as gauge. The implementation of the cost function in our established basic model can be done with very little effort. For *takeBus()* and *crossRoad()* process, we can add a hidden event: $tau\{cost = cost + x\}$, where $x$ is 10 or 2. For bus changes, we can add another hidden event with a conditional branch:

$$tau\{if(!currentBus.isEqual(LineX))\{cost = cost + 5\}\}$$

where *LineX* is the bus line to be taken next.

**Algorithm 1** newBFSVerification()

```
initialize queue: working;
current ← InitialStep;
τ ← ∞;
repeat
  value ← EvaluateCost(current);
  if current.SatisfyGoal() then
    if value < τ then
      τ ← value;
    end if
  end if
  if value > τ then
    continue;
  end if
  for all step ∈ current.MakeOneMove() do
    working.Enque(step);
  end for
until working.Count() ⩽ 0
```

However, the introduction of cost function also increases the complexity of the problem. The original optimal planning problem can be solved by a simple breadth-first search. As the size of optimal solutions in this context is usually small, the execution time is also relatively short. Unfortunately, the default "reachability-with" checking algorithm in PAT searches the whole state space for a maximum/minimum value of a given variable. The execution time is considerably long for this kind of searching according to our experiments. To resolve the problem of long execution time, we design a new searching algorithm with the assumption that all cost values are non-negative integers. Once a solution is found in the searching, we update the threshold $\tau$ with its cost value. In the following search, if the cost of the current partial plan exceeds $\tau$, we consider it a dead-end since no further transitions could make the cost lower. This pruning of the search space largely reduces the execution time and memory usage to a satisfactory level and still preserves the optimality of the solutions. The new algorithm *newBFSVerification()* is given in Algorithm 1. In this algorithm, *working* is the task queue used in the BFS; $\tau$ is the temporary variable to stored the current best value met so far; *value* stores the cost valuation of the *current* state; *current.SatisfyGoal* returns true if the goal is satisfied for the current state; *current.MakeOneMove* returns the all possible outgoing transitions from the current state. The state pruning happens if *value* is greater than $\tau$.

### C. Search Space Pruning

As mentioned in the previous subsection, one of the reasons for producing suboptimal solutions is that the number of bus changes is uncontrolled. Taking an example shown in Figure

5, bus line $b_1$ and $b_2$ both travel along the path $\langle s_1, s_2, s_3 \rangle$. The route of $b_1$ is shown in solid lines while the route of $b_2$ is shown in dashed lines. We refer to a particular edge between two stops by the corresponding action name. For instance, $\mathit{TakeBus}(b_1, s_1)$ refers to the solid edge between $s_1$ and $s_2$.

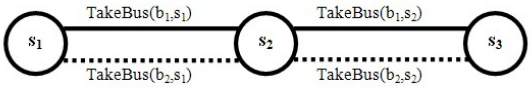



Fig. 5: An example bus line configuration

As illustrated in Figure 6, the basic model produces unsatisfactory solutions when there obviously exists better ones. The partial solution "$\mathit{TakeBus}(b_1, s_1) \Rightarrow \mathit{TakeBus}(b_2, s_2)$" introduces a redundant bus change from $b_1$ to $b_2$. To prune the search space and speed up the verification, we have to restrict that a user is not going to change a bus if it is not necessary. This constraint can be easily captured by adding a new method "*bool IsRedundent(BusLine CurrentBus,int CurrentStop)*" to the defined type $\langle \mathit{BusLine} \rangle$. In the guard condition of process *BusLine2*, we can define a contraint *!Line2.IsRedundent(currentBus, currentStop)* to avoid this transition if the change to *Line2* is considered redundant.


Fig. 6: A solution produced by the basic model

The criteria for deciding whether an action $\mathit{TakeBus}(b_i, s_j)$ is redundant or not given the current bus line is $b_k$ can be formulated as follows.

*Definition 3:* An action $\mathit{TakeBus}(b_i, s_j)$ is not redundant if one of the followings holds:

1) $b_i = b_k$
2) $b_i \in t(s_j) \wedge b_k \in t(s_j) \wedge b_i(s_j) \neq b_k(s_j) \wedge \exists m \in \mathbb{N}_1$, $b_i(s_j)^{-m} \neq b_k(s_j)^{-m}$
3) 1 and 2 do not hold and $b_i(s_j) \neq b_k(s_j) \wedge b_i^{-1}(s_j) \neq b_k^{-1}(s_j)$

Definition 3 can be casually interpreted as, "when a user is going to change to a different bus that does not form a special pattern with the current bus as shown in Figure 7 and shares the same previous stop or next stop with the current bus, the change is considered redundant.

The basic idea is to stay on one bus as long as possible. This can be enforced by simply ignoring the transitions to a bus having the same previous stop as the current one, because the transition to that bus should happen earlier (not necessarily from the current bus) or does not happen at all. As is shown in Figure 8a, the partial solution "$\mathit{TakeBus}(b_2, s_1) \Rightarrow \mathit{TakeBus}(b_1, s_2)$" is not valid as at $s_2$, $b_1$ and $b_2$ have the same previous stop $s_1$. A valid path is "$\mathit{TakeBus}(b_1, s_1) \Rightarrow \mathit{TakeBus}(b_1, s_2)$". Similarly, the transitions to a bus having the same next stop as the current one should also be avoided, because the transition can happen later (not necessarily from the current bus) or does not happen at all. As shown in Figure 8b, the partial solution "$\mathit{TakeBus}(b_1, s_1) \Rightarrow \mathit{TakeBus}(b_2, s_2)$" is not valid as at $s_2$, $b_1$ and $b_2$ have the same next stop $s_3$. A valid path is "$\mathit{TakeBus}(b_1, s_1) \Rightarrow \mathit{TakeBus}(b_1, s_2)$".

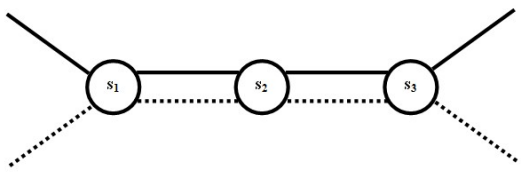
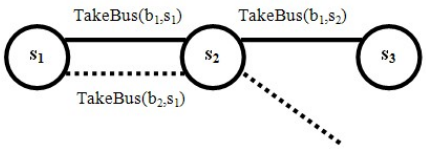
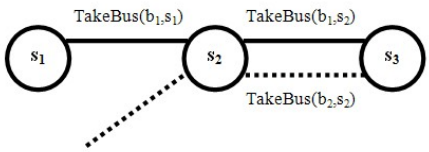



Fig. 7: Special pattern of two overlapping bus lines




(a) Same Previous Stop




(b) Same Next Stop

Fig. 8: Redundant bus changes

However, after enforcing these two basic rules, the transition can never happen between two lines forming the special pattern illustrated in Figure 7. When two bus lines form such a overlapping pattern, a bus change at the end of the overlapping segment, which is $s_3$ in this case, is not considered redundant. The reason why we force the bus change to occur at the end of the overlapping segment is that this ensures that necessary bus change happens only once within the overlapping range.

### D. Performance Evaluation

In this subsection, we evaluate the performance as well as the solution quality of the two modified planning models discussed in the previous sections. We tested all (3660) starting stop and destination stop combinations on the three models. The length of the shortest solution was got by solving the shortest path problem using Dijkstra algorithm after we converted the original map to a directed graph with path cost 1 for each edge. Table IV shows the comparison results.

In the table, all values are average among the 3660 test cases. From the comparison, we can see that the *search space*

TABLE IV: Comparison results of three route planning models

| Model | States | Transitions | Time(s) | Memory(KB) | Cost | Length |
|---|---|---|---|---|---|---|
| Basic | 1029.46 | 1070.93 | 0.0448 | 11119.91 | 58.23 | 5.51 |
| Cost | 1125.31 | 1169.82 | 0.0483 | 11281.58 | 56.02 | 5.59 |
| Prune | 158.48 | 185.77 | 0.0179 | 9197.95 | 56.79 | 5.51 |

*pruning* model performs the best in terms of execution time and memory space. In fact, a large portion of redundant transitions is pruned and the search space is reduced to a minimal. At the same time, the *search space pruning* model also preserves the make-span optimality. In addition, the model also produces low cost plans with an average total cost of 56.79 which is slightly higher than the optimal value 56.02. Among all of them, 89.6% of the solutions are in fact cost-optimal. The *cost function* model guarantees the lowest total cost as it is designed to do so. However, it is a little inefficient on the memory usage, as the plan metric optimization is indeed expensive. Some solutions are not the shortest as the *Cross* actions have less cost but are still counted towards the total length of the plans.

## IV. Conclusion

In this paper, we explored the use of model checking techniques in 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 classical planning problems. In our experiments, we compared the performance and capabilities of different tools including PAT, Spin and NuSMV. PAT is proved to be most suitable for solving various kind of planning problems. The experimental results also indicated 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 planning service. We applied our approach to a case study on the "Tranport4You" IPTM system. We implemented a route planning module for the system by exploiting the model checking power of PAT. By following the formal definitions of the route planning problem, we designed a basic CSP# model. We further improved the model in two ways. One is introducing a cost function for measuring plan quality, while the other is adding in domain specific control knowledge for search space pruning. Finally, we compared the different approaches we 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. Our case study project won the "Formal Methods Award" at the Student Contest on Software Engineering in ICSE'11.

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 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.

## 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 manuscript.

[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*, ser. SAICSIT '08. New York, 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] S. Russell and P. Norvig, *Artificial Intelligence*. Pearson, 2010, ch. Classical Planning.

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

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

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

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

[15] J. Sun, Y. Liu, J. S. Dong, and C. Chen, "Integrating specification and programs for system modeling and verification," in *Proceedings of the 2009 Third IEEE International Symposium on Theoretical Aspects of Software Engineering*, ser. TASE '09. Washington, DC, USA: IEEE Computer Society, 2009, pp. 127–135.

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

[17] 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. SARA'07. Berlin, Heidelberg: Springer-Verlag, 2007, pp. 200–214.

[18] 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*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1993, pp. 248–253.