Formal Methods in System Design manuscript No.
(will be inserted by the editor)

Model Checking Approach to Automated Planning

Yi Li · Jin Song Dong · Jing Sun · Yang Liu · Jun Sun

Received: date / Accepted: date

Abstract Model checking provides a way to automatically explore the state space of a finite state system based on desired properties, whereas planning is to produce a sequence of actions that leads from the initial state to the target goal states. Previous research in this field proposed a number of approaches for connecting model checking with planning problem solving. In this paper, we investigate the feasibility of using an established model checking framework, Process Analysis Toolkit (PAT), as a planning solution provider for upper layer applications. To achieve this, we first carry out a number of experiments on different model checking tools in order to compare their performance and capabilities on planning problem solving. Our experimental results suggest that solving planning problems using model checkers is not only possible but also practical. We then propose a formal semantic mapping from the standard Planning Domain Description Language (PDDL) to the Labeled Transition System (LTS), based on which a planning module was implemented as a part of the PAT framework. Lastly, we demonstrate and evaluate the approach of

Y. Li
Department of Computer Science, University of Toronto, Canada
E-mail: liyi@cs.toronto.edu

J.S. Dong
Department of Computer Science, National University of Singapore, Singapore
E-mail: dcsdjs@nus.edu.sg

J. Sun
Department of Computer Science, The University of Auckland, New Zealand
E-mail: j.sun@cs.auckalnd.ac.nz

Y. Liu
School of Computer Engineering, Nanyang Technological University, Singapore
E-mail: yangliu@ntu.edu.sg

J. Sun
Singapore University of Technology and Design, Singapore
E-mail: sunjun@sutd.edu.sg

using PAT as planning service via a case study on a public transportation management system.

Keywords Model Checking · Deterministic Planning · Formal Specification & Verification

1 Introduction

Model checking [26] has emerged as a powerful automatic technique for verifying models of software and hardware systems against their specifications. The system model is exhaustively explored and checked by model checkers to ensure that the desired properties are guaranteed in all cases. In general, what we care the most about the system models 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, which represents a finite path π that leads to the violation state from the initial state s. Some model checkers are able to provide shortest counterexamples. A shortest counterexample is defined as the minimal size path π^* that leads to a state s' where the safety property is violated.

Related Works. Since late 90s, several pieces of work indicated that model checking can also be applied to the planning domain. Cimatti et al. [5] presented a decision procedure for planning problems written in a high level action language called $\mathcal{AR}$ in 1997. The decision procedure was implemented as a tool MBP using the symbolic model checking technique [24]. They translated action descriptions to propositional formulas in order to apply model checking on them. This is one of the earliest attempt in the stream of planning as model checking and the results are encouraging. Nevertheless, $\mathcal{AR}$ is no longer at the forefront of domain description language. Many advanced features like plan metrics and preferences which are very common nowadays cannot be represented using propositional logic.

Berardi and Giuseppe [2] compared the performance of the two well-known model checkers, Spin [13] and SMV [24], with some well established planners (IPP [16], which was one of the best performers in AIPS'98 competition; FF [12], which was among the best performers in AIPS'00; and TLPLAN [1], 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.

Hörne and Poll [14] investigated the feasibility of using two different model checking techniques for solving a number of classical AI planning problems. ProB [17] 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 [4], which represents models using Binary Decision Diagrams (BDDs) [3]. 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 are 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.

Clearly, a classical planning problem can be easily converted into a model checking problem. The fact that this approach is feasible is supported by [8]. In that paper, the authors suggest 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.

After all these successful attempts, the natural next step would have been serious tool development, but that requires considerable effort in the implementation. Indeed, there is no serious tool developed in the recent years. The biggest problem remaining of using model checkers for planning purposes is that there does not exist a good automatic translation from the planning domain to the required model checking domains. All prior works mentioned previously relied on manual translation in their experiments, while this process is error-prone and valuable information in the original planning models are often lost along the way. Depending on the underlying algorithms, the encoding of the problems usually has huge effects on the performance and even the quality of the solutions produced. For example, the type information in planning models can be very complicated in some cases and most of the model checkers lack direct support for types. Fig. 1 shows the type declaration written in the Planning Domain Definition Language (PDDL) [23]. In this example, a type hierarchy is declared using the basic syntax construct $\texttt{obj}_0 \cdots \texttt{obj}_n - \texttt{type}$ specifying that *place* and *locatable* belong to *object*; *locatable* has two kinds, namely *soldier* and *torch*, and so on. There is no easy way to encode such information (at least it requires careful thinking and smart manipulation if done by hand) which unfortunately has a direct impact on the

```
(:types place locatable - object
        soldier torch - locatable
        north south - place
        soldier0 soldier1 - soldier)
```

Fig. 1: Illustration of the type hierarchy in PDDL.

problem search space. Typing information places constraints on the parameters of action schemas which effectively limit the number of possible state transitions one needs to consider. Furthermore, some advanced features of PDDL are tightly integrated with the language syntax and semantics. For instance, quantifier keywords `forall` and `exists` used in *goal* and *action effect* descriptions cannot be implemented without typing. Therefore, we would like to propose a formal operational semantics for PDDL in terms of the Labeled Transition System (LTS) such that planning problems can be automatically and precisely recognized and solved by model checkers. We hope that this effort can help create more possibilities and revive the interest in this area.

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. Modern model checkers have sophisticated techniques for handling large state spaces, which are 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.

Contributions. This research is divided into two stages, corresponding to the two closely related problems that we consider, i.e., *planning via model checking* and *PAT as planning service*. We conduct 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 some sophisticated planners for certain categories of problems and the performance of model checkers can even be further improved by exploiting domain specific knowledge. We define the formal semantics of PDDL in terms of LTS and implement an automatic verification tool as a sub-module in the Process Analysis Toolkit (PAT) [32]. The planning module takes a subset of the PDDL 2.1 language as input and is able to produce both *sequential* and *optimal* plans. We further investigate the possibility of using the planning module with specifically designed searching algorithms to serve as a planning solution provider for upper layer applications. The case study on a public transportation management system demonstrates

that the *planning as model checking* approach is not only possible but also practical.

Paper Organization. The rest of the paper is organized as follows. Sect. 2 reviews and compares the performance of different planning tools. In Sect. 3, we define the formal operational semantics of PDDL as LTS. Sect. 4 presents a case study in which we implement the *planning as model checking service* approach within a real application. We compare and analyze different modeling techniques with concrete evaluation results. Sect. 5 concludes the paper and maps out the future directions.

2 Review of Tools

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.

2.1 Tools and Techniques

NuSMV. NuSMV is an extension of the symbolic model checker SMV [24] developed at the Carnegie Mellon University known as CMU SMV. NuSMV is written in ANSI C and is a joint project between the Embedded Systems Unit in the Center for Information Technology at FBK-IRST, the Model Checking group at Carnegie Mellon University, the Mechanized Reasoning Group at University of Genova and the Mechanized Reasoning Group at University of Trento. The latest version NuSMV2 is distributed under an OpenSource license [4]. Models of NuSMV are described as a hierarchy of modules which can be instantiated semantically similar to call-by-reference. NuSMV allows for Boolean, integer and enumerated types for state variables [4]. However, array indices in NuSMV must be statically evaluated to integer constants. This constraint largely limits the expressiveness of the model. The modeling for common operations on a list of objects 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 `TRANS` constraint, or the `ASSIGN` constraint where a system of equations labeled as `next(identifier):=expression` describes how the underlying finite state machine evolves over time [4]. Specifications can be expressed in both CTL and LTL. NuSMV supports several kinds of model checking modes, including CTL checking, LTL checking and invariant checking. We will compare the performance of using different model checking modes for planning in Sect. 2.2.

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 the modeling language called "Promela" (Process Meta Language). The language allows for dynamic creation of concurrent processes. Communication via message channels can be defined to be synchronous or asynchronous [13]. Promela loosely follows the Communicating Sequential Process (CSP) [10] and hence models in CSP# [30] can be converted to it with minimal efforts. Guarded expressions are well supported, therefore 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 run-time options for simulation and verification that can be explored. Spin allows users to prune the search space using "never-claims" which are equivalent to safety properties. With this method it becomes possible to quickly verify whether a given safety property holds in the context of the model, even when a complete verification is considered to be infeasible [13]. After verification is finished, Spin is able to perform a simulation guided by the error trail. In the simulation mode, step-by-step display of the counterexample trace is better supported compared with NuSMV.

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

PAT. Process Analysis Toolkit (PAT) [20,32,22,21] 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# [30,31], which is an extension to Communicating Sequential Process (CSP) [10] by embedding data operations. CSP# combines high-level compositional operators from process algebra with program-like codes, which makes the language much more expressive. Other supported modeling languages in PAT include Stateful Timed CSP [36] for real time systems, PCSP [39] for modeling probabilistic behaviors and PRTS [38] for modeling of current, real-time and probabilistic systems with hierarchical structures.

Most importantly, PAT supports the verification of trace refinement checking [29,40], linear temporal logic verification with fairness assumption [35], bounded model checking [33,34], fair model checking of parameterized systems [37], BDD-based discrete analysis of timed systems [25] and assume-guarantee model checking [18,19].

One of the unique features of PAT is that it allows users to define static functions and data types as C# libraries [28]. 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 in the modeling of some special algorithms.

Metric-FF. Metric-FF [11] 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. Metric-FF accepts two searching parameters g and h for assigning different 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.

SatPlan. SatPlan [15] 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.

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 finding a plan with the minimum total action cost.

2.2 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. The enemy is chasing them and in the middle of the night. They arrive at a bridge that spans a river

0	1	2
3	4	5
6	7	8

Fig. 2: Goal setting of *the sliding game problem*.

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 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.

– *The sliding game problem*, is sometimes also referred as *the eight-tiles problem*. There are eight tiles, that are numbered from 1 to 8 and 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 this puzzle is to arrange the eight tiles into increasing order as shown in Fig. 2.

Note that *the bridge crossing problem* is a plan existence problem with constraints on the total time. The goal is to find a feasible plan that can be executed within 60 minutes. There is no plan optimization involved in this problem. PAT is able to find the *shortest witness trace* by using the breadth-first search algorithm, i.e., the returned counterexample trace is guaranteed to be the shortest one. Otherwise, if a depth-first search is performed then the first counterexample trace encountered is reported. Therefore, for *the bridge crossing problem* where shortest witness trace is not required, we use the depth-first search mode. For *the sliding game problem*, where an optimal solution is expected, we use the *shortest witness trace* option instead. The counterexamples provided by NuSMV are always the shortest, while this is not the case for Spin. We use NuSMV to generate optimal solutions for *the sliding game problem*, and collect the performance data of Spin only for reference.

To measure execution time 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

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

Table 1: Time cost of each soldier.

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. In contrast, execution time obtained from other tools was rounded to two decimal places.

The experimental results are presented in the following two subsections, where INVAR denotes using invariant mode of NuSMV, LTL/CTL denotes using LTL/CTL model checking mode of NuSMV, RW indicates that PAT is under the *reachability-with* mode, and DFS and BFS denote PAT using depth-first and breadth-first model checking algorithms respectively. Time is measured in seconds unless otherwise indicated.

2.2.1 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 search algorithms, PAT also supports *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, for fair comparison, we recorded the time taken by PAT under the depth-first search mode. We ran Metric-FF on *the bridge crossing problem* with parameters $g = 100$ and $h = 1$, which emphasis the plan quality over the performance to increase the possibility of getting an solution within the time limit.

This set of experiments are tailored to show how the model checkers compete on plan existence problems that deal with numerical constraints. The time cost of each soldier is listed in Table 1 and the experiment results are summarized in Table 2. In 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. Symbol m in the table means that the particular tool (mode) ran out of memory and did not find 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 obtained from Metric-FF is still not guaranteed. So the Metric-FF column is only for reference.

When the number of soldiers reaches 8, NuSMV is not able to build a model according to the model descriptions due to memory shortage. This is likely to be related to the inefficient encoding that we have to use in the models

#Soldiers	Time*	Metric-FF	PAT		NuSMV			Spin
			RW	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

Table 2: Experimental results for *the bridge crossing problem*.

because of the restrictions on array indices. The invariant checking mode performs generally better than CTL and LTL checking modes, because CTL and LTL model checking algorithms have to explore a search space that involves both the model and the properties. But invariant (reachability) checking only explore the model's space[1].

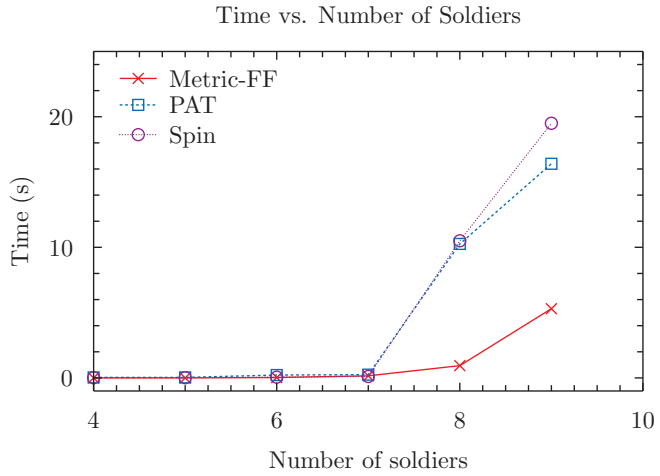


Fig. 3: Execution time comparison of PAT, Spin and Metric-FF on *the bridge crossing problem*.

Fig. 3 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

[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.

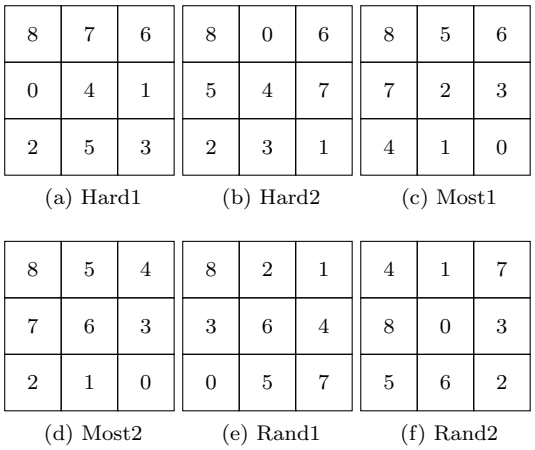


Fig. 4: Initial configurations of *the sliding game problem* instances.

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. 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 problem instances, e.g., 8 and 9 soldiers, PAT starts to perform better that Spin.

2.2.2 The Sliding Game Problem

Optimal AI planning is a PSPACE-complete problem in general. For many problems studied in the planning literature, the plan optimization problem has been shown to be NP-hard [9]. The *sliding game problem* 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 *sliding game problem* is NP-hard [27]. The difficulties of the problem instances are measured by the lengths of their optimal solutions. There is also an approximated measurement called the *Manhattan distance* (MD), 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. The MD of a problem instance is the sum of the MDs of all eight tiles to their positions in the goal setting. 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 largest number of 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 obtained

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

Table 3: Experimental results for *the sliding game problem*.

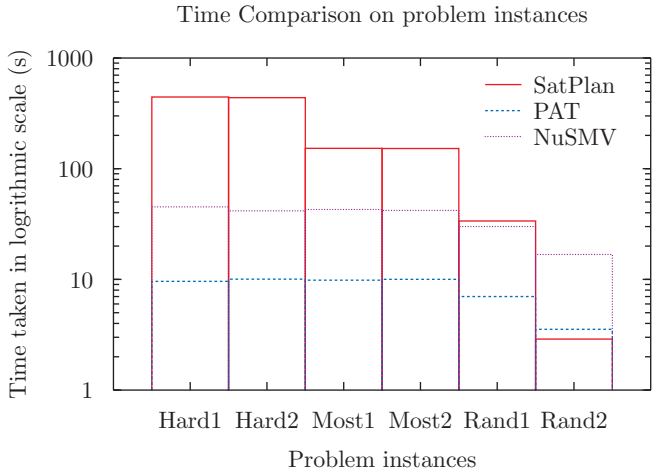


Fig. 5: Execution time comparison of PAT, NuSMV and SatPlan on *the sliding game problem* shown on a logarithm scale.

from SatPlan are used only as reference. The initial configurations of all the six problem instances are shown in Fig. 4.

The results are summarized in Table 3. In the table, "> 600" indicates that no solution was found after 10 minutes. The column "L*" and "MD" show the length of the optimal solutions and the *Manhattan distance* of the problem instance respectively. 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 the given 10 minutes. Similarly as in the previous example, the invariant checking mode performs much better compared to the other two modes. From Fig. 5 we can clearly see that the execution time of SatPlan for different problem instances varies significantly. The performance of SatPlan depends largely on the length of the optimal solutions. "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 performs better than NuSMV and SatPlan on plan optimization problems with our best effort in modeling the same problems with different domain languages. 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.

3 Operational Semantics of PDDL

Manual encoding of planning problems in the respective model description languages is cumbersome and error-prone. 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 can also be different when the environment changes. This is where the concept of *replan* comes into play. Using model checkers as service enables real-time *replanning* by generating problem descriptions at runtime, and modifying models with the latest environment variables. The idea of using model checkers as planning service is only possible if there exists an automatic tool that connects planning and model checking domains.

To achieve this goal, we define a formal operational semantics of the planning language in terms of executable systems, which can be understood directly by model checkers. Essentially, we change the planning problem to a verification problem.

In this section, we describe the operational semantics of PDDL in terms of LTS. Our goal is to provide a guide for implementing an automatic translator from PDDL to model description languages recognized by model checkers. We make two basic assumptions on the input language:

- 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 [6] when the original model is written without typed arguments.
- Durative actions are absent (referred as *simple planning instance* in [7]).

3.1 PDDL Formalization

PDDL is the standard language for describing classical planning problems and is widely used by many planners. Essentially following [7], a PDDL simple planning instance consists of two types of files, i.e., the domain and the problem.

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

Fig. 6: PDDL action schema for taking bus.

Definition 1 (Simple Planning Instance) A simple planning instance is defined to be a pair $I = (Dom, Prob)$, where $Dom = (Fs, Rs, As, arity)$ is a 4-tuple consisting of (finite sets of) function symbols, relation symbols, actions (non-durative), and a function mapping all of the symbols to their respective arities. $Prob = (Os, Init, G)$ is a triple consisting of the objects in the domain, the initial state specification and the goal state specification.

Actions are grouped as a set of action schemas. The schema consists of the action name, a list of parameters, a precondition and effects. The PDDL code in Fig. 6 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*. The type that follows each parameter constraints the type of objects that the variable can be instantiated to. In the following sections, we only consider flattened and fully grounded actions, meaning that the actions in As do not contain conditional effects and quantifiers. There are standard ways to get rid of conditional effects and instantiate action schemas using proper objects as described in [7]. PDDL 2.1 also allows for numerical optimization criteria to be specified. In the *TakeBus* example, the values of `time-cost` and `money-cost` are increased in the effects. The optimization criterion, also known as the *plan metric*, consists of numerical expressions to be maximized or minimized, e.g., `(:metric minimize(time-cost))` requires that the value of the function `time-cost` to be minimized.

3.2 Operational Semantic

The *primitive numeric expressions* of a planning instance, $PNEs$, are the terms constructed by applying the function symbols in Fs to the objects drawn from Os. Similarly, the *atoms* of a planning instance, denoted by $Atoms$, are the expressions formed by applying the relation symbols in Rs to the objects (with respect to arities and type constraints).

Definition 2 (System State) A system state is composed of two components (F, R) where $F \in PNEs \times \mathbb{R}_{\perp}$ maps primitive numeric expressions to

their values ($\perp$ denotes *undefined value*) and $R \in Atoms \times \{true, false\}$ maps atoms to Boolean values.

We refer to F and R as the valuation functions. A system transition is of the form $(F, R) \xrightarrow{a} (F', R')$ where a is an action in As. For each flattened and grounded action a,

- Pre_a denotes the precondition of a, which is a propositional expression over $Atoms$, e.g., `(and (At b from) (At p from))`;
- Add_a is the positive effect of a, which is the set of ground atoms that are asserted as positive literals, e.g., `(At b to)`;
- Del_a is the negative effect of a, which is the set of ground atoms that are asserted as negative literals, e.g., `(not (At b from))`;
- NE_a is the numeric effect of a, which is the set of assignment propositions, e.g., `(assign time 0)`.

The set of all effects of action a is written as Eff_a. We use the notation $(F, R) \overset{\mathit{Eff}_a}{\rightarrowtail} (F', R')$ for a *pseudo system transition* caused by the effects of action a. A pseudo transition by Eff_a can be understood as a system transition caused by an action a' such that $\mathit{Eff}_{a'} = \mathit{Eff}_a$ and $Pre_{a'} = true$. A group of pseudo transitions can be aggregated to form a real state transition by following the action semantics.

The operational semantics is systematically defined by associating a set of firing rules with each PDDL language construct. Fig. 7 illustrates the firing rules related to actions. We omit the rules for initializing initial states here, since they are very similar to what we have for actions. The semantics of Add_a and Del_a can be mapped to the rewriting of the corresponding values of atoms (*PosEffect* and *NegEffect* in Fig. 7). Positive effects update atom values to *true* while negative effects update them to *false*. There are five kinds of numeric effects, i.e., *assignment*, *increase*, *decrease*, *scale up*, and *scale down*. We have shown the semantics for the assignment effect as in Rule *AssignEffect*. The rest simply correspond to the shorthand operators `+=`,`-+`,`*=` and `/=`. Sometimes a number of effects are grouped by the keyword *and*. For any valid action, the effects in a group should be consistent with each other. Therefore, we are able to define the combined effects of two sub-effects as the function rewriting of one over the other (Rule *And*). Rule *Precond* captures how applicability of action is checked, i.e., the state transition is executed if and only if the precondition for the corresponding action holds.

Example 1 Recall the action schema *TakeBus* in Fig. 6, a corresponding state transition in LTS would be,

$$(F, R) \xrightarrow{TakeBus} (F \oplus \{\, tcost' \mapsto tcost + 10,\ mcost' \mapsto mcost + 2\},\ R \oplus \{\, At(b, from) \mapsto true,\ At(b, to) \mapsto true,\ At(p, to) \mapsto true,\ At(p, from) \mapsto false\},$$

where the valuation functions in F and R are updated according to $\mathit{Eff}_{TakeBus}$. This transition is enabled if and only if $(F, R) \models Pre_{TakeBus}$.

$$\frac{r \in \mathit{Atoms}}{(F,R) \overset{r}{\rightarrowtail} (F, R \oplus \{r \mapsto \mathit{true}\})} \quad [\ \mathit{PosEffect}\ ]$$

$$\frac{r \in \mathit{Atoms}}{(F,R) \overset{(\mathit{not}\ r)}{\rightarrowtail} (F, R \oplus \{r \mapsto \mathit{false}\})} \quad [\ \mathit{NegEffect}\ ]$$

$$\frac{f \in \mathit{PNEs},\ e\ \mathit{evaluates\ to}\ \mathbb{R}_\perp}{(F,R) \overset{(\mathit{assign}\ f\ e)}{\rightarrowtail} (F \oplus \{f \mapsto \mathit{Eval}(e,F)\}, R)} \quad [\ \mathit{AssignEffect}\ ]$$

$$\frac{e_1 \in \mathit{Eff}_a,\ e_2 \in \mathit{Eff}_a,\ (F,R) \overset{e_1}{\rightarrowtail} (F',R'),\ (F,R) \overset{e_2}{\rightarrowtail} (F'',R'')}{(F,R) \overset{(\mathit{and}\ e_1\ e_2)}{\rightarrowtail} (F' \oplus F'', R' \oplus R'')} \quad [\ \mathit{And}\ ]$$

$$\frac{(F,R) \models \mathit{Pre}_a,\ (F,R) \overset{\mathit{Eff}_a}{\rightarrowtail} (F',R')}{(F,R) \xrightarrow{a} (F',R')} \quad [\ \mathit{Precond}\ ]$$

Fig. 7: Operational semantics for PDDL actions where $a \in As$, $\oplus$ is function rewriting operator, *Eval* evaluates an assignment proposition.

Theorem 1 (Correctness of Action Mapping) *Let $a \in As$ be a fully flattened and grounded action, (F,R) and (F',R') be two system states, $(F,R) \xrightarrow{a} (F',R')$ appears in the labeled transition relationship if and only if (F,R) satisfies the precondition of a and executing a updates the system states to (F',R').*

Let the initial state be $\mathit{Init} = (F_0, R_0)$, goal state be $G = (F_{n+1}, R_{n+1})$, the transition system of a simple planning instance is a LTS $L_R^F = (S, \mathit{Init}, \rightarrow)$ where S is the set of reachable system states and $\rightarrow$ is a labeled transition relationship conforming to the operational semantics presented in Fig. 7. A solution to the instance is a finite sequence of system states conforming to the transition relations in L_R^F.

Theorem 2 (Correctness of Plan Mapping) *Sequence $\langle a_0, \ldots, a_n \rangle$ is a solution to the planning problem I if and only if there exists a sequence $\pi = \langle s_0, \ldots, s_{n+1} \rangle$ where $s_0 = \mathit{Init}$, $s_{n+1} \in G$, $s_i \in S$ and $s_i \xrightarrow{a_i} s_{i+1}$ for all $0 \leq i \leq n$.*

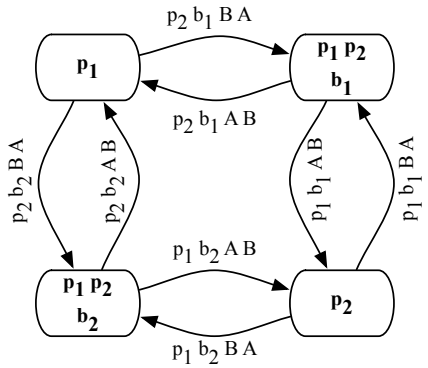


Fig. 8: Illustration of the LTS state transitions in the *TakeBus* example.

Proof

According to Thm. 1, executing action a_i changes the system state from s_i to s_{i+1}. By simple induction, executing the a finite sequence of actions $\langle a_0, \ldots, a_n \rangle$ updates the system state from $s_0 = \mathit{Init}$ to $s_{n+1} \in G$, which corresponds exactly to the definition of a *plan* to I (sketch).

A plan is optimal if and only if there does not exist another such sequence π' such that $\mid \pi' \mid < \mid \pi \mid$. A plan maximizes a numerical expression e if and only if there does not exist another sequence π'' such that $\mathit{Eval}(e, s_n) < \mathit{Eval}(e, s''_m)$, where s_n and s''_m are the last state in π and π'' respectively.

Example 2 Consider the *TakeBus* example with two buses b_1 and b_2, two passengers p_1 and p_2 and two stops A and B. Ignore functions at the moment for simplicity. Fig. 8 shows a part of the state transition for the example. In the diagram, each node is labeled with the objects (buses and passengers) at stop A. The objects that do not appear in the node are at stop B. Every arrow is labeled with the parameters for the action $\mathit{TakeBus}(?p, ?b, ?\mathit{from}, ?\mathit{to})$, where $?p$ is a passenger, $?b$ is a bus; $?\mathit{from}$ and $?\mathit{to}$ are the source and destination stops.

3.3 Implementation

We have implemented a planning tool that supports PDDL as a module of the PAT model checking framework. The tool is available for download at `http://www.comp.nus.edu.sg/~pat/plan`. Fig. 9 demonstrates the architectural design of the planning module. The editor is featured with powerful

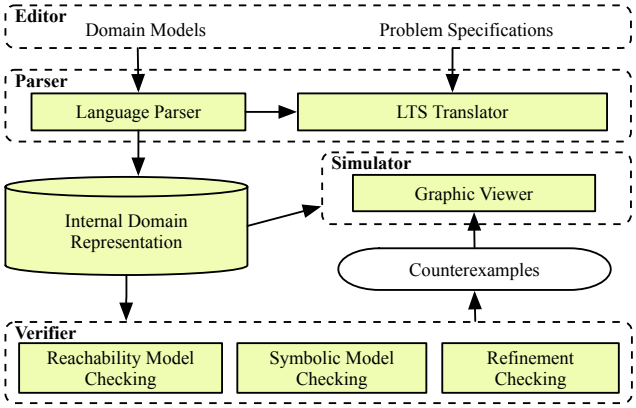


Fig. 9: Architecture of the planning module in PAT

text editing and syntax highlighting. Multiple domain and problem files can be edited as a group. The parser transforms the PDDL models as well as the problem instances into the internal domain representation. The simulator allows users to perform various simulation tasks on the input models, including complete state graph generation, automatic simulation, user interactive simulation and trace replay. The module relies on various underlying model checking techniques provided by the PAT verifier. The counterexample found by the verifier can be displayed as a planning solution.

We have tested the tool on a number of planning problems including the sliding game and bridge crossing problems and receive good results. The formal operational semantics we defined earlier enables the automatic mapping from the planning domain to the model checking domain. We are one step closer to the goal of model checking as planning service. The prototype we have built could serve as a platform for experimenting on variate heuristics and algorithms in the future. We hope our efforts in implementing the tool can help revive the interests in this research area.

4 Case Study – Transport4You

In this section, we present a case study on "Transport4You" which was submitted 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 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

[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`.

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 architectural design of the IPTM system is shown in Fig. 10.

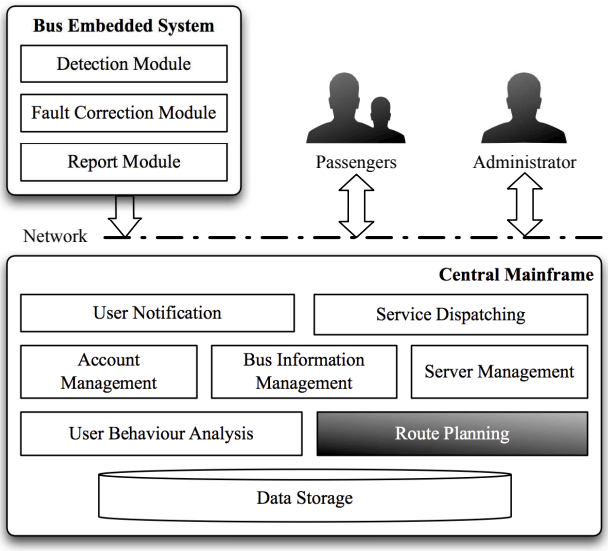


Fig. 10: System architecture diagram of the "Transport4You" IPTM system.

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 report detection results 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.

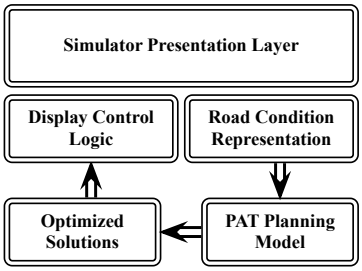


Fig. 11: Simulator architecture diagram.

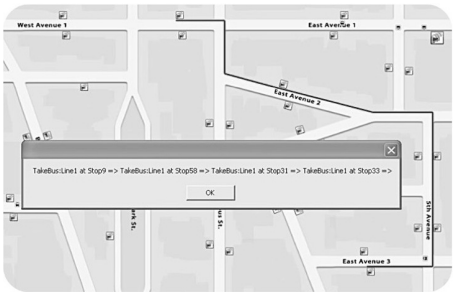


Fig. 12: Simulator screen shot of route planning results.

As is shown in Fig. 11, the simulator generates a new planning domain model according to the latest road conditions and bus line configurations, whenever the environment changes. Users choose their starting positions and destination stations on the simulator interface. By clicking the "Plan" button, the system generates a problem instance according to what have been chosen and pass it to the PAT planning module. After interpreting the returned results, the simulator is able to display the planned route and detailed instructions to users through the display interface, as is shown in Fig. 12. The *route planning* module works correctly even when there are real time changes on road conditions. When interruptions of bus lines are detected, the administrator updates the road condition database immediately (automatically or manually). The planning results are, therefore, guaranteed to be accurate based on the most updated data.

4.1 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 3 (Route Planning Task) 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. $\forall\, s \in s_{start} \forall\, b \in B, s \in dom(b) \longrightarrow b^{-1}(s) = \alpha$. $\forall\, s \in s_{end} \forall\, b \in B, s \in dom(b) \longrightarrow b(s) = \beta$. $\forall\, b \in B, b^{-1}(\alpha) = \alpha \wedge b(\beta) = \beta$.

- $t : S \rightarrow B_S$ is a function where $B_S \subseteq B$. $t(s)$ is the set of available bus lines at stop s, i.e., $B_S = \{b_i \in B \mid s \in dom(b_i)\}$.
- $c : S \rightarrow S$ 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 the passenger is stop s.

The definition is 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 4 (Route Planning Domain) 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 an atom $s \in S$, 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.

4.1.1 Environment Variables

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

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

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

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

In the above code, the instantiation of $\langle BusLine \rangle$ takes in two parameters, including a sequence of bus stops and an integer that indicates the bus 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.

4.1.2 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 (which is the current location of the passenger), while *currentBus* is a temporary variable storing the enabled bus line in the execution of the current action.

$$
\begin{aligned}
&var\ currentStop = Stop5;\\
&var\ B0 = [-2];\\
&var\langle BusLine\rangle currentBus = new\ BusLine(B0, -1);
\end{aligned}
$$

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 it is initialized with some negative integer to avoid confusion.

4.1.3 State Transition Functions

Now we 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 (a switch-case like conditional branch). 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.

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

The process *takeBus()* chooses one of the bus lines that available in the current bus stop and hands over control to it. For example, at *Stop5*, there

are two bus lines available, namely *BusLine1* and *BusLine5*. Then we define processes that describe the behaviours of bus lines (*BusLine1* to *BusLine14*).

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

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 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.

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

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 a state transition, the process also hands over its control to *takeBus()*. 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.

4.1.4 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 the user. It is *Stop53* in this example.

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

4.2 The Cost Function Approach

The basic model we discussed earlier is able to solve the Route Planning problem. It even provides optimal plans in terms of 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 to measure the cost incurred with the execution of an action. 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.

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, 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 τ with its cost value (line 8). In the following search, if the cost of the current partial plan exceeds τ, 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 Fig. 13, where *working* is

```
1: working ← ∅
2: current ← InitialStep
3: τ ← ∞
4: repeat
5:    value ← EvaluateExpression(current)
6:    if current.SatisfyGoal() then
7:       if value < τ then
8:          τ ← value
9:       end if
10:   end if
11:   if value > τ then
12:      continue
13:   end if
14:   for all step ∈ current.MakeOneMove() do
15:      working.Enque(step)
16:   end for
17: until working.Count() ⩽ 0
```

Fig. 13: The newBFSVerification() algorithm.

the task queue used in the BFS; τ is the temporary variable to store the current best value explored 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 set of all possible outgoing transitions from the current state. The state pruning happens if *value* is greater than τ (line 12).

4.3 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 Fig. 14, 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.

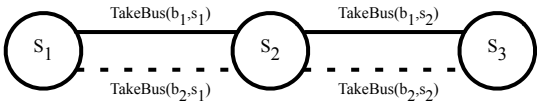


Fig. 14: An example of bus line configuration.

As illustrated in Fig. 15, the basic model produces unsatisfactory solutions when there 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 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 $\langle BusLine \rangle$. 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.

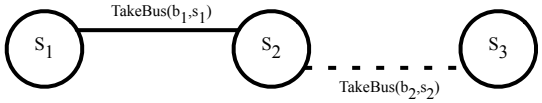


Fig. 15: A unsatisfactory solution produced by the basic model.

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

Definition 5 (Redundant Action) An action $TakeBus(b_i, s_j)$ 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 5 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 Fig. 16 and shares the same previous stop or next stop with the current bus, the change is considered redundant.

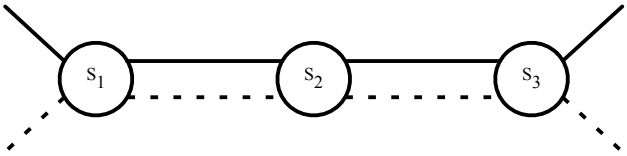
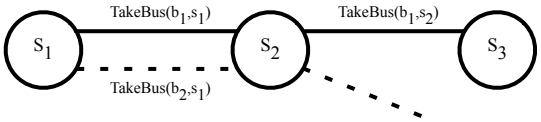


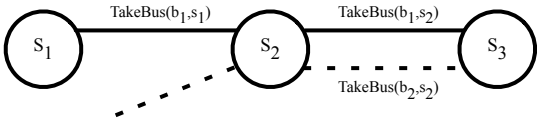
Fig. 16: Special pattern of two overlapping bus lines.

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 Fig. 17a, the partial solution "$TakeBus(b_2, s_1) \Rightarrow TakeBus(b_1, s_2)$" is not satisfactory as at s_2, b_1 and b_2 have the same previous stop s_1. A good path is "$TakeBus(b_1, s_1) \Rightarrow TakeBus(b_1, s_2)$". 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 Fig. 17b, the partial solution "$TakeBus(b_1, s_1) \Rightarrow TakeBus(b_2, s_2)$" is not satisfactory as at s_2, b_1 and b_2 have the same next stop s_3. A good path in this case is "$TakeBus(b_1, s_1) \Rightarrow TakeBus(b_1, s_2)$".



(a) Same Previous Stop



(b) Same Next Stop

Fig. 17: Redundant bus changes.

However, after enforcing these two basic rules, the transition can never happen between two lines forming the special pattern illustrated in Fig. 16. 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 that we force the bus change to occur at the end of the overlapping segment is that this ensures that a necessary bus change only happens once within the overlapping range.

4.4 Performance Evaluation

In this subsection, we evaluate the performance as well as the solution quality of the two modified planning models discussed previously. We tested all (3660) starting stop and destination stop combinations on the three models. The length of the shortest solution was calculated 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 4 shows the comparison results.

	States	Time(s)			Memory(KB)			Cost	Length
		Avg.	Max	Mean	Avg.	Max	Mean		
Basic	1029	0.045	0.596	0.028	11120	31521	10088	58.23	5.51
Cost	1125	0.048	0.599	0.030	11282	34704	10191	56.02	5.59
Prune	158	0.018	0.050	0.017	9198	10568	9118	56.79	5.51

Table 4: Comparison results of three route planning models.

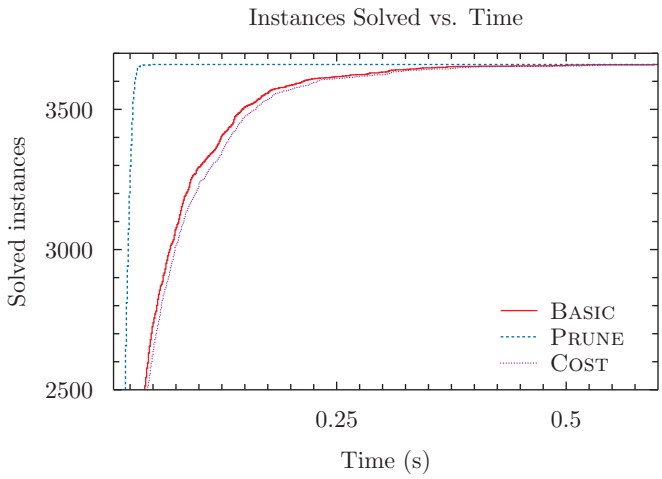


Fig. 18: No. of instances solved vs. time for three route planning models.

In the table, column under "States" is the average number of states searched when finding the plan among the 3660 test cases. Column "Time" and "Memory" list the statistics for three models including the average, maximum and median values of time as well as memory consumption. From the comparison, we can easily see that the Prune model performs the best in terms of execution time and memory consumption. In fact, a large portion of redundant transitions are pruned and the search space is reduced to a minimal. In Fig. 18, it is clear that the Prune model solved all instances in a very short time (less than 0.05 seconds) while the other two spent more time on the harder problems. But still, most of the instances (82%) were solved in less than 0.075 seconds.

Meanwhile, the Prune model also preserves the make-span optimality while maintaining low total costs. The Prune model produced plans with an average total cost of 56.79 which is slightly higher than the optimal value 56.02 achieved by the Cost model. The average cost for the Basic model is 58.23, which is much higher than the other two. Note that 89.6% of the solutions from the Prune model are cost-optimal compared with 65.7% from the Basic model. We believe that this is a strong indication that the control knowledge is helpful in not only expediting the searching but also maintaining lower action costs. 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 requires exploration of a larger state space. Some solutions are not the shortest because the *Cross* actions have less cost but are still counted towards the total length of the plans.

To summarize, the Prune model is very efficient in solving the route planning problem and the experimental results indicate that the algorithm effectively controls the search space while maintain high plan qualities. We

believe that this approach can be applied to larger systems and the combination with the Cost model can help produce cost-optimal plans if needed. The prototype used in the experiment and more detailed results can be found at `http://www.comp.nus.edu.sg/~pat/plan`.

5 Conclusion

In this paper, we explored the use of model checking techniques in the AI planning domain. We believe our work has established a good starting point in this direction towards more practical applications. We first examined the feasibility of using different model checkers in solving classical planning problems. In our experiments, we compared the performance and capabilities of different model checking tools including PAT, Spin and NuSMV. PAT is proved to be most suitable for solving a larger range of planning problems. The experimental results also indicate that some model checkers, for example PAT, can even compete with sophisticated planners in certain domains.

Based on the performance evaluation, we suggested the approach of using PAT as planning service. Lack of automatic tool support for translating planning domain languages into models that can be recognized and solved by model checkers has long been a critical problem in the *planning as model checking* stream. Manual translation is often inaccurate and error-prone. We presented a formal semantic mapping from PDDL to LTS which enables model checkers to solve planning problems. A planning module that directly works on PDDL was implemented in the PAT framework. We also applied our approach to a case study on the "Tranport4You" IPTM system. We started with a basic model based on the system specifications and further improved the model in two ways. One of them introduces cost functions to optimize plan quality, while the other exploits domain specific control knowledge for search space pruning. Finally, we compared different approaches based on their execution time, memory efficiency and plan quality. The case study project won the "Formal Methods Award" at the Student Contest on Software Engineering of the 33^{rd} International Conference on Software Engineering (ICSE) in Hawaii, USA in May 2011.

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. In addition, we are interested in extending the planning module in PAT to support a more recent version of PDDL, version 3.1 with *durative actions* and *action preferences*. We would also like to explore more heuristics and implement a mechanism to automatically exploit domain specific knowledge. Last but not least, we recommend that more research should be done on applying model checking as planning services. The applications of this technique can be extended to a larger range on real-life problems appeared in various fields.

Acknowledgements The authors would like to thank their teammates in the ICSE 2011 SCORE contest, Mr. Hang Yang and Mr. Huanan Wu, for their valuable contributions to the implementation of the "Transport4You" system. This work is partially supported by the research grant TDSI-11-002-1A "Model Checking System of Systems" and NAP project "Formal Verification on Cloud".

References

1. Bacchus, F., Kabanza, F., Sherbrooke, U.D.: Using Temporal Logics to Express Search Control Knowledge for Planning. Artificial Intelligence **16**, 123–191 (2000)
2. Berardi, D., Giacomo, G.D.: Planning via Model Checking: Some Experimental Results (2000). Unpublished manuscript
3. Bryant, R.E.: Symbolic Boolean Manipulation with Ordered Binary-decision Diagrams. ACM Computing Surveys **24**, 293–318 (1992)
4. Cavada, R., Cimatti, A., Jochim, C.A., Keighren, G., Olivetti, E., Pistore, M., Roveri, M., Tchaltsev, A.: NuSMV 2.5 User Manual. CMU and ITC-irst (2005)
5. Cimatti, A., Giunchiglia, E., Giunchiglia, F., Traverso, P.: Planning via Model Checking: A Decision Procedure for $\mathcal{AR}$. Recent Advances in AI Planning pp. 130–142 (1997)
6. Fox, M., Long, D.: The Automatic Inference of State Invariants in TIM. Journal of Artificial Intelligence Research **9**, 367–421 (1998)
7. Fox, M., Long, D.: PDDL2.1: An Extension to PDDL for Expressing Temporal Planning Domains. Journal of Artificial Intelligence Research **20**(1), 61–124 (2003)
8. Giunchiglia, F., Traverso, P.: Planning as Model Checking. In: S. Biundo, M. Fox (eds.) Recent Advances in AI Planning, *LNCS*, vol. 1809, pp. 1–20. Springer Berlin Heidelberg (2000)
9. Gregory, P., Long, D., Fox, M.: A meta-CSP Model for Optimal Planning. In: Proceedings of the 7th International conference on Abstraction, reformulation, and approximation, SARA '07, pp. 200–214. Springer-Verlag, Berlin, Heidelberg (2007)
10. Hoare, C.A.R.: Communicating Sequential Processes. Communications of the ACM **21(8)**, 666–677 (1978)
11. Hoffmann, J.: Extending FF to Numerical State Variables. In: Proceedings of the 15th European Conference on Artificial Intelligence (ECAI-02), pp. 571–575. Wiley, Lyon, France (2002)
12. Hoffmann, J., Nebel, B.: The FF Planning System: Fast Plan Generation Through Heuristic Search. Journal of Artificial Intelligence Research **14**, 253–302 (2001)
13. Holzmann, G.J.: The SPIN Model Checker: Primer and Reference Manual. Addison-Wesley Professional (2003)
14. Hörne, T., van der Poll, J.A.: 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, SAICSIT '08, pp. 114–123. ACM, New York, NY, USA (2008)
15. Kautz, H.A., Selman, B., Hoffmann, J.: SatPlan: Planning as Satisfiability. In: Abstracts of the 5^{th} International Planning Competition (2006)
16. Koehler, J., Nebel, B., Hoffmann, J., Dimopoulos, Y.: Extending Planning Graphs to an ADL Subset. In: S. Steel, R. Alami (eds.) Recent Advances in AI Planning, *LNCS*, vol. 1348, pp. 273–285. Springer Berlin Heidelberg (1997)
17. Leuschel, M., Butler, M.: ProB: A Model Checker for B. In: K. Araki, S. Gnesi, D. Mandrioli (eds.) FME 2003: Formal Methods, *LNCS*, vol. 2805, pp. 855–874. Springer Berlin Heidelberg (2003)
18. Lin, S.W., André, É., Dong, J.S., Sun, J., Liu, Y.: An Efficient Algorithm for Learning Event-Recording Automata. In: T. Bultan, P.A. Hsiung (eds.) Automated Technology for Verification and Analysis, *LNCS*, vol. 6996, pp. 463–472. Springer Berlin Heidelberg (2011)
19. Lin, S.W., Liu, Y., Sun, J., Dong, J.S., André, É.: Automatic Compositional Verification of Timed Systems. In: D. Giannakopoulou, D. Méry (eds.) FM 2012: Formal Methods, *LNCS*, vol. 7436, pp. 272–276. Springer Berlin Heidelberg (2012)

20. Liu, Y., Sun, J., Dong, J.S.: An Analyzer for Extended Compositional Process Algebras. In: Companion of the 30^{th} international conference on software engineering, ICSE Companion '08, pp. 919–920. ACM, New York, NY, USA (2008)
21. Liu, Y., Sun, J., Dong, J.S.: Analyzing Hierarchical Complex Real-time Systems. In: The ACM SIGSOFT International Symposium on the Foundations of Software Engineering (FSE 2010), pp. 511–527 (2010)
22. Liu, Y., Sun, J., Dong, J.S.: Developing Model Checkers Using PAT. In: Proceedings of the 8^{th} International Symposium on Automated Technology for Verification and Analysis, ATVA '10, pp. 371–377 (2010)
23. McDermott, D.V.: PDDL - The Planning Domain Definition Language. Yale Center for Computational Vision and Control (1998)
24. McMillan, K.L.: Symbolic Model Checking: an Approach to the State Explosion Problem. Ph.D. thesis, Carnegie Mellon University, Pittsburgh, PA, USA (1992)
25. Nguyen, T.K., Sun, J., Liu, Y., Dong, J.S., Liu, Y.: Improved BDD-Based Discrete Analysis of Timed Systems. In: D. Giannakopoulou, D. Méry (eds.) FM 2012: Formal Methods, *LNCS*, vol. 7436, pp. 326–340. Springer Berlin Heidelberg (2012)
26. Peled, D., Pelliccione, P., Spoletini, P.: Wiley Encyclopedia of Computer Science and Engineering, chap. Model Checking. John Wiley & Sons (2009)
27. Reinefeld, A.: Complete Solution of the Eight-puzzle and the Benefit of Node Ordering in IDA*. In: Proceedings of the 13^{th} international joint conference on Artificial intelligence, *IJCAI '93*, vol. 1, pp. 248–253. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA (1993)
28. Sun, J., Liu, Y., Cheng, B.: Model Checking a Model Checker: A Code Contract Combined Approach. In: J.S. Dong, H. Zhu (eds.) Formal Methods and Software Engineering, *LNCS*, vol. 6447, pp. 518–533. Springer (2010)
29. Sun, J., Liu, Y., Dong, J.S.: Model Checking CSP Revisited: Introducing a Process Analysis Toolkit. In: ISoLA 2008, pp. 307–322. Springer (2008)
30. Sun, J., Liu, Y., Dong, J.S., Chen, C.: Integrating Specification and Programs for System Modeling and Verification. In: W.N. Chin, S. Qin (eds.) Proceedings of the 2009 Third IEEE International Symposium on Theoretical Aspects of Software Engineering, TASE '09, pp. 127–135. IEEE Computer Society, Washington, DC, USA (2009)
31. Sun, J., Liu, Y., Dong, J.S., Liu, Y., Shi, L., André, E.: Modeling and Verifying Hierarchical Real-time Systems using Stateful Rimed CSP. ACM Transactions on Software Engineering and Methodology **22**(1), 3:1–3:29 (2013)
32. Sun, J., Liu, Y., Dong, J.S., Pang, J.: PAT: Towards Flexible Verification under Fairness. In: The 21^{th} International Conference on Computer Aided Verification (CAV 2009), pp. 709–714. Springer, Grenoble (2009)
33. Sun, J., Liu, Y., Dong, J.S., Sun, J.: Bounded Model Checking of Compositional Processes. In: 2nd IEEE Theoretical Aspects of Software Engineering Conference (TASE 2008), pp. 23–30. IEEE Computer Society (2008)
34. Sun, J., Liu, Y., Dong, J.S., Sun, J.: Compositional Encoding for Bounded Model Checking. Frontiers of Computer Science in China **2**(4), 368–379 (2008)
35. Sun, J., Liu, Y., Dong, J.S., Wang, H.H.: Specifying and Verifying Event-based Fairness Enhanced Systems. In: Proceedings of the 10^{th} International Conference on Formal Engineering Methods (ICFEM 2008), pp. 318–337. Springer (2008)
36. Sun, J., Liu, Y., Dong, J.S., Zhang, X.: Verifying Stateful Timed CSP using Implicit Clocks and Zone Abstraction. In: The 11^{th} International Conference on Formal Engineering Methods (ICFEM 2009), pp. 581–600 (2009)
37. Sun, J., Liu, Y., Roychoudhury, A., Liu, S., Dong, J.S.: Fair Model Checking of Parameterized Systems. In: Proceedings of the 6^{th} International Symposium on Formal Methods (FM 2009), pp. 123–139 (2009)
38. Sun, J., Liu, Y., Song, S., Dong, J.S., Li, X.: PRTS: An Approach for Model Checking Probabilistic Real-Time Hierarchical Systems. In: S. Qin, Z. Qiu (eds.) Formal Methods and Software Engineering, *LNCS*, vol. 6991, pp. 147–162. Springer Berlin Heidelberg (2011)
39. Sun, J., Song, S.Z., Liu, Y.: Model Checking Hierarchical Probabilistic Systems. In: J. Dong, H. Zhu (eds.) Formal Methods and Software Engineering, *LNCS*, vol. 6447, pp. 388–403. Springer Berlin Heidelberg (2010)

40. Wang, T., Song, S., Sun, J., Liu, Y., Dong, J.S., Wang, X., Li, S.: More Anti-chain Based Refinement Checking. In: T. Aoki, K. Taguchi (eds.) Formal Methods and Software Engineering, *LNCS*, vol. 7635, pp. 364–380. Springer Berlin Heidelberg (2012)