B.Comp. Dissertation

Model Checking as Planning and Service

By

Li Yi

Department of Computer Science

School of Computing

National University of Singapore

2010/2011

B.Comp. Dissertation

Model Checking as Planning and Service

By

Li Yi

Department of Computer Science

School of Computing

National University of Singapore

2010/2011

Project No: H011820
Advisor: Assoc Prof DONG Jin Song
Deliverables:
Report: 1 Volume
Source Code: 1 CD

Abstract

Model checking provides us a way to automatically verify hardware and software models specified using precise formal languages, whereas the goal of planning is to produce a sequence of actions that leads from the initial state to the desired goal states. Recently, a number of papers have reported that model checkers can also be used to solve AI planning problems. In this thesis, 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 addition, a case study on a public transportation management system has been developed to demonstrate the idea of using model checking as planning service. 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.

Subject Descriptors:
D.2.4 Software/Program Verification
I.1.2 Algorithms
I.2.8 Problem Solving, Control Methods, and Search

Keywords:
Model checking, deterministic planning, algorithm, planning service

Implementation Software and Hardware:
PAT 3.3.0, NuSMV 2.5.2, Spin 6.0.1, Metric-FF, SatPlan2006, Windows XP SP3, Ubuntu 10.04, Microsoft Visual Studio 2008, 32-bit Intel PC

Acknowledgement

First and foremost, I would like to express my gratitude to my supervisor, Dr. Dong Jin Song for his unreserved encouragement and guidance. I thank him for providing me valuable advices and great opportunities during the last two years of my undergraduate study. He has helped me start doing research and fall in love with it.

I also want to thank Dr. Chen Chunqing, and Ms. Zheng Manchun for their support and suggestions on my thesis. I am heartily thankful to Dr. Liu Yang, whose guidance from the initial to the final level enabled me to develop an understanding of the subject.

I would like to thank my fellow teammates in the ICSE 2011 SCORE contest, Mr. Yang Hang and Mr. Wu Huanan for the days and nights we worked together. The "Transport4You" system would not have been possible without their efforts.

Lastly, I wish to thank sincerely and deeply my friends and families. Without them, I would not have been able to complete this thesis.

List of Figures

List of Tables

Table of Contents

Chapter 1

Introduction

Model checking is traditionally used as an automatic technique for verifying critical software and hardware systems. The system model is constructed by describing all dynamic behaviours using precise formal model description languages. The model is then exhaustively explored and checked by model checkers to ensure that desired properties are guaranteed in all cases. Recently, several papers show that model checking can also be applied to AI planning domain. Some experiment results indicate that the performance of model checkers are comparable to that of some planners and the performance of model checking can even be further improved by exploiting domain-specific knowledge.

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.

1.1 Two Problems

In this thesis, we consider two separate problems, 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

of connecting them together such that the tools that support model checking can also be used to find solutions for planning problems.

1.1.1 Model Checking

Model checking is an automatic technique for verifying models of software or hardware systems against their specification (Peled et al., 2009). The system under consideration is formally specified using suitable model description language that describes the behaviours of each component, and this specification is commonly referred to as a *system model*. In general, what we care the most about the system model is whether some safety properties, usually described in temporal logics such as *Linear Temporal Logic (LTL)* and *Computation Tree Logic (CTL)*, are satisfied. A safety property can be casually understood as "something bad never happens". Given a system model $\mathcal{M}$, an initial state s, and a formula φ which specifies the safety 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.

Typically, a counterexample is given by model checkers when the safety property is found to be violated. A counterexample is a finite path π that leads to the "bad thing" 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.

1.1.2 Planning

In this thesis, we only consider classical planning problems which have only deterministic actions and assume complete information about the planning states. Essentially following (Russell and Norvig, 2010), we define a classical planning problem to be a three-tuple (S_0, G, A) where S_0 represents the initial state, G represents the set of goal states and A represents a finite set of deterministic actions. Each *state* is represented as a conjunction of fluents that are ground, functionless atoms. Each *action* $a \in A$ itself is described by a tuple $(pre(a), add(a), del(a))$ where $pre(a)$ represents the precondition to be satisfied before the action can be executed, $add(a)$ and $del(a)$ represent the positive and negative

effects after the action is executed. Therefore the state resulting from executing action a in state s can be expressed as $Result(s, a) = (s - del(a)) \bigcup add(a)$. Finally, the goal G is a set of planning states satisfying a propositional property specifying the final states of a plan. Therefore, a plan p is a finite sequence of actions $\langle a_0, a_1, ..., a_n \rangle$, such that the execution of p yields a state $s \in G$.

The Planning Domain Definition Language (PDDL) (McDermott, 1998) is currently the standard language for representing classical planning problems and is widely used by many planners. Actions are grouped as a set of action schemas in PDDL. The schema consists of the action name, a list of all the variables used in the schema, a precondition and an effect.

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

Figure 1.1: PDDL action schema for taking bus

The PDDL code in Figure 1.1 is an example of an action schema for taking a bus from a bus stop *from* to another bus stop *to*. The precondition for the action schema is that both the bus and the passenger are at *from* and the effect is that they are transferred to a new location *to*. In the later extensions of PDDL, such as PDDL 2.1 where typing system is added, the type predicates like *Bus(b)* is not needed anymore. PDDL 2.1 also allows for optimization criteria to be specified. The optimization criterion, also called *plan metric*, consists of numerical expressions to be maximized or minimized.

1.2 Our Solution

Clearly, a classical planning problem can be easily converted into a model checking problem. The fact that this approach is feasible is supported by (Giunchiglia and Traverso,

2000), which state that, planning should be done by semantically checking the truth of a formula, planning as model checking is conceptually similar to planning as propositional satisfiability. Given a planning problem (S_0, G, A), one can construct a system model $\mathcal{M}$ by translating every action $a \in A$ into a corresponding state transition function first. The initial state S_0 can also be mapped to the initial state s of model $\mathcal{M}$ by assigning value to each variable accordingly. Then for the goal state G, which can be expressed using a propositional formula φ, we can construct a safety property $\mathbf{G}\neg\varphi$ that requires the formula φ never to hold, such that the model checker is able to search for a counterexample path that leads to a state where φ holds. The resulting plan is optimal in terms of make-span when the counterexample path is the shortest. We shall discuss the detailed translation process in Chapter 4.

1.3 Thesis Overview

This thesis is divided into two parts, corresponding to the two separate research problems that we considered: *planning via model checking* and *PAT as planning service*. Part I consists of a thorough presentation and evaluation of our approach of using model checker to find solutions for classical AI planning problems. We start in Chapter 2 by giving detailed background information on the topic, including the related works have been done and the descriptions of various tools we used for the experiments. Chapter 3 presents the methodologies and results of the experiments as well as the analysis. Taking PAT as an example, we then introduce the process of translating PDDL domain descriptions to system models that are recognized by model checkers in Chapter 4.

In Part II, we start from a case study of a transportation management system in Chapter 5. Formal methods are extensively applied in the developing process of the system including *planning via model checking*. A new PAT model checking module is developed to support the system as a planning service. Different model checking algorithms designed for the module are compared and analysed in Chapter 6. Finally, we conclude the discussion in Chapter 7 by summarizing the contributions and observations we made.

Part I

Planning via Model Checking

Chapter 2

Background

We start this chapter by first looking at some related research that has been done on this topic. Readers already familiar with *model checking* and *planning* may still find it useful, as new issues arise when we put together and compare the two problems. This chapter also introduces terminology and techniques that will be used extensively in later chapters.

2.1 Related Work

In a research paper (Berardi and Giacomo, 2000), the authors compared the performance of two well-known model checkers, Spin and SMV, with some state-of-the-art planners (IPP (Koehler et al., 1997), which was one of the best performers in AIPS98 competition; FF (Hoffmann and Nebel, 2001), which was among the best performers in AIPS00; and TLPLAN (Bacchus et al., 2000), 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 AIPS98 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.

In another research paper (Hörne and van der Poll, 2008), the authors 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, an extension of the symbolic model checker SMV. With NuSMV the problem is represented using Binary Decision Diagrams (BDDs) (Bryant, 1992). 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.

2.2 PAT

Process Analysis Toolkit (PAT) (Sun et al., 2009) 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 automated refinement checking, model checking of LTL extended with events and various ways of system simulations. PAT is designed to verify event-based compositional models specified using CSP#. CSP# is an extension to Communicating Sequential Process (CSP) (Hoare, 1978) by embedding data operations. It combines high-level compositional operators from process algebra with program-like codes, which makes the language much more expressive.

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

PAT is designed as an flexible and modularized framework. It allows users to build

customized model checking modules easily. The language syntax, semantics, model checking algorithms, reduction techniques, and abstraction techniques can all be tailored for a specific domain. We shall explore this feature later in Chapter 6 to customize searching algorithms for planning purpose. PAT also has a more user-friendly user interface both for verification and simulation compared with the tools that we will look at in the later sections.

2.3 NuSMV

NuSMV is an extension of the symbolic model checker SMV (McMillan, 1992) 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 (Cavada et al., 2005).

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 (Cavada et al., 2005). However, array indices in NuSMV must be statically evaluated to integer constants. This constraint largely limits the expressiveness of the model. The modelling for common operations on a list of state variables is sometimes cumbersome in NuSMV. In general, such operations have to be manually coded by enumerating all the possible cases.

The descriptions of transition relations between the current and next state pairs can be done by either using the `ASSIGN` constraint where a system of equations labelled as `next(identifier):=expression` describing how the FSM evolves over time, or the `TRANS` constraint (Cavada et al., 2005). Specifications can be expressed in both CTL and LTL. NuSMV supports several kinds of model checking modes, namely CTL checking,

LTL checking, invariant checking and bounded model checking. We will compare the performance of using different model checking modes for planning in Chapter 3.

2.4 Spin

Spin is an established explicit state model checker developed at Bell Labs in the original Unix group of the Computing Sciences Research Center, starting in 1980. Spin models are described in a modelling language called "Promela" (Process Meta Language). The language allows for the dynamic creation of concurrent processes. Communication via message channels can be defined to be synchronous or asynchronous (Holzmann, 2003). Promela loosely follows CSP and hence our models in CSP# can be converted to it with minimal efforts. Guarded expressions are well supported, so that preconditions for actions can be easily enforced in the model. Promela also allows C-style macro definitions, which reduces the code length and facilitates the generalization of the model.

Spin has a number of runtime options for simulation as well as verification that can be explored. The maximum search depth can be adjusted according to the size of the model. Spin also allows users to prune the search space using "never-claims" which are equivalent to safety properties. With this method it becomes possible to verify quickly whether a given safety property holds in the context of the model, even when a complete verification is considered to be infeasible (Holzmann, 2003). After verification is finished, Spin is able to perform a simulation guided by the error trail. In simulation mode, step-by-step display of the counterexample trace is better supported by its user interface compared with that of NuSMV.

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

2.5 SatPlan

SatPlan is an award winning planner for optimal planning created by Prof. Henry Kautz, Dr. Jörg Hoffmann and Shane Neph. SatPlan2004 takes the first place for optimal deterministic planning at the International Planning Competition at the 14th International Conference on Automated Planning & Scheduling. SatPlan accepts the STRIPS subset of PDDL and finds plans with the shortest make-span. It encodes the planning problem into a SAT formulation with length k and check the satisfiability using a SAT solver. If the searching times out, then k is increased by one and the process is repeated (Kautz et al., 2006).

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

2.6 Metric-FF

Metric-FF (Hoffmann, 2002) is a domain independent planning system developed by Dr. 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. As mentioned, PDDL 2.1 allows numerical plan metrics. Figure 2.1 shows an example of plan metrics used in domain descriptions.

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

Figure 2.1: PDDL 2.1 action schema for taking bus with plan metrics

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

Chapter 3

Experimental Results

In this Chapter, we compare the performance of PAT (3.3.0 academic version), NuSMV (pre-compiled version 2.5.2), Spin (pre-compiled version 6.0.1) 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 following.

- *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 which is the border between the two countries at war. The bridge has been damaged and can only carry two soldiers at a time. Furthermore, several landmines have been placed on the bridge and a torch is needed to sidestep all the mines. The enemy is on their tail, so the soldiers know that they have only 60 minutes to cross the bridge. The soldiers only have a single torch and they are not equally injured. The extent of their wounds have an effect on the time it takes to get across. So the time needed for each soldier are 5, 10, 20, 25 minutes respectively. The goal is to find a solution to get all the soldiers to cross the bridge to safety in 60 minutes or less.

- *The sliding game problem*, is sometimes also referred as *the eight-tiles problem*. We have eight tiles, numbered from 1 to 8, that are arranged in a 3×3 matrix. The first tile, which is at the top-left corner is empty and marked by 0. A tile can only

be shifted horizontally or vertically into the empty space. The goal of the puzzle is to arrange the eight tiles into the setting shown in Figure 3.1.

0	1	2
3	4	5
6	7	8

Figure 3.1: Initial setting of *the sliding game problem*

3.1 Methodology

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 has an option named "Shortest Witness Trace" in the verification window. When this option is selected, PAT performs a breadth-first search in the state space, and the returned counterexample trace is guaranteed the shortest. Otherwise, a depth-first search is performed and the first counterexample trace encountered is displayed. Therefore, for *the bridge crossing problem* where shortest witness trace is not needed, we used the depth-first search mode; for *the sliding game problem*, for which an optimal solution is expected, we enabled the "Shortest Witness Trace" option instead. The counterexample provided by NuSMV is always shortest, so it can also be used to generate optimal solutions for *the sliding game problem*. Unfortunately, as mentioned, the counterexample produced by Spin is not always shortest. However, we still collected the performance data for reference.

To generalize the problem and get the experimental results in a broader range, we expanded the original *bridge crossing problem* to versions with up to 9 soldiers. Except the breadth-first and depth-first search, PAT also supports "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 also was also run one more time using the depth-first search mode. We also ran Metric-FF on *the bridge crossing problem* with parameters $g = 100$ and $h = 1$, which emphasises the plan quality over the performance to increase the possibility of getting an solution within the time limit.

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 (Gregory et al., 2007). 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 (Reinefeld, 1993). 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.

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. To run Spin, we used the Unix simulator Cygwin. Spin displays the execution time in a separate window using an embedded Tcl/Tk environment. PAT and NuSMV were tested in Windows XP SP3, while SatPlan and Metric-FF were tested in Ubuntu 10.04 environment. Except for NuSMV, all other tools provide accurate statistics including the execution time at the end of each session. For NuSMV, we made use of the *source* command to invoke the *time* command right before and after the model checking sessions to record the execution time. Unfortunately, the *time* command in NuSMV provides time data that is accurate to only one decimal place. On contrast, execution time data getting from other tools was rounded to two decimal places. All the experimental results were collected on an Dell desktop with an Intel Core 2 Duo E6550

2.33GHz processor and 3.25GB RAM.

3.2 Results

In this section we presents the experimental results. In the following tables, INVAR denotes using invariant mode of NuSMV, LTL/CTL denotes using LTL/CTL model checking mode of NuSMV, WITH denotes PAT under "reachability-with" mode, and DFS/BFS denotes PAT using depth-first/breadth-first search. Time is in seconds unless otherwise indicated.

3.2.1 The Bridge Crossing Problem

This set of experiments are tailored to show how the model checkers compete on plan existence problems that deal with time constraints. The time cost of each soldier is listed in Table 3.1 below.

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

Table 3.1: Time cost of each soldier

The results are summarized in Table 3.2. 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 getting form Metric-FF is still not guaranteed. So the data is only used as a benchmark for comparisons.

#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

Table 3.2: Experimental results for *the bridge crossing problem*

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. With regard to Temporal model checking in NuSMV, the performance is better using LTL than CTL.

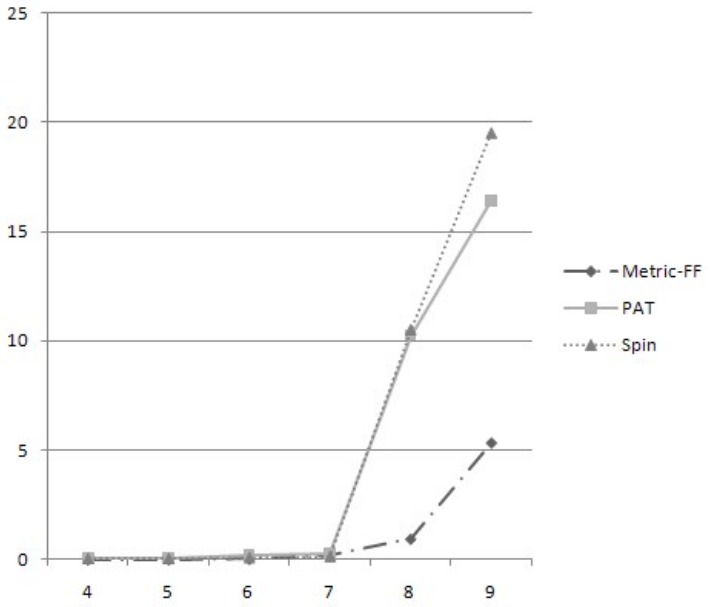


Figure 3.2: Execution time comparison of PAT, Spin and Metric-FF on *the bridge crossing problem*

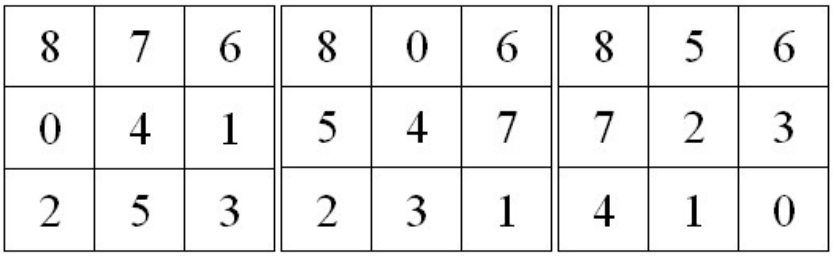
Figure 3.2 shows that the time needed for *the bridge crossing problem* increases rapidly when the number of soldiers increases. For example, the execution time for Spin increases

by nearly 100 times when the number of soldiers increases from 7 to 8. It is clear that the state space expands in a very fast speed. Planners such as Metric-FF handle this kind of problem in a very different way from model checkers. Metric-FF performs a standard weighted A* search which exploits the power of heuristics and sacrifices the optimality to speed up the searching. That is the reason why Metric-FF performs much better than the other two.

The performance of PAT and Spin is similar on this problem domain. For smaller instances, for example, when the number of soldiers ranges from 4 to 7, Spin performs better than PAT, although the difference is relatively small. For larger instances like the problem with 8 or 9 soldiers, PAT starts to perform better that Spin.

3.2.2 The Sliding Game Problem

This set of experiments are designed to show how different model checkers perform on optimal deterministic planning problems. The results getting from SatPlan are used for reference. The initial configurations of all the six problem instances are shown in Figure 3.3.



(a) Hard1 (b) Hard2 (c) Most1

(d) Most2 (e) Rand1 (f) Rand2

Figure 3.3: Initial configurations of *the sliding game problem* instances

The results are summarized in Table 3.3. 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.

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

Table 3.3: Experimental results for *the sliding game problem*

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.

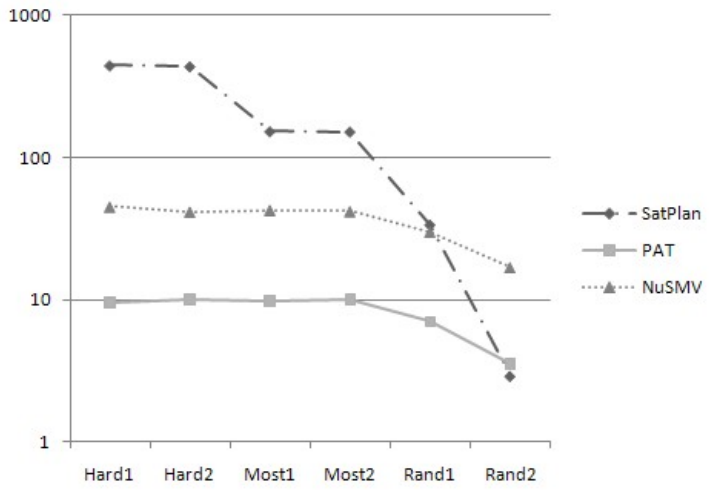


Figure 3.4: Execution time comparison of PAT, NuSMV and SatPlan on *the sliding game problem*, shown on a logarithm scale

From Figure 3.4 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 that belongs to the category of explicit state model checkers performs better than symbolic model checker NuSMV and SAT based planner SatPlan on plan optimization problems. Although we cannot generalize the argument without further experiments and justifications, this empirical finding still proves the feasibility of applying PAT to the optimal deterministic planning domain.

Chapter 4

From PDDL to CSP#

In this Chapter, we describe the source-to-source translation from PDDL to CSP#. Our goal is to formulate some casual rules that can act as a guide when doing the translation. The translation is based on two basic assumptions:

- The PDDL domain descriptions are written in the subset of PDDL 2.1 that includes STRIPS-like operators with literals having typed arguments and numerical plan metrics. The typing can be easily done by hand or a tool such as TIM (Fox and Long, 1998) when the original model is written without typed arguments.
- The translation should keep, as far as possible, the naming as well as the structures of the original PDDL domain descriptions.

In the following sections, we shall explain the translation process using a running example, *the bridge crossing problem* that was used in our experiments.

4.1 Typing

PDDL has a special syntax for declaring object and parameter types. If types are to be used in a domain, the domain file should include a declaration: (: `types NAME1` ... `NAME_N`). Figure 4.1 shows the typing declaration for *the bridge crossing problem*.

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

Figure 4.1: PDDL typing declaration in the domain definition file for *the bridge crossing problem*

We use a hierarchical typing system, where *place* and *locatable* are of primitive type *object*, while *soldier* and *torch* belong to *locatable*. After the declaration of types in the domain definition file, objects can be defined with types in the problem description file. Figure 4.2 shows the object definitions for *the bridge crossing problem*.

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

Figure 4.2: PDDL object definitions in the problem description file for *the bridge crossing problem*

Object *torch* and *soldier0...3* are of *locatable* type, while *north* and *south* are of *place* type. For every group of objects of the same type, we declare a constant enumeration in CSP# as shown in Figure 4.3.

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

Figure 4.3: CSP# enumeration declaration for *the bridge crossing problem*

4.2 Predicates

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

1. *void setPredicate(predicateName, x, y, value);*

2. *bool tryPredicate(predicateName, x, y);*

3. *int snapShot();*

The example above only shows the methods for predicates with an arity of two. *setPredicate* is used to set the value of a predicate with its name specified as the first parameter. *tryPredicate* returns the value of a predicate and *snapShot* returns an integer that represents the current snapshot of the predicate database. To use the self-defined data-type, the C# library has to be imported and instantiated first. For each predicate declared in the domain definition, we also need a corresponding enumeration type in CSP# as shown in Figure 4.4 and Figure 4.5.

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

Figure 4.4: PDDL predicate declaration for *the bridge crossing problem*

```
#import "Predicate";
var < Predicate > pre = new Predicate();

enum {At};
```

Figure 4.5: CSP# predicate declaration for *the bridge crossing problem*

4.3 Initial State

Figure 4.6 and Figure 4.7 show the translation of initial state specifications from PDDL to CSP#. For the functions `time` and `time-cost` in PDDL, we simply use an integer array and an integer variable in CSP# to represent them respectively. The initialization of predicates are done within the process *ini()*.

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

Figure 4.6: PDDL initial state specifications for *the bridge crossing problem*

$$
\begin{array}{l}
var\ time[4] = [5, 10, 20, 25]; \\
var\ time\_cost\ =\ 0; \\
\\
ini()\ =\ initial\{pre.setPredicate(At, soldier0, south, true); \\
\qquad\qquad pre.setPredicate(At, soldier1, south, true); \\
\qquad\qquad pre.setPredicate(At, soldier2, south, true); \\
\qquad\qquad pre.setPredicate(At, soldier3, south, true); \\
\qquad\qquad pre.setPredicate(At, torch, south, true)\}\ \rightarrow\ Skip;
\end{array}
$$

Figure 4.7: CSP# initial state specifications for *the bridge crossing problem*

4.4 Actions

With object types and predicates ready, the translation of actions is straightforward. The preconditions are translated as guard conditions of processes. The effects are translated as statement blocks after event names. Conditional effects can also be easily converted into conditional branches that are well supported in CSP#. The updates for plan metrics can be mapped into simple data operations. Figure 4.8 and Figure 4.9 show the semantically equivalent action definitions for "south to north" in PDDL and CSP# respectively.

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

Figure 4.8: PDDL action definition "south to north" for *the bridge crossing problem*

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

Figure 4.9: CSP# action definition "south to north" for *the bridge crossing problem*

In Figure 4.9, *x!=y* in the guard condition is to ensure the two parameters x and y are distinct, which is implicitly enforced in PDDL. To establish the state transition system, we also need another process to choose among different actions. As is shown in Figure 4.10, the process *Trans()* first makes a snapshot of the current predicate database, then nondeterministically chooses one action and proceeds. The parameters for the actions are also nondeterministically chosen among the available objects that are of the suitable types. This is done by using the syntax sugar "indexed event list" that takes in parameters within the corresponding enumeration range.

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

Figure 4.10: CSP# *Trans()* process for *the bridge crossing problem*

4.5 Goal

The goal of a PDDL problem description contains logic formulas of predicates and possibly also specifies the plan optimization criteria. The translation of optimization criteria can be achieved by using the keyword "*reaches ... with ...*". Figure 4.11 and Figure 4.12 show the corresponding goal states definitions of PDDL and CSP#.

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

Figure 4.11: PDDL goal definitions for *the bridge crossing problem*

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

$$
\#assert\ Plan\ reaches\ goal\ with\ min(time\_\ cost);
$$

Figure 4.12: CSP# goal definitions for *the bridge crossing problem*

Clearly, using the newly defined $\langle Predicate \rangle$ type, the translation from PDDL to CSP# is intuitive as far as we can see in this example. Translation tools can even be developed to automate the process. However, the translated CSP# model uses a number of complex data structures and language constructs. The performance is not as good as hand coded ones that use only primitive data types and are specifically optimized for

their own purposes. More research should be done no this topic in the future to improve the translation accuracy and efficiency.

Part II

PAT as Planning Service

Chapter 5

Case Study: Transport4You

When doing the experiments in Part I, we also felt that the generalization of the problems should be a priority because the coding 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.

In this chapter, we complete a case study on "Transport4You" which is a project submission by us for the 33^{rd} International Conference on Software Engineering (ICSE) - Student Contest on Software Engineering (SCORE). The project is already selected as one of the finalists (5 out of 56 submissions) which are going to be 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 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 Figure 5.1.

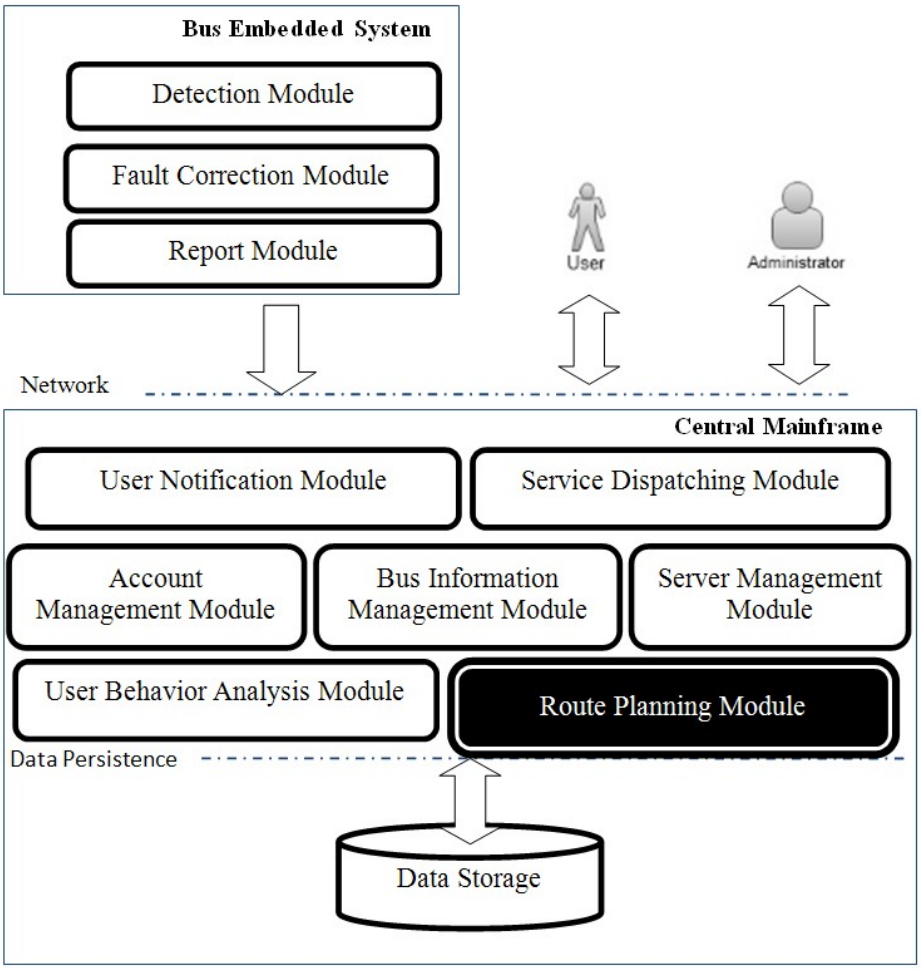


Figure 5.1: System architecture diagram of the "Transport4You" IPTM system

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 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.

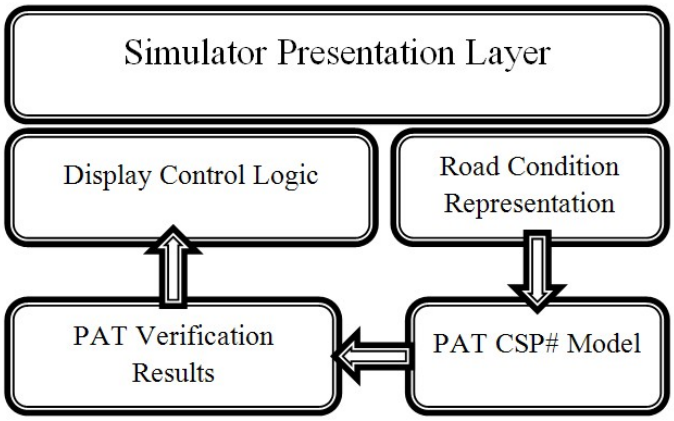


Figure 5.2: Simulator architecture diagram

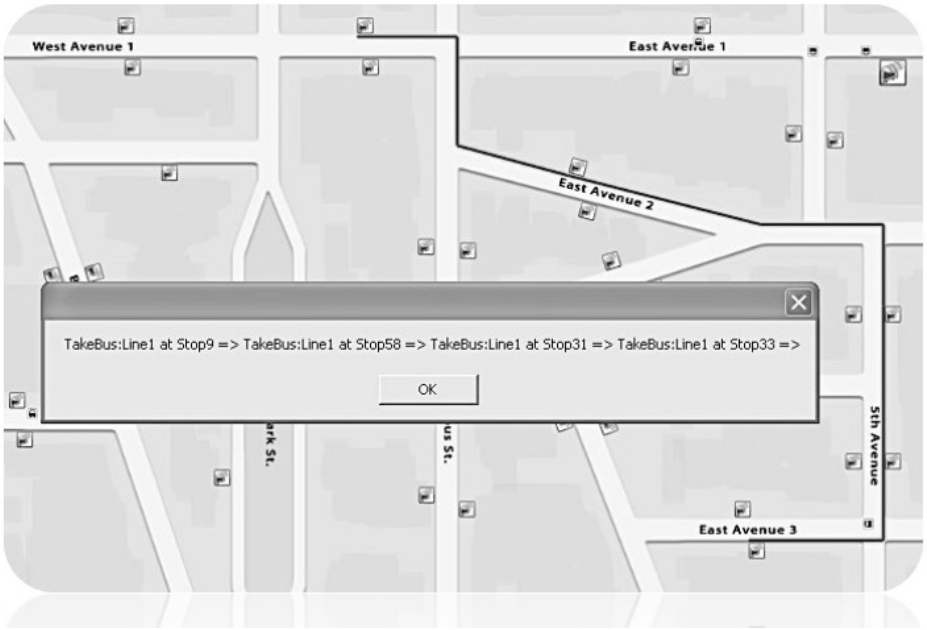


Figure 5.3: Simulator screen shot of route planning results

As is shown in Figure 5.2 and Figure 5.3, 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. 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.

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. Of course, we can also build our own planning modules with customized searching algorithms. We shall further discuss this in Chapter 6.

Chapter 6

Route Planning Model Design

In this chapter, 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.

6.1 Basic Model

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 terminal $s_{start} \subseteq S$, and end terminal $s_{end} \subseteq S$, 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., $B_S = \{b_i \in B \mid s \in dom(b_i)\}$.*

- $c : S \rightarrow S$ *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 user is at 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.

6.1.1 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 ... 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<BusLine> Line1 = new BusLine(sLine1,1);
var sLine2 = [TerminalC, Stop56, Stop52, Stop32, Stop30, Stop59, Stop10, Stop8, Stop6, TerminalA];
var<BusLine> Line2 = new BusLine(sLine2,2);
...
...
var sLine14 = [TerminalC, Stop34, Stop32, Stop30, Stop16, TerminalB];
var<BusLine> 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.

6.1.2 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, while *currentBus* is only for record in the current model.

//Initial State
var currentStop = Stop5;
var B0 = [-2];
var<BusLine> 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.

6.1.3 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 key word "*case*". The key word is used as the following,

case{
cond1: P1
cond2: P2
default: P
}

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;} ->takeBus();

...

...

BusLine14=

TakeBus.14{currentStop=Line14.NextStop(currentStop); currentBus=Line14;} ->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: crosscurrentStop=Stop6->takeBus()

currentStop==Stop7: crosscurrentStop=Stop8->takeBus()

...

...

currentStop==Stop35: crosscurrentStop=Stop34->takeBus()

currentStop==Stop34: crosscurrentStop=Stop35->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 return 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.

6.1.4 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 user that is *Stop53* in our example.

//Goal States
#define goal currentStop==Stop53;

6.2 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 occurs. 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 ← EvaluateExpression(current);
  if current.ImplyCondition() 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 are usually small, the execution time is also relatively short. Unfortunately, "reachability-with" checking in the "CSP Module" 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 designed 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. 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 Algorithm 1.

6.3 Search Space Pruning

As mentioned in the previous section, one of the causes for producing suboptimal solutions is that the number of bus changes is uncontrolled. Taking an example shown in Figure 6.1, 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, $TakeBus(b_1, s_1)$ refers to the solid edge between s_1 and s_2.

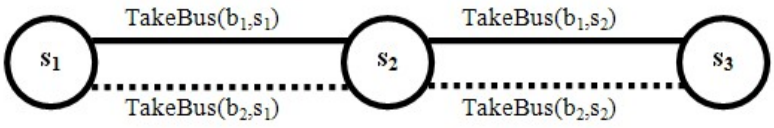


Figure 6.1: An example bus line configuration

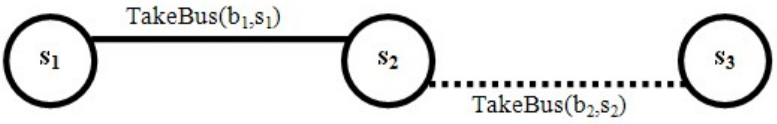


Figure 6.2: A solution produced by the basic model

As illustrated in Figure 6.2, the basic model produces unsatisfactory solutions when there obviously exists better ones. The partial solution "$TakeBus(b_1, s_1) \Rightarrow TakeBus(b_2, s_2)$"

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 $\langle BusLine \rangle$. In the guard condition of process *BusLine2*, adding *!Line2.IsRedundent(currentBus, currentStop)* will avoid this transition if the change to *Line2* is considered redundant.

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 3 *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 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 6.3 and shares the same previous stop or next stop with the current bus, the change is considered redundant.

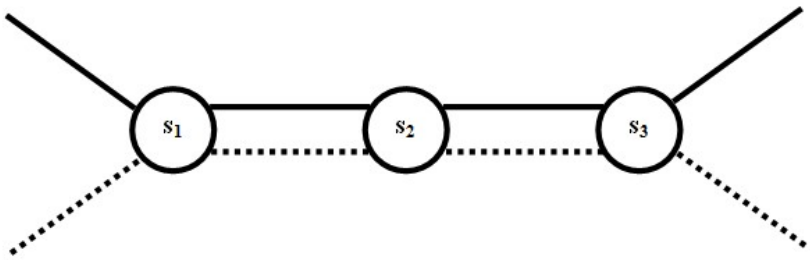
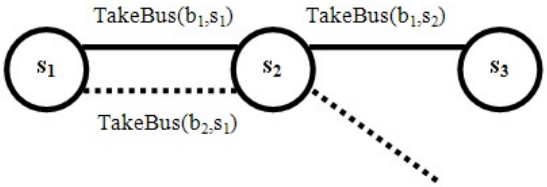


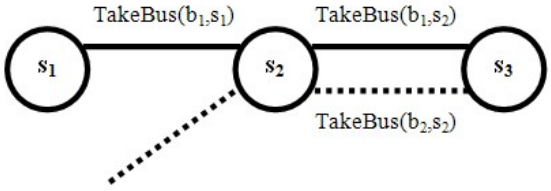
Figure 6.3: Special pattern of two overlapping bus lines

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 Figure 6.4a, the partial solution

"$TakeBus(b_2, s_1) \Rightarrow TakeBus(b_1, s_2)$" is not valid as at s_2, b_1 and b_2 have the same previous stop s_1. A valid path is "$TakeBus(b_1, s_1) \Rightarrow TakeBus(b_1, s_2)$". 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 is shown in Figure 6.4b, the partial solution "$TakeBus(b_1, s_1) \Rightarrow TakeBus(b_2, s_2)$" is not valid as at s_2, b_1 and b_2 have the same next stop s_3. A valid path is "$TakeBus(b_1, s_1) \Rightarrow TakeBus(b_1, s_2)$". However, after enforcing these two basic rules, the transition can never happen between two lines forming the special pattern illustrated in Figure 6.3. 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.



(a) Same Previous Stop



(b) Same Next Stop

Figure 6.4: Redundant bus changes

6.4 Performance Comparison

In this section, we compare 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 get 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 6.1 shows the comparison results.

	States	Transitions	Time(s)	Memory(KB)	Cost	Cost(s)	Length	Length(s)
Basic	1029.46	1070.93	0.0448	11119.91	58.23	1254	5.51	0
Cost	1125.31	1169.82	0.0483	11281.58	56.02	0	5.59	247
Prune	158.48	185.77	0.0179	9197.95	56.79	379	5.51	0

Table 6.1: Comparison results of three route planning models

In the table, all values are average among the 3660 test cases except "Cost(s)" and "Length(s)" which indicate the number of solutions with suboptimal total cost and suboptimal length respectively. From the comparison, we can see that the *search space pruning* model performs the best in terms of execution time and memory space. In fact, a large portion of redundant transitions are 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.

Chapter 7

Conclusion and Future Work

7.1 Summary

In this thesis, we focused on an attempt of using model checking techniques on AI planning domain. We believe our research effort to be a good start in this direction towards more practical applications.

In the first part, we 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, NuSMV and Spin. PAT is proved to be the most suitable one for solving various kind of planning problems. The experimental results also indicate that some model checkers can even compete with sophisticated planners in certain domains. We also suggested a way of translating PDDL to CSP#, which may serve as a basis for developing automated translation tools.

In the second part, we analysed 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. Then following the formal definitions of the route planning problem, we designed a basic CSP# model. We further improved the model in two ways. One of them is introducing cost function for measuring plan quality, while the other approach is adding in domain specific control knowledge for search space pruning. In the end, we compared the different approaches we attempted on their time and memory efficiency as well as their plan quality.

7.2 Recommendations for Future Work

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

References

Bacchus, F., Kabanza, F., and Sherbrooke, U. D. (2000). Using temporal logics to express search control knowledge for planning. *Artificial Intelligence*, 16:123–191.

Berardi, D. and Giacomo, G. D. (2000). Planning via model checking: Some experimental results. unpublished manuscript.

Bryant, R. E. (1992). Symbolic boolean manipulation with ordered binary-decision diagrams. *ACM Computing Surveys*, 24:293–318.

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

Fox, M. and Long, D. (1998). The automatic inference of state invariants in TIM. *Journal of Artificial Intelligence Research*, 9:367–421.

Giunchiglia, F. and Traverso, P. (2000). Planning as model checking. In Biundo, S. and Fox, M., editors, *Recent Advances in AI Planning*, volume 1809 of *Lecture Notes in Computer Science*, pages 1–20. Springer Berlin / Heidelberg.

Gregory, P., Long, D., and Fox, M. (2007). A meta-CSP model for optimal planning. In *Proceedings of the 7th International conference on Abstraction, reformulation, and approximation*, SARA'07, pages 200–214, Berlin, Heidelberg. Springer-Verlag.

Hoare, C. A. R. (1978). Communicating Sequential Processes. *Communications of the ACM*, 21(8):666–677.

Hoffmann, J. (2002). Extending FF to numerical state variables. In *Proceedings of the 15th European Conference on Artificial Intelligence (ECAI-02)*, pages 571–575, Lyon, France.

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

Holzmann, G. J. (2003). *The SPIN Model Checker: Primer and Reference Manual*. Addison-Wesley Professional.

Hörne, T. and van der Poll, J. A. (2008). 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, pages 114–123, New York, NY, USA. ACM.

Kautz, H. A., Selman, B., and Hoffmann, J. (2006). SatPlan: Planning as satisfiability. In *Abstracts of the 5th International Planning Competition*.

Koehler, J., Nebel, B., Hoffmann, J., and Dimopoulos, Y. (1997). Extending planning graphs to an ADL subset. pages 273–285. Springer-Verlag.

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

McMillan, K. L. (1992). *Symbolic Model Checking: an approach to the state explosion problem*. PhD thesis, Carnegie Mellon University.

Peled, D., Pelliccione, P., and Spoletini, P. (2009). *Wiley Encyclopedia of Computer Science and Engineering*, chapter Model Checking. John Wiley & Sons.

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

Russell, S. and Norvig, P. (2010). *Artificial Intelligence*, chapter Classical Planning. Pearson.

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

Appendix A

Selected Model Source Code

A.1 CSP# Model for *the sliding game problem*

1. $//Initial -\rightarrow Goal$
2. // 0 1 2 | 3 5 6
3. // 3 4 5 | 0 2 7
4. // 6 7 8 | 8 4 1
5. $\#define\ N\ 9;$
6. $var\ board\ =\ [3,5,6,0,2,7,8,4,1];$
7. $hvar\ emptypos\ =\ 3;$

8. $Game() = Left()\ \square\ Right()\ \square\ Up()\ \square\ Down();$

9. $Left() = [emptypos! = 2 \&\& emptypos! = 5 \&\& emptypos! = 8]goleft$
10. $\{board[emptypos] = board[emptypos + 1];\ board[emptypos + 1] = 0;$
11. $emptypos = emptypos + 1\}$
12. $\rightarrow\ Game();$
13. $Right() = [emptypos! = 0 \&\& emptypos! = 3 \&\& emptypos! = 6]goright$
14. $\{board[emptypos] = board[emptypos - 1];\ board[emptypos - 1] = 0;$
15. $emptypos = emptypos - 1\}$
16. $\rightarrow\ Game();$
17. $Up() = [emptypos! = 6 \&\& emptypos! = 7 \&\& emptypos! = 8]goup$
18. $\{board[emptypos] = board[emptypos + 3];\ board[emptypos + 3] = 0;$
19. $emptypos = emptypos + 3\}$
20. $\rightarrow\ Game();$
21. $Down() = [emptypos! = 0 \&\& emptypos! = 1 \&\& emptypos! = 2]godown$
22. $\{board[emptypos] = board[emptypos - 3];\ board[emptypos - 3] = 0;$
23. $emptypos = emptypos - 3\}$
24. $\rightarrow\ Game();$

25. $\#assert\ Game()\ reaches\ goal;$
26. $\#define\ goal\ (\&\&\ i : \{0..N-2\}@(board[i] == i + 1))\ \&\&\ board[N-1] == 0;$

A.2 CSP# Model for *the bridge crossing problem*

```
1. #define Max 60;
2. #define N 4;
3. #define North 0;
4. #define South 1;

5. var val[9] = [5, 10, 20, 25, 30, 45, 60, 80, 100];
6. var At_ North[9] = [1, 1, 1, 1, 1, 1, 1, 1, 1];
7. var At_ South[9] = [0, 0, 0, 0, 0, 0, 0, 0, 0];
8. var time = 0;
9. var torch : {0..1} = North;
10. var north1;
11. var north2;
12. var south1;

13. NtoS() =□ x : {0..N − 1}@
14.         [At_ North[x] == 1]Select_ North1.x{At_ North[x] = 0; north1 = x} → Skip;
15.         (□ y : {0..N − 1}@
16.         [At_ North[y] == 1]Select_ North2.y{At_ North[y] = 0; north2 = y} → Skip);

17. StoN() =□ x : {0..N − 1}@
18.         [At_ South[x] == 1]Select_ South1.x{At_ South[x] = 0; south1 = x} → Skip;

19. soldier() = if(torch == North){NtoS()}else{StoN()};

20. Cross() = if(torch == North){cross_ north{
21.                 if(val[north1] > val[north2]){
22.                     time = time + val[north1];
23.                     At_ South[north1] = 1;
24.                     At_ South[north2] = 1; }
25.                 else{
26.                     time = time + val[north2];
27.                     At_ South[north1] = 1;
28.                     At_ South[north2] = 1; }
29.                 torch = 1; } → Skip}
30.         else{cross_ south{
31.                   time = time + val[south1];
32.                 At_ North[south1] = 1;
33.                 torch = 0; } → Skip};

34. Game() = if(time > Max){Skip}
35.         else{soldier(); Cross(); Game()};

36. ////////////////The Properties/////////////////
37. #define constraint time <= Max;
38. #define goal (&& i : {0..N − 1}@(At_ South[i] == 1)) && constraint;
39. #assert Game() reaches goal;
40. #assert Game() reaches goal with min(time);
```